



ИНСАЙТ ОТ ЭКСПЕРТА

Golang для профи

Работа с сетью, многопоточность,
структуры данных и машинное
обучение с Go

Второе издание



Михалис Цукалос

Packt

Mastering Go

Second Edition

Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures

Mihalis Tsoukalos

Packt>

BIRMINGHAM - MUMBAI

Михалис Цукалос

Golang для профи

Работа с сетью, многопоточность,
структуры данных и машинное
обучение с Go

Второе издание



Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.988.02-018.1

УДК 004.43

Ц85

Цукалос Михалис

Ц85 Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go. — СПб.: Питер, 2020. — 720 с.: ил. — (Серия «Для профессионалов»).

ISBN 978-5-4461-1617-1

Go является языком высокопроизводительных систем будущего. Эта книга показывает, как заставить Go работать на реальных производственных системах.

Для программистов, которые уже знакомы с основами языка Go, эта книга содержит примеры, шаблоны и четкие объяснения, которые помогут вам глубоко понять возможности Go и применить их в своей работе по программированию.

Книга охватывает нюансы Go с подробными руководствами по типам и структурам, пакетам, параллелизму, сетевому программированию, дизайну компиляторов, оптимизации и многому другому. Каждая глава заканчивается упражнениями и ресурсами, чтобы полностью внедрить ваши новые знания.

16+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.1

УДК 004.43

Права на издание получены по соглашению с Packt Publishing. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1838559335 англ.

© Packt Publishing 2019

First published in the English language under the title 'Mastering Go — Second Edition — (9781838559335)'

ISBN 978-5-4461-1617-1

© Перевод на русский язык ООО Издательство «Питер», 2020

© Издание на русском языке, оформление ООО Издательство «Питер», 2020

© Серия «Для профессионалов», 2020

Краткое содержание

Об авторе	21
О научном редакторе.....	22
Предисловие.....	23
Глава 1. Go и операционная система	29
Глава 2. Go изнутри	70
Глава 3. Работа с основными типами данных Go	120
Глава 4. Использование составных типов данных.....	163
Глава 5. Как улучшить код Go с помощью структур данных.....	216
Глава 6. Неочевидные знания о пакетах и функциях Go.....	273
Глава 7. Рефлексия и интерфейсы на все случаи жизни.....	330
Глава 8. Как объяснить UNIX-системе, что она должна делать	364
Глава 9. Конкурентность в Go: горутины, каналы и конвейеры	429
Глава 10. Конкурентность в Go: расширенные возможности	452
Глава 11. Тестирование, оптимизация и профилирование кода.....	510
Глава 12. Основы сетевого программирования на Go	580
Глава 13. Сетевое программирование: создание серверов и клиентов	638
Глава 14. Машинное обучение на Go	683
Что дальше?	718

Оглавление

Об авторе	21
О научном редакторе.....	22
Предисловие.....	23
Для кого предназначена эта книга.....	23
О чем эта книга	23
Как получить максимальную пользу от этой книги.....	26
Загрузите файлы с примерами кода.....	27
Где скачать цветные иллюстрации.....	27
Условные обозначения	27
От издательства	28
Глава 1. Go и операционная система.....	29
История Go	29
Куда движется Go?	30
Преимущества Go	30
Идеален ли Go?	31
Утилита godoc.....	32
Компиляция Go-кода.....	33
Выполнение Go-кода	34
Два правила Go	35
Правило пакетов Go: не нужен — не подключай.....	35
Правильный вариант размещения фигурных скобок — всего один.....	36
Как скачивать Go-пакеты	37

Стандартные потоки UNIX: stdin, stdout и stderr	39
Вывод результатов	39
Использование стандартного потока вывода	41
Получение данных от пользователя.....	42
Что такое := и = ...	42
Чтение стандартного потока ввода	43
Работа с аргументами командной строки	45
Вывод ошибок	47
Запись в журнальные файлы	49
Уровни журналирования	49
Средства журналирования	50
Серверы журналов.....	50
Пример Go-программы, которая записывает информацию в журнальные файлы.....	51
Функция log.Fatal().....	54
Функция log.Panic()	54
Запись в специальный журнальный файл	56
Вывод номеров строк в записях журнала	58
Обработка ошибок в Go	59
Тип данных error.....	59
Обработка ошибок.....	61
Использование Docker	64
Упражнения и ссылки	68
Резюме	69
Глава 2. Go изнутри	70
Компилятор Go	71
Сборка мусора	72
Трехцветный алгоритм.....	74
Подробнее о работе сборщика мусора Go	78
Хеш-таблицы, срезы и сборщик мусора Go.....	79
Небезопасный код	82

Пакет <code>unsafe</code>	84
Еще один пример использования пакета <code>unsafe</code>	84
Вызов С-кода из Go	86
Вызов С-кода из Go в одном файле	86
Вызов из Go С-кода в отдельных файлах	87
С-код	87
Go-код	88
Сочетание кода на Go и С	89
Вызов Go-функций из С-кода	90
Go-пакет	90
С-код	91
Ключевое слово <code>defer</code>	92
Использование <code>defer</code> для журналирования	95
Функции <code>panic()</code> и <code>recover()</code>	97
Самостоятельное использование функции <code>panic()</code>	98
Две полезные UNIX-утилиты	99
Утилита <code>strace</code>	100
Утилита <code>dtrace</code>	101
Среда Go	102
Команда <code>go env</code>	104
Go-ассемблер	105
Узловые деревья	106
Хотите знать больше о <code>go build</code> ?	111
Создание кода WebAssembly	113
Краткое введение в WebAssembly	113
Почему WebAssembly так важен	113
Go и WebAssembly	114
Пример	114
Использование сгенерированного кода WebAssembly	115
Общие рекомендации по программированию на Go	117
Упражнения и ссылки	118
Резюме	118

Глава 3. Работа с основными типами данных Go	120
Числовые типы данных.....	120
Целые числа	121
Числа с плавающей точкой	121
Комплексные числа.....	121
Числовые литералы в Go 2	123
Циклы Go.....	124
Цикл for	124
Цикл while	125
Ключевое слово range.....	125
Пример применения нескольких циклов Go.....	125
Массивы в Go.....	127
Многомерные массивы	128
Недостатки массивов Go	130
Срезы в Go	131
Выполнение основных операций со срезами	131
Автоматическое расширение срезов	133
Байтовые срезы	135
Функция сору()	135
Многомерные срезы.....	137
Еще один пример использования срезов.....	137
Сортировка срезов с помощью sort.Slice()	139
Добавление массива к срезу	141
Хеш-таблицы Go	142
Запись в хеш-таблицу со значением nil	144
Когда использовать хеш-таблицы	145
Константы Go	145
Генератор констант iota	147
Указатели в Go	149
Зачем нужны указатели	152
Время и дата	152
Работа с временем.....	153

Синтаксический анализ времени.....	154
Работа с датами.....	156
Синтаксический анализ дат.....	156
Изменение формата даты и времени.....	157
Измерение времени выполнения программы.....	159
Измерение скорости работы сборщика мусора Go.....	160
Веб-ссылки и упражнения.....	161
Резюме.....	161
Глава 4. Использование составных типов данных.....	163
Составные типы данных.....	164
Структуры.....	164
Указатели на структуры.....	166
Ключевое слово new.....	168
Кортежи.....	169
Регулярные выражения и сопоставление с образцом.....	170
Немного теории.....	171
Простой пример.....	171
Более сложный пример.....	174
Проверка IPv4-адресов.....	176
Строки.....	180
Что такое руны.....	183
Пакет unicode.....	184
Пакет strings.....	185
Оператор switch.....	189
Вычисление числа π с высокой точностью.....	192
Разработка на Go хранилища типа «ключ — значение».....	195
Go и формат JSON.....	200
Чтение данных из формата JSON.....	200
Сохранение данных в формате JSON.....	202
Использование функций Marshal() и Unmarshal().....	203
Синтаксический анализ данных в формате JSON.....	205

Go и XML.....	207
Чтение XML-файла	210
Настройка вывода данных в формате XML	211
Go и формат YAML	213
Дополнительные ресурсы	213
Упражнения.....	214
Резюме	214
Глава 5. Как улучшить код Go с помощью структур данных.....	216
О графах и узлах	217
Сложность алгоритма	217
Двоичные деревья в Go	218
Реализация двоичного дерева в Go.....	219
Преимущества двоичных деревьев.....	221
Пользовательские хеш-таблицы в Go.....	222
Реализация пользовательской хеш-таблицы в Go	222
Реализация функции поиска	225
Преимущества пользовательских хеш-таблиц	226
Связные списки в Go.....	226
Реализация связного списка в Go.....	227
Преимущества связных списков	231
Двусвязные списки в Go	231
Реализация двусвязного списка в Go	232
Преимущества двусвязных списков.....	235
Очереди в Go.....	236
Реализация очереди в Go.....	236
Стеки в Go	239
Реализация стека в Go.....	239
Пакет container	242
Использование пакета container/heap	242
Использование пакета container/list	245
Использование пакета container/ring.....	247

Генерация случайных чисел.....	248
Генерация случайных строк	251
Генерация безопасной последовательности случайных чисел.....	253
Выполнение матричных вычислений.....	255
Сложение и вычитание матриц	255
Умножение матриц	258
Деление матриц.....	261
Разгадывание головоломок sudoku.....	267
Дополнительные ресурсы	270
Упражнения.....	271
Резюме	272
Глава 6. Неочевидные знания о пакетах и функциях Go.....	273
Что такое Go-пакеты.....	274
Что такое функции Go	274
Анонимные функции	275
Функции, которые возвращают несколько значений	275
Функции, возвращающие именованные значения	277
Функции, принимающие указатели	279
Функции, которые возвращают указатели.....	280
Функции, которые возвращают другие функции	281
Функции, которые принимают другие функции в качестве параметров....	282
Функции с переменным числом параметров.....	283
Разработка Go-пакетов	285
Компиляция Go-пакета.....	287
Закрытые переменные и функции	287
Функция init().....	287
Go-модули	290
Создание и использование Go-модулей.....	290
Использование двух версий одного и того же Go-модуля.....	298
Где хранятся Go-модули	299
Команда go mod vendor	300

Как писать хорошие Go-пакеты.....	300
Пакет syscall	302
Как на самом деле работает fmt.Println().....	304
Пакеты go/scanner, go/parser и go/token.....	306
Пакет go/ast.....	307
Пакет go/scanner.....	307
Пакет go/parser.....	309
Практический пример	311
Поиск имен переменных заданной длины	313
Шаблоны для текста и HTML.....	318
Вывод простого текста.....	318
Вывод текста в формате HTML.....	320
Дополнительные ресурсы	327
Упражнения.....	328
Резюме	328
Глава 7. Рефлексия и интерфейсы на все случаи жизни.....	330
Методы типов	330
Интерфейсы в Go.....	332
Операции утверждения типа.....	333
Как писать свои интерфейсы	335
Использование интерфейса Go	336
Использование переключателей для интерфейсов и типов данных.....	338
Рефлексия	340
Простой пример рефлексии	340
Более сложный пример рефлексии	343
Три недостатка рефлексии.....	345
Библиотека reflectwalk	346
Объектно-ориентированное программирование на Go	348
Основы git и GitHub	351
Использование git.....	351

Отладка с помощью Delve	357
Пример отладки	358
Дополнительные ресурсы	362
Упражнения	362
Резюме	362
Глава 8. Как объяснить UNIX-системе, что она должна делать	364
О процессах в UNIX	365
Пакет flag	365
Пакет viper	370
Простой пример использования viper	371
От flag к viper	372
Чтение конфигурационных файлов в формате JSON	373
Чтение конфигурационных файлов в формате YAML	375
Пакет cobra	377
Простой пример cobra	378
Создание псевдонимов команд	382
Интерфейсы io.Reader и io.Writer	385
Буферизованный и небуферизованный ввод и вывод в файл	385
Пакет bufio	386
Чтение текстовых файлов	386
Построчное чтение текстового файла	386
Чтение текстового файла по словам	388
Посимвольное чтение текстового файла	390
Чтение из /dev/random	392
Чтение заданного количества данных	393
Преимущества двоичных форматов	395
Чтение CSV-файлов	396
Запись в файл	399
Загрузка и сохранение данных на диске	401
И снова пакет strings	404

Пакет bytes	406
Полномочия доступа к файлам	407
Обработка сигналов в UNIX	408
Обработка двух сигналов	409
Обработка всех сигналов	411
Программирование UNIX-каналов на Go	413
Реализация утилиты cat(1) на Go	414
Структура syscall.PtraceRegs	416
Отслеживание системных вызовов	418
Идентификаторы пользователя и группы	422
Docker API и Go	423
Дополнительные ресурсы	426
Упражнения	427
Резюме	428
Глава 9. Конкурентность в Go: горутины, каналы и конвейеры	429
О процессах, потоках и горутинах	430
Планировщик Go	430
Конкурентность и параллелизм	431
Горутины	431
Создание горутины	432
Создание нескольких горутин	433
Как дождаться завершения горутин, прежде чем закончить программу	435
Что происходит, если количество вызовов Add() и Done() не совпадает	437
Каналы	439
Запись в канал	439
Чтение из канала	440
Прием данных из закрытого канала	442
Каналы как аргументы функции	443
Конвейеры	444

Состояние гонки	447
Сравнение моделей конкурентности в Go и Rust.....	449
Сравнение моделей конкурентности в Go и Erlang	449
Дополнительные ресурсы	450
Упражнения.....	450
Резюме	451
Глава 10. Конкурентность в Go: расширенные возможности	452
И снова о планировщике Go	453
Переменная среды GOMAXPROCS.....	455
Ключевое слово select	456
Принудительное завершение горутин.....	459
Принудительное завершение горутин, способ 1.....	459
Принудительное завершение горутин, способ 2.....	461
И снова о Go-каналах	463
Сигнальные каналы	464
Буферизованные каналы.....	464
Нулевые каналы	466
Каналы каналов	467
Выбор последовательности исполнения горутин	470
Как не надо использовать горутин.....	472
Общая память и общие переменные	473
Тип sync.Mutex.....	474
Тип sync.RWMutex.....	478
Пакет atomic	481
Совместное использование памяти с помощью горутин	483
И снова об операторе go.....	485
Распознавание состояния гонки	488
Пакет context	493
Расширенный пример использования пакета context.....	497
Еще один пример использования пакета context	502
Пулы обработчиков.....	503

Дополнительные ресурсы	508
Упражнения.....	508
Резюме	509
Глава 11. Тестирование, оптимизация и профилирование кода.....	510
Оптимизация	511
Оптимизация кода Go	512
Профилирование кода Go	513
Стандартный Go-пакет net/http/pprof	513
Простой пример профилирования.....	513
Удобный внешний пакет для профилирования.....	521
Веб-интерфейс Go-профилировщика.....	523
Утилита go tool trace.....	527
Тестирование кода Go	532
Написание тестов для существующего кода Go.....	532
Тестовое покрытие кода	536
Тестирование HTTP-сервера с базой данных.....	539
Пакет testing/quick.....	545
Бенчмаркинг кода Go.....	551
Простой пример бенчмаркинга	552
Неправильно определенные функции бенчмаркинга.....	557
Бенчмаркинг буферизованной записи	558
Обнаружение недоступного кода Go	562
Кросс-компиляция	564
Создание примеров функций	565
От кода Go до машинного кода	567
Использование ассемблера в Go	568
Генерация документации	570
Использование образов Docker	575
Дополнительные ресурсы	577
Упражнения.....	578
Резюме	579

Глава 12. Основы сетевого программирования на Go	580
Что такое net/http, net и http.RoundTripper	581
Тип http.Response	581
Тип http.Request	582
Тип http.Transport	582
Что такое TCP/IP	583
Что такое IPv4 и IPv6.....	584
Утилита командной строки nc(1).....	584
Чтение конфигурации сетевых интерфейсов.....	585
Выполнение DNS-поиска	589
Получение NS-записей домена.....	591
Получение MX-записей домена	593
Создание веб-сервера на Go	594
Использование пакета atomic.....	597
Профилирование HTTP-сервера	599
Создание веб-сайта на Go.....	604
HTTP-трассировка	613
Тестирование HTTP-обработчиков	616
Создание веб-клиента на Go	618
Как усовершенствовать наш веб-клиент Go	620
Задержки HTTP-соединений.....	623
Подробнее о SetDeadline.....	625
Установка периода ожидания на стороне сервера	625
Еще один способ определить период ожидания	627
Инструменты Wireshark и tshark	629
Go и gRPC.....	629
Определение файла описания интерфейса	629
gRPC-клиент	632
gRPC-сервер	633
Дополнительные ресурсы	635
Упражнения.....	636
Резюме	637

Глава 13. Сетевое программирование: создание серверов и клиентов	638
Работа с HTTPS-трафиком.....	639
Создание сертификатов	639
HTTPS-клиент	640
Простой HTTPS-сервер.....	642
Разработка TLS-сервера и TLS-клиента	643
Стандартный Go-пакет net	646
TCP-клиент	646
Другая версия TCP-клиента.....	648
TCP-сервер	650
Другая версия TCP-сервера.....	652
UDP-клиент.....	654
Разработка UDP-сервера.....	656
Конкурентный TCP-сервер	658
Удобный конкурентный TCP-сервер	662
Создание образа Docker для TCP/IP-сервера на Go	668
Дистанционный вызов процедур.....	670
RPC-клиент	671
RPC-сервер	672
Низкоуровневое сетевое программирование.....	674
Получение необработанных сетевых данных ICMP.....	676
Дополнительные ресурсы	680
Упражнения.....	681
Резюме	682
 Глава 14. Машинное обучение на Go	 683
Вычисление простых статистических показателей	684
Регрессия	688
Линейная регрессия.....	688
Реализация линейной регрессии	688
Вывод данных.....	690
Классификация	694

Кластеризация.....	698
Выявление аномалий.....	700
Нейронные сети.....	702
Анализ выбросов	704
Работа с TensorFlow	707
Поговорим о Kafka	712
Дополнительные ресурсы	716
Упражнения.....	717
Резюме	717
Что дальше?	718

Об авторе

Михалис Цукалос (Mihalis Tsoukalos) — администратор UNIX, программист, администратор баз данных и математик. Любит писать технические книги и статьи, узнавать что-то новое. Помимо этой книги, Михалис написал *Go Systems Programming*, а также более 250 технических статей для многих журналов, включая Sys Admin, MacTech, Linux User and Developer, Usenix ;login:, Linux Format и Linux Journal. Сфера научных интересов Михалиса — базы данных, визуализация, статистика и машинное обучение.

Вы можете связаться с автором через его сайт <https://www.mtsoukalos.eu/> или по адресу @mactsouk.

Михалис также увлекается фотографией.

Я благодарю сотрудников Packt Publishing за помощь в написании этой книги, в том числе моего научного редактора Мэта Райера (Mat Ryer), а также Кишора Рита (Kishor Rit) за ответы на все мои вопросы и поддержку в период написания книги.

Эту книгу я посвящаю памяти моих любимых родителей Иоанниса и Аргетты.

О научном редакторе

Мэт Райер (Mat Ryer) пишет компьютерные программы с шести лет: сначала на BASIC для ZX Spectrum, а затем, вместе со своим отцом, — на AmigaBASIC и AMOS для Commodore Amiga. Много времени он потратил на копирование вручную кода из журнала Amiga Format, изменение значений переменных или ссылок операторов GOTO, чтобы посмотреть, что из этого выйдет. Тот же дух исследования и одержимость программированием привели 18-летнего Мэтта к работе в местной организации в Мансфилде (Великобритания), где он начал создавать веб-сайты и другие онлайн-сервисы.

После нескольких лет работы с различными технологиями в разных областях не только в Лондоне, но и по всему миру Мэт обратил внимание на новый язык системного программирования под названием Go, впервые использованный в Google. Поскольку Go решал очень актуальные и остросовременные технические проблемы, Мэт начал использовать этот язык для решения задач, когда Go находился еще на стадии бета-тестирования, и с тех пор продолжает программировать на нем. Мэт работал над разными проектами с открытым исходным кодом, создал несколько пакетов Go, в том числе Testify, Moq, Silk и Is, а также инструментарий для разработчиков в MacOS — BitBar.

С 2018 года Мэт — соучредитель компании Machine Box, но он по-прежнему принимает участие в конференциях, пишет о Go в своем блоге и является активным участником сообщества Go.

Предисловие

Книга «Golang для профи: работа с сетью, многопоточность, структуры данных и машинное обучение с Go», второе издание, которую вы сейчас держите в руках, поможет вам стать самым лучшим разработчиком на Go!

В книге много свежих интересных тем, включая совершенно новую главу об использовании Go для машинного обучения. Вы также найдете здесь описания и примеры кода для пакетов Viper и Cobra Go, gRPC, вы узнаете, как работать с образами Docker, файлами YAML, пакетами `go/scanner` и `go/token`, научитесь генерировать из Go код WebAssembly. В общей сложности второе издание книги «Golang для профи» расширилось более чем на 130 страниц.

Для кого предназначена эта книга

Эта книга рассчитана на начинающих и программистов среднего уровня, работающих с языком Go, которые стремятся перейти на новый уровень знаний Go, а также на опытных разработчиков, пишущих на других языках программирования, которые хотят изучать Go, не возвращаясь к циклу `for`.

Часть информации из второго издания можно найти в другой книге автора — *Go Systems Programming*. Основное различие между этими двумя книгами заключается в том, что *Go Systems Programming* посвящена разработке системных инструментов с использованием возможностей языка Go, а «Golang для профи» — разъяснению возможностей и внутренних особенностей самого Go, что позволит вам стать лучшим разработчиком на Go. Обе книги могут служить справочниками после того, как вы их прочитаете один-два раза.

О чем эта книга

Глава 1 «Go и операционная система» начинается с истории возникновения языка Go и его преимуществ, затем приводится описание утилиты `godoc` и объяснение, как компилировать и выполнять Go-программы. Далее речь пойдет о выводе данных

и получении данных от пользователя, об аргументах командной строки программы и использовании журнальных файлов. Последний раздел первой главы посвящен обработке ошибок, которая в Go играет ключевую роль.

Глава 2 «Go изнутри» посвящена сборщику мусора Go и принципам его работы. Затем поговорим о небезопасном коде и о пакете `unsafe`, а также о том, как вызывать из Go-программы код на C, а из C-программы — код на Go.

Вы узнаете, как использовать ключевое слово `defer`, а также познакомитесь с утилитами `strace(1)` и `dtrace(1)`. В оставшихся разделах этой главы вы изучите, как получить информацию о вашей среде Go, как использовать ассемблер и как генерировать из Go код `WebAssembly`.

Глава 3 «Работа с основными типами данных Go» посвящена типам данных, предоставляемым в Go: массивам, срезам и хеш-таблицам, а также указателям, константам, циклам, функциям для работы с датами и временем. Вы точно не захотите пропустить эту главу!

Глава 4 «Использование составных типов данных» начинается с изучения структур Go и ключевого слова `struct`, после чего речь пойдет о кортежах, строках, рунах, байтовых срезах и строковых литералах. Из оставшейся части главы вы узнаете о регулярных выражениях и сопоставлении с образцом, об операторе `switch`, пакетах `strings` и `math/big`, о разработке на Go хранилища типа «ключ — значение» и о работе с файлами форматов XML и JSON.

Глава 5 «Как улучшить код Go с помощью структур данных» посвящена разработке пользовательских структур данных в тех случаях, когда стандартные структуры Go не соответствуют конкретной задаче. Здесь же рассмотрено построение и применение бинарных деревьев, связанных списков, пользовательских хеш-таблиц, стеков и очередей, а также их преимущества. В этой главе продемонстрировано использование структур из стандартного пакета Go `container`, а также показано, как можно использовать Go для проверки головоломок судoku и генерации случайных чисел.

Глава 6 «Неочевидные знания о пакетах и функциях Go» посвящена пакетам и функциям, в том числе использованию функции `init()`, стандартного Go-пакета `syscall`, а также пакетов `text/template` и `html/template`. Кроме того, вы узнаете, как применять расширенные пакеты `go/scanner`, `go/parser` и `go/token`. Эта глава определенно улучшит ваши навыки разработки на Go!

В *главе 7 «Рефлексия и интерфейсы на все случаи жизни»* обсуждаются три передовые концепции Go: рефлексия, интерфейсы и методы типов. Кроме того, в этой главе описываются объектно-ориентированные возможности Go и способы отладки Go-программ с помощью `Delve`.

Глава 8 «Как объяснить UNIX-системе, что она должна делать» посвящена системному программированию на Go. Здесь рассматриваются такие темы, как пакет `flag` для работы с аргументами командной строки, обработка сигналов UNIX, файловый ввод и вывод, пакеты `bytes`, `io.Reader` и `io.Writer`, а также

обсуждается использование пакетов `Viper` и `Cobra` Go. Напомню: если вы действительно увлекаетесь системным программированием на Go, то я настоятельно рекомендую вам после прочтения этой книги приобрести и прочесть *Go Systems Programming!*

В главе 9 «Конкурентность в Go: горутины, каналы и конвейеры» обсуждаются горутины, каналы и конвейеры — то, что позволяет реализовать конкурентность на Go.

Вы также узнаете, чем различаются между собой процессы, потоки и горутины, познакомитесь с пакетом `sync` и особенностями работы планировщика Go.

Глава 10 «Конкурентность в Go: расширенные возможности» является продолжением предыдущей главы. Прочитав ее, вы станете повелителем горутин и каналов! Вы глубже изучите планировщик Go, научитесь использовать мощное ключевое слово `select`, узнаете о различных типах каналов Go, а также о разделяемой памяти, мьютексах, типах `sync.Mutex` и `sync.RWMutex`. В последней части главы говорится о пакете `context`, пулах обработчиков и о том, как распознать состояние гонки (race conditions).

В главе 11 «Тестирование, оптимизация и профилирование кода» обсуждаются тестирование, оптимизация и профилирование кода, а также кросс-компиляция под разные платформы, создание документации, тестирование производительности Go-кода, создание тестовых функций и поиск неиспользуемого Go-кода.

Глава 12 «Основы сетевого программирования на Go» посвящена пакету `net/http` и тому, как разрабатывать веб-клиенты и веб-серверы на Go. Далее рассмотрены структуры `http.Response`, `http.Request` и `http.Transport`, а также тип `http.NewServeMux`. Вы даже узнаете, как разработать на Go целый веб-сайт! Кроме того, в этой главе вы научитесь читать конфигурацию сетевых интерфейсов и выполнять на Go DNS-поиск, а также использовать с Go gRPC.

В главе 13 «Сетевое программирование: создание серверов и клиентов» рассказывается о работе с HTTPS-трафиком и создании на Go серверов и клиентов UDP и TCP с использованием функций из пакета `net`. Здесь же рассмотрены такие темы, как построение клиентов и серверов RPC, разработка на Go многопоточного TCP-сервера и чтение «сырых» сетевых пакетов.

В главе 14 «Машинное обучение на Go» рассказывается о реализации на Go алгоритмов машинного обучения, включая классификацию, кластеризацию, обнаружение аномалий, выбросы, нейронные сети и TensorFlow, а также работу Go с Apache Kafka.

Эту книгу можно разделить на три логические части. В первой части подробно рассматриваются некоторые важные концепции Go, включая пользовательский ввод и вывод, загрузку внешних Go-пакетов, компиляцию Go-кода, вызов из Go кода на C и создание кода WebAssembly, а также использование основных и составных типов Go.

Вторая часть начинается с главы 5 и включает в себя главы 6 и 7. Эти три главы посвящены организации Go-кода в виде пакетов и модулей, структуре Go-проектов и некоторым дополнительным функциям Go.

Последняя часть включает в себя оставшиеся семь глав и посвящена более практическим темам Go. В главах 8–11 рассказывается о системном программировании на Go, реализации конкурентности на Go, тестировании, оптимизации и профилировании кода. В последних трех главах этой книги речь идет о сетевом программировании и машинном обучении на Go.

Книга включает в себя материалы о Go и WebAssembly, использовании Docker с Go, создании профессиональных утилит работы с командной строкой с помощью пакетов Viper и Cobra, обработке JSON и YAML, выполнении операций с матрицами, работе с головоломками судoku, пакетами `go/scanner` и `go/token`, а также с `git`(1) и GitHub, пакетом `atomic`, об использовании в Go gRPC и HTTPS.

В книге представлены относительно небольшие, но полные Go-программы, которые иллюстрируют описанные концепции. У этого есть два основных преимущества: во-первых, вам не придется просматривать бесконечный листинг, чтобы изучить всего одну методику, а во-вторых, вы сможете использовать этот код как отправную точку при создании собственных приложений и утилит.



Поскольку контейнеры и Docker очень важны, я включил в эту книгу примеры исполняемых Go-файлов, которые используются в образах Docker, ведь образы Docker являются отличным способом развертывания серверного программного обеспечения.

Как получить максимальную пользу от этой книги

Для работы с книгой вам потребуется UNIX-компьютер и установленная на нем относительно новая версия Go. Это может быть любой компьютер с операционной системой Mac OS X, macOS или Linux. Большая часть представленного здесь кода также будет работать и на компьютерах под управлением Microsoft Windows.

Чтобы извлечь максимальную пользу из этой книги, вы должны как можно быстрее применить знания, полученные при прочтении каждой главы, в своих программах и посмотреть, что работает, а что — нет! Как я уже говорил, попробуйте выполнять все упражнения, приведенные в конце каждой главы, или создавайте собственные задачи программирования.

Загрузите файлы с примерами кода

Примеры кода для этой книги размещены на GitHub по адресу <https://github.com/PacktPublishing/Mastering-Go-Second-Edition>. В случае если код обновится, он будет также размещен в уже существующем репозитории GitHub.

Чтобы скачать файлы с кодом, нужно выполнить следующие действия.

1. Перейдите по указанной ссылке на сайт github.com.
2. Нажмите кнопку Clone or Download.
3. Щелкните кнопкой мыши на ссылке Download ZIP.
4. Скачайте архив с файлами примеров.

Когда загрузится файл с архивом, распакуйте его или извлеките из него папку с упакованными файлами, используя последнюю версию одного из следующих архиваторов:

- ❑ WinRAR или 7-Zip для Windows;
- ❑ Zipeg, iZip или UnRarX для Mac;
- ❑ 7-Zip или PeaZip для Linux.

По адресу <https://github.com/PacktPublishing/> вам доступны и другие примеры кода из нашего богатого каталога книг и видео. Все это к вашим услугам!

Где скачать цветные иллюстрации

Мы предоставляем вам PDF-файл с цветными скриншотами и диаграммами, использованными в этой книге. Вы можете его скачать по этому адресу: <https://static.packt-cdn.com/downloads/9781838559335.pdf>.

Условные обозначения

В этой книге используются следующие текстовые обозначения.

Код в тексте: размещенные в тексте книги кодовые слова, имена таблиц из базы данных, папок и файлов; расширения файлов, пути, фиктивные URL-адреса, вводимые пользователем данные и учетные записи Twitter. Например: «Первый способ похож на использование команды `map(1)`, но только для функций и пакетов Go».

Блоки кода представлены следующим образом:

```
package main
import (
    "fmt"
)
```



```
func main() {  
    fmt.Println("This is a sample Go program!")  
}
```

Когда нужно обратить ваше внимание на определенную часть блока кода, соответствующие строки или элементы выделяются жирным шрифтом:

```
import (  
    "fmt"  
)  
func main() {  
    fmt.Println("This is a sample Go program!")  
}
```

Все, что вводится или выводится в командной строке, записывается следующим образом:

```
$ date  
Sat Oct 21 20:09:20 EEST 2017  
$ go version  
go version go1.12.7 darwin/amd64
```

Курсивом обозначаются новые термины, **жирным шрифтом** — важные слова. То, что вы видите на экране (например, слова в меню или диалоговых окнах), отображается в тексте так: «Выберите на панели Administration раздел System info».



Так выглядят советы и рекомендации.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Go и операционная система

Эта глава — введение в различные аспекты языка Go, которые будут очень полезными для начинающих. Более опытные разработчики на Go также могут использовать эту главу в качестве курса повышения квалификации. Как часто бывает с большинством практических предметов, лучший способ что-то усвоить — поэкспериментировать с этим. В данном случае экспериментировать означает самостоятельно писать Go-код, совершать собственные ошибки и учиться на них. Только не позволяйте этим ошибкам и сообщениям о них отбивать у вас охоту учиться дальше!

В этой главе рассмотрены следующие темы:

- ☐ история и будущее языка программирования Go;
- ☐ преимущества Go;
- ☐ компиляция Go-кода;
- ☐ выполнение Go-кода;
- ☐ загрузка и использование внешних Go-пакетов;
- ☐ стандартный ввод, вывод и сообщения об ошибках в UNIX;
- ☐ вывод данных на экран;
- ☐ получение данных от пользователя;
- ☐ вывод данных о стандартных ошибках;
- ☐ работа с лог-файлами;
- ☐ использование Docker для компиляции и запуска исходных файлов Go;
- ☐ обработка ошибок в Go.

История Go

Go — это современный универсальный язык программирования с открытым исходным кодом, выпуск которого официально состоялся в конце 2009 года. Go планировался как внутренний проект Google — это означает, что сначала он был запущен в качестве эксперимента и с тех пор вдохновлялся многими другими

языками программирования, в том числе C, Pascal, Alef и Oberon. Создателями Go являются профессиональные программисты Роберт Гризмер (Robert Griesemer), Кен Томсон (Ken Thomson) и Роб Пайк (Rob Pike). Они разработали Go как язык для профессионалов, позволяющий создавать надежное, устойчивое и эффективное программное обеспечение. Помимо синтаксиса и стандартных функций, в состав Go входит довольно богатая стандартная библиотека.

На момент публикации этой книги последней стабильной версией Go была версия 1.14. Однако даже если номер вашей версии выше, книга все равно будет актуальной.

Если вы устанавливаете Go впервые, начните с посещения веб-сайта <https://golang.org/dl/>. Однако с большой вероятностью в вашем дистрибутиве UNIX уже есть готовый к установке пакет для языка программирования Go, поэтому вы можете получить Go с помощью обычного менеджера пакетов.

Куда движется Go?

Сообщество Go уже обсуждает следующую полноценную версию Go, которая будет называться Go 2, но пока еще не появилось ничего определенного.

Цель нынешней команды по разработке Go 1 — сделать так, чтобы Go 2 больше развивался по инициативе сообщества. В целом это неплохая идея, однако всегда есть риск, когда слишком много людей участвуют в принятии важных решений относительно языка программирования, который изначально создавался и разрабатывался как внутренний проект небольшой группы гениальных профессионалов.

Некоторые крупные изменения, рассматриваемые для Go 2, — это дженерики, управление версиями пакетов и улучшенная обработка ошибок. Все новые функции в настоящее время находятся на стадии обсуждения, и вам не стоит о них беспокоиться — однако нужно иметь представление о направлении, в котором движется Go.

Преимущества Go

У языка Go много преимуществ. Некоторые из них уникальны для Go, а другие свойственны и иным языкам программирования.

Среди наиболее значимых преимуществ и возможностей Go можно отметить следующие.

- ❑ Go — современный язык программирования, созданный опытными разработчиками. Его код понятен и легко читается.
- ❑ Цель Go — счастливые разработчики. Именно счастливые разработчики пишут самый лучший код!
- ❑ Компилятор Go выводит информативные предупреждения и сообщения об ошибках, которые помогут вам решить конкретную проблему. Проще говоря,

компилятор Go будет вам помогать, а не портить жизнь, выводя бессмысленные сообщения!

- ❑ Код Go является переносимым, особенно между UNIX-машинами.
- ❑ Go поддерживает процедурное, параллельное и распределенное программирование.
- ❑ Go поддерживает *сборку мусора*, поэтому вам не придется заниматься выделением и освобождением памяти.
- ❑ У Go нет *препроцессора*; вместо этого выполняется высокоскоростная компиляция. Вследствие этого Go можно использовать как язык сценариев.
- ❑ Go позволяет создавать веб-приложения и предоставляет простой веб-сервер для их тестирования.
- ❑ В стандартную библиотеку Go входит множество пакетов, которые упрощают жизнь разработчика. Функции, входящие в стандартную библиотеку Go, предварительно тестируются и отлаживаются людьми, разрабатывающими Go, а значит, в основном работают без ошибок.
- ❑ По умолчанию в Go используется *статическая компоновка* — это значит, что создаваемые двоичные файлы легко переносятся на другие компьютеры с той же ОС. Как следствие, после успешной компиляции Go-программы и создания исполняемого файла не приходится беспокоиться о библиотеках, зависимостях и разных версиях этих библиотек.
- ❑ Для разработки, отладки и тестирования Go-приложений вам не понадобится *графический интерфейс пользователя* (Graphical User Interface, GUI), так как Go можно использовать из командной строки — именно так, как, мне кажется, предпочитают многие пользователи UNIX.
- ❑ Go поддерживает Unicode, а следовательно, вам не понадобится дополнительная обработка для вывода символов на разных языках.
- ❑ В Go сохраняется принцип независимости, потому что несколько независимых функций работают лучше, чем много взаимно перекрывающихся.

Идеален ли Go?

Идеального языка программирования не существует, и Go не исключение. Есть языки программирования, которые эффективнее в некоторых других областях программирования, или же мы их просто больше любим. Лично я не люблю Java, и хотя раньше мне нравился C++, сейчас он мне не нравится. C++ стал слишком сложным как язык программирования, а код на Java, на мой взгляд, не очень красиво смотрится.

Вот некоторые из недостатков Go:

- ❑ у Go нет встроенной поддержки *объектно-ориентированного программирования*. Это может стать проблемой для тех программистов, которые привыкли писать

объектно-ориентированный код. Однако вы можете имитировать наследование в Go, используя композицию;

- ❑ некоторые считают, что Go никогда не заменит C;
- ❑ С все еще остается более быстрым, чем любой другой язык системного программирования, главным образом потому, что UNIX написана на C.

Несмотря на это, Go — вполне достойный язык программирования. Он вас не разочарует, если вы найдете время для его изучения и использования.

Что такое препроцессор

Как я уже говорил, в Go нет препроцессора, и это хорошо. Препроцессор — это программа, которая обрабатывает входные данные и генерирует выходные данные, которые будут использоваться в качестве входных для другой программы. В контексте языков программирования входные данные препроцессора — это исходный код программы, который будет обработан препроцессором и затем передан на вход компилятора языка программирования.

Самый большой недостаток препроцессора — он ничего не знает ни о базовом языке, ни о его синтаксисе! Это значит, что, когда используется препроцессор, нельзя гарантировать, что окончательная версия кода будет делать именно то, что вы хотите: препроцессор может изменить и логику, и семантику исходного кода.

Препроцессор используется в таких языках программирования, как C, C++, Ada, PL/SQL. Печально известный препроцессор C обрабатывает строки, которые начинаются с символа # и называются *директивами* или *прагмами*. Таким образом, директивы и прагмы не являются частью языка программирования C!

Утилита godoc

В дистрибутив Go входит множество инструментов, способных значительно упростить жизнь программиста. Одним из таких инструментов является утилита `godoc`¹, которая позволяет просматривать документацию загруженных функций и пакетов Go без подключения к Интернету.

Утилита `godoc` может выполняться как обычное приложение командной строки, которое выводит данные на терминал, или же как приложение командной строки, которое запускает веб-сервер. В последнем случае для просмотра документации Go вам понадобится браузер.



Если ввести в командной строке просто `godoc`, без каких-либо параметров, то получим список параметров командной строки, поддерживаемых `godoc`.

¹ Если `godoc` на вашем компьютере не установлена, просто выполните такую команду:
\$ go get golang.org/x/tools/cmd/godoc. — Здесь и далее *примеч. науч. ред.*

Первый способ аналогичен использованию команды `man(1)`, только для функций и пакетов Go. Например, чтобы получить информацию о функции `Printf()` из пакета `fmt`, необходимо ввести команду:

```
$ go doc fmt.Printf
```

Аналогичным образом можно получить информацию обо всем пакете `fmt`, введя следующую команду:

```
$ go doc fmt
```

Второй способ требует выполнения `godoc` с параметром `-http`:

```
$ godoc -http=:8001
```

Число в предыдущей команде, в данном случае равное `8001`, — это номер порта, который будет прослушивать HTTP-сервер. Вы можете указать любой доступный номер порта, если у вас есть необходимые привилегии. Однако обратите внимание, что номера портов от 0 до 1023 зарезервированы и могут использоваться только пользователем `root`, поэтому лучше избегать использования одного из этих портов и выбирать какой-нибудь другой, если только он еще не используется другим процессом.

Вместо знака равенства в предыдущей команде можно поставить символ пробела. Следующая команда полностью эквивалентна предыдущей:

```
$ godoc -http :8001
```

Если после этого ввести в браузере URL-адрес `http://localhost:8001/pkg/`, то вы получите список доступных пакетов Go и сможете просмотреть их документацию.

Компиляция Go-кода

Из этого раздела вы узнаете, как скомпилировать код на Go. Хорошая новость: вы можете скомпилировать код Go из командной строки без графического приложения. Более того, для Go не имеет значения имя исходного файла с текстом программы, если именем пакета является `main` и в нем есть только одна функция `main()`. Дело в том, что именно с функции `main()` начинается выполнение программы. Из-за этого в файлах одного проекта не может быть нескольких функций `main()`.

Нашей первой скомпилированной Go-программой будет программа с именем `aSourceFile.go`, которая содержит следующий код Go:

```
package main
import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

Обратите внимание, что сообщество Go предпочитает называть исходный файл `Go source_file.go`, а не `aSourceFile.go`. В любом случае, что бы вы ни выбрали, будьте последовательны.

Чтобы скомпилировать `aSourceFile.go` и создать *статически скомпонованный* исполняемый файл, нужно выполнить следующую команду:

```
$ go build aSourceFile.go
```

В результате будет создан новый исполняемый файл с именем `aSourceFile`, который теперь нужно выполнить:

```
$ file aSourceFile
aSourceFile: Mach-O 64-bit executable x86_64
$ ls -l aSourceFile
-rwxr-xr-x 1 mtsouk staff 2007576 Jan 10 21:10 aSourceFile
$ ./aSourceFile
This is a sample Go program!
```

Основная причина, по которой файл `aSourceFile` такой большой, заключается в том, что он статически скомпонован, другими словами, для его работы не требуется никаких внешних библиотек.

Выполнение Go-кода

Есть другой способ выполнить Go-код, при котором не создаются постоянных исполняемых файлов — генерируется лишь несколько временных файлов, которые впоследствии автоматически удаляются.



Этот способ позволяет использовать Go как язык сценариев, подобно Python, Ruby или Perl.

Итак, чтобы запустить `aSourceFile.go`, не создавая исполняемый файл, необходимо выполнить следующую команду:

```
$ go run aSourceFile.go
This is a sample Go program!
```

Как видим, результат выполнения этой команды точно такой же, как и раньше.



Обратите внимание, что при запуске `go run` компилятору Go по-прежнему нужно создать исполняемый файл. Только вы его не видите, потому что он автоматически выполняется и так же автоматически удаляется после завершения программы. Из-за этого может показаться, что нет необходимости в исполняемом файле.

В этой книге для выполнения примеров кода в основном будет использоваться `go run`; в первую очередь потому, что так проще, чем сначала запускать `go build`, а затем — исполняемый файл. Кроме того, `go run` после завершения программы не оставляет файлов на жестком диске.

Два правила Go

В Go приняты строгие правила кодирования. Они помогут вам избежать ошибок и багов в коде, а также позволят облегчить чтение кода для сообщества Go. В этом разделе представлены два правила Go, о которых вам необходимо знать.

Как я уже говорил, пожалуйста, помните, что компилятор Go будет помогать вам, а не усложнять жизнь. Основная цель компилятора Go — компилировать код и повышать его качество.

Правило пакетов Go: не нужен — не подключай

В Go приняты строгие правила использования пакетов. Вы не можете просто подключить пакет на всякий случай и в итоге не использовать его.

Рассмотрим следующую простую программу, которая сохраняется как `packageNotUsed.go`:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("Hello there!")
}
```



В этой книге вам встретится множество сообщений об ошибках, ошибочных ситуациях и предупреждений. Считаю, что изучение кода, который не компилируется, не менее (а иногда и более!) полезно, чем просто чтение Go-кода, который компилируется без каких-либо ошибок. Компилятор Go обычно выводит информативные сообщения об ошибках и предупреждения. Эти сообщения, скорее всего, помогут вам устранить ошибочную ситуацию, поэтому не стоит недооценивать их.

Если попытаться выполнить `packageNotUsed.go`, то программа не будет выполнена, а мы получим от Go следующее сообщение об ошибке:

```
$ go run packageNotUsed.go
# command-line-arguments
./packageNotUsed.go:5:2: imported and not used: "os"
```


Если удалить пакет `os` из списка `import` программы, то `packageNotUsed.go` от-
лично скомпилируется — попробуйте сами.

Сейчас еще не время говорить о том, как нарушать правила Go, однако суще-
ствует способ обойти такое ограничение. Он показан в следующем Go-коде, кото-
рый сохраняется в файле `packageNotUsedUnderscore.go`:

```
package main

import (
    "fmt"
    _ "os"
)

func main() {
    fmt.Println("Hello there!")
}
```

Как видим, если в списке `import` поставить перед именем пакета символ под-
черкивания, то мы не получим сообщение об ошибке в процессе компиляции, даже
если этот пакет не используется в программе:

```
$ go run packageNotUsedUnderscore.go
Hello there!
```



Причина, по которой Go позволяет обойти это правило, станет более
понятной в главе 6.

Правильный вариант размещения фигурных скобок — всего один

Рассмотрим следующую Go-программу с именем `curly.go`:

```
package main

import (
    "fmt"
)

func main()
{
    fmt.Println("Go has strict rules for curly braces!")
}
```

Все выглядит просто отлично, но если вы попытаетесь это выполнить, то будете
весьма разочарованы, потому что код не скомпилируется и, соответственно, не за-
пустится, а вы получите следующее сообщение о *синтаксической ошибке*:

```
$ go run curly.go
# command-line-arguments
./curly.go:7:6: missing function body for "main"
./curly.go:8:1: syntax error: unexpected semicolon or newline before {
```

Официально смысл этого сообщения об ошибке разъясняется так: во многих контекстах Go требует использования точки с запятой как признака завершения оператора и поэтому компилятор автоматически вставляет точки с запятой там, где считает их необходимыми. Поэтому при размещении открывающей фигурной скобки (`{`) в отдельной строке компилятор Go поставит точку с запятой в конце предыдущей строки (`func main()`) — это и есть причина сообщения об ошибке.

Как скачивать Go-пакеты

Стандартная библиотека Go весьма обширна, однако бывают случаи, когда необходимо загрузить внешние пакеты Go, чтобы использовать их функциональные возможности. В этом разделе вы узнаете, как загрузить внешний Go-пакет и где он будет размещен на вашем UNIX-компьютере.



Имейте в виду, что недавно в Go появился новый функционал — модули, которые все еще находятся в стадии разработки и поэтому могут внести изменения в работу с внешним Go-кодом. Однако процедура загрузки на компьютер отдельного Go-пакета останется прежней.

Вы узнаете намного больше о пакетах и модулях Go из главы 6.

Рассмотрим следующую простую Go-программу, которая сохраняется как `getPackage.go`:

```
package main

import (
    "fmt"
    "github.com/mactsouk/go/simpleGitHub"
)

func main() {
    fmt.Println(simpleGitHub.AddTwo(5, 6))
}
```

В одной из команд `import` указан интернет-адрес — это значит, что в программе используется внешний пакет. В данном случае внешний пакет называется `simpleGitHub` и находится по адресу `github.com/mactsouk/go/simpleGitHub`.

Если вы попытаетесь сразу выполнить `getPackage.go`, то будете разочарованы:

```
$ go run getPackage.go
getPackage.go:5:2: cannot find package
"github.com/mactsouk/go/simpleGitHub" in any of:
  /usr/local/Cellar/go/1.9.1/libexec/src/github.com/mactsouk/go/simpleGitHub
    (from $GOROOT)
  /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub (from $GOPATH)
```

Как видно, необходимо установить недостающий пакет на ваш компьютер. Чтобы скачать этот пакет, нужно выполнить следующую команду:

```
$ go get -v github.com/mactsouk/go/simpleGitHub
github.com/mactsouk/go (download)
github.com/mactsouk/go/simpleGitHub
```

Загруженные файлы вы найдете в следующем каталоге:

```
$ ls -l ~/go/src/github.com/mactsouk/go/simpleGitHub/
total 8
-rw-r--r--  1 mtsouk  staff  66 Oct 17 21:47 simpleGitHub.go
```

Однако команда `go get` также компилирует пакет. Соответствующие файлы размещаются здесь:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
-rw-r--r--  1 mtsouk  staff 1050 Oct 17 21:47
/Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
```

Теперь без проблем выполняйте `getPackage.go`:

```
$ go run getPackage.go
11
```

Временные файлы загруженного Go-пакета можно удалить:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
cd /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub
rm -f simpleGitHub.test simpleGitHub.test.exe
rm -f /Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/
simpleGitHub.a
```

Аналогичным образом с помощью UNIX-команды `rm(1)` можно удалить весь загруженный и размещенный на локальном компьютере Go-пакет, чтобы удалить его исходный Go-код после использования `go clean`:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
$ rm -rf ~/go/src/github.com/mactsouk/go/simpleGitHub
```

После выполнения этих команд, чтобы использовать Go-пакет, вам придется снова его загрузить.

Стандартные потоки UNIX: stdin, stdout и stderr

В любой ОС UNIX есть три файла, которые постоянно открыты для своих процессов. Помните, что для UNIX все является файлом, даже если это принтер или мышь.

В качестве внутреннего представления для доступа ко всем открытым файлам UNIX использует *файловые дескрипторы* — положительные целые числа. Это гораздо красивее, чем длинные пути.

Таким образом, по умолчанию любая UNIX-система поддерживает три специальных стандартных имени файла: `/dev/stdin`, `/dev/stdout` и `/dev/stderr`, к которым также можно получить доступ, используя файловые дескрипторы 0, 1 и 2 соответственно. Эти три файловых дескриптора называются *стандартными потоками ввода, вывода и ошибок* соответственно. Кроме того, на компьютере с MacOS файловый дескриптор 0 может быть доступен как `/dev/fd/0`, а на компьютере с Debian Linux — как `/dev/fd/0` или `/dev/pts/0`.

Для доступа к стандартному потоку ввода Go использует `os.Stdin`, для доступа к стандартному потоку вывода — `os.Stdout` и для доступа к стандартному потоку ошибок — `os.Stderr`. Несмотря на то что для доступа к этим устройствам все равно можно использовать `/dev/stdin`, `/dev/stdout` и `/dev/stderr` или дескрипторы соответствующих файлов, все же лучше, безопаснее и легче придерживаться варианта с `os.Stdin`, `os.Stdout` и `os.Stderr`, предлагаемого Go.

Вывод результатов

Подобно C в UNIX, Go предлагает различные способы вывода результатов на экран. Все функции вывода, использованные в этом разделе, требуют подключения стандартного Go-пакета `fmt`. Функции будут продемонстрированы в программе `printing.go`, которую мы разделим на две части.

Самый простой способ вывести что-нибудь в Go — использовать функцию `fmt.Println()` или `fmt.Printf()`. У функции `fmt.Printf()` много общего с C-функцией `printf(3)`. Вместо `fmt.Print()` также можно использовать `fmt.Println()`. Основное различие между функциями `fmt.Print()` и `fmt.Println()` заключается в том, что последняя при каждом вызове автоматически добавляет в конец вывода символ новой строки.

В то же время наибольшим различием между `fmt.Println()` и `fmt.Printf()` является то, что в `fmt.Printf()` нужно указывать *спецификатор формата* для всего, что вы хотите вывести на экран, точно так же, как в C-функции `printf(3)`. С одной стороны, это означает, что вы лучше контролируете то, что делаете, но с другой — приходится писать больше кода. В Go эти спецификаторы формата называются *глаголами*. Подробнее о глаголах читайте на странице <https://golang.org/pkg/fmt/>.

Если перед выводом на экран нужно выполнить какое-либо форматирование или разместить несколько переменных, то лучше воспользоваться `fmt.Printf()`. Однако, если вы хотите вывести только одну переменную, то, возможно, лучше выбрать `fmt.Print()` или `fmt.Println()`, в зависимости от того, нужен ли вам в конце символ новой строки или нет.

Первая часть `printing.go` содержит следующий Go-код:

```
package main

import (
    "fmt"
)

func main() {
    v1 := "123"
    v2 := 123
    v3 := "Have a nice day\n"
    v4 := "abc"
```

В этой части мы видим импорт пакета `fmt` и определение четырех Go-переменных. Символ `\n`, использованный в `v3`, является символом разрыва строки. Однако если вы просто хотите вставить разрыв строки в выводимые данные, то вместо чего-то вроде `fmt.Print("\n")` можете просто вызвать `fmt.Println()` без аргументов.

Вторая часть `printing.go` выглядит так:

```
    fmt.Print(v1, v2, v3, v4)
    fmt.Println()
    fmt.Println(v1, v2, v3, v4)
    fmt.Print(v1, " ", v2, " ", v3, " ", v4, "\n")
    fmt.Printf("%s%d %s %s\n", v1, v2, v3, v4)
}
```

В этой части выводятся четыре переменные с использованием функций `fmt.Println()`, `fmt.Print()` и `fmt.Printf()`, чтобы лучше понять, чем эти функции различаются между собой.

Если выполнить `print.go`, то получим следующий вывод:

```
$ go run printing.go
123123Have a nice day
abc
123 123 Have a nice day
abc
123 123 Have a nice day
abc
123123 Have a nice day
abc
```

Как видим, функция `fmt.Println()`, в отличие от `fmt.Print()`, также добавляет пробел между выводимыми параметрами.

В результате вызов наподобие `fmt.Println(v1, v2)` эквивалентен `fmt.Print(v1, " ", v2, "\n")`.

Кроме `fmt.Println()`, `fmt.Print()` и `fmt.Printf()`, которые являются простейшими функциями для вывода данных на экран, существует также семейство S-функций, в которое входят функции `fmt.Sprintln()`, `fmt.Sprint()` и `fmt.Sprintf()`. Они используются для построения строк по заданному формату.

Наконец, существует семейство F-функций, в которое входят функции `fmt.Fprintln()`, `fmt.Fprint()` и `fmt.Fprintf()`. Они используются для записи в файлы с помощью `io.Writer`.



Подробнее об интерфейсах `io.Writer` и `io.Reader` вы прочитаете в главе 8.

В следующем разделе вы узнаете, как выводить данные на экран, используя стандартный поток вывода, что является довольно распространенной практикой в мире UNIX.

Использование стандартного потока вывода

Стандартный поток вывода более или менее эквивалентен выводу на экран. Однако его использование иногда требует применения функций, не входящих в пакет `fmt`, именно поэтому стандартному потоку вывода посвящен отдельный раздел.

Рассмотрим соответствующий метод в программе `stdOUT.go`, которую разделим на три части. Первая часть программы выглядит так:

```
package main
```

```
import (
    "io"
    "os"
)
```

Итак, в `stdOUT.go` вместо пакета `fmt` будет использоваться пакет `io`. Пакет `os` применяется для чтения программой аргументов командной строки и для доступа к `os.Stdout`.

Вторая часть `stdOUT.go` содержит следующий Go-код:

```
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
}
```

Переменная `myString` содержит текст, который будет выведен на экран. Этот текст является либо первым аргументом командной строки программы, либо, если программа выполнялась без аргументов командной строки, — жестко закодированным текстовым сообщением.

Третья часть программы выглядит следующим образом:

```
io.WriteString(os.Stdout, myString)
io.WriteString(os.Stdout, "\n")
}
```

В данном случае функция `io.WriteString()` работает так же, как `fmt.Print()`, однако принимает только два параметра. Первый из них — это файл, в который будут записываться результаты, в данном случае `os.Stdout`, а второй — строковая переменная.



Строго говоря, первый параметр функции `io.WriteString()` должен иметь тип `io.Writer`, что требует в качестве второго параметра срез байтов. Однако в данном случае строковая переменная отлично справляется со своей задачей. Подробнее о срезах вы узнаете в главе 3.

Запуск `stdOUT.go` приведет к следующему результату:

```
$ go run stdOUT.go
Please give me one argument!
$ go run stdOUT.go 123 12
123
```

Как видим, функция `io.WriteString()` отправляет содержимое своего второго параметра на экран, если первым параметром является `os.Stdout`.

Получение данных от пользователя

Есть три основных способа получения данных от пользователя: прочитать аргументы командной строки программы, предложить пользователю ввести данные или прочитать внешние файлы. В этом разделе описаны первые два способа. Чтобы узнать, как прочитать внешний файл, обратитесь к главе 8.

Что такое `:=` и `=`

Прежде чем продолжить, полезно узнать об использовании оператора `:=` и его отличии от `=`. Официальное название `:=` — *сокращенный оператор присваивания*. Сокращенный оператор присваивания можно использовать вместо объявления `var` с неявным типом.



Использование `var` в Go встречается редко. Ключевое слово `var` в Go-программах используется главным образом для объявления глобальных переменных, а также для объявления переменных без начального значения. Причина первого варианта заключается в том, что каждое утверждение, существующее вне кода функции, должно начинаться с ключевого слова, такого как `func` или `var`. Это означает, что сокращенный оператор присваивания нельзя использовать вне функции, потому что он там недоступен.

Оператор `:=` работает следующим образом:

```
m := 123
```

Результатом выполнения этого оператора станет создание целочисленной переменной с именем `m` и значением `123`.

Но если вы попытаетесь использовать `:=` для уже объявленной переменной, то компиляция завершится неудачно и выдаст следующее предельно ясное сообщение об ошибке:

```
$ go run test.go
# command-line-arguments
./test.go:5:4: no new variables on left side of :=
```

Возможно, у вас возникнет вопрос: что произойдет, если мы ожидаем от функции два значения или более и хотим использовать существующую переменную для одного из них? Какой оператор применять: `:=` или `=`? Ответ прост: оператор `:=`, как в следующем примере кода:

```
i, k := 3, 4
j, k := 1, 2
```

Поскольку переменная `j` впервые встречается во втором выражении, то мы использовали `:=`, даже если в первом выражении значение `k` уже определено.

Возможно, вам покажется скучным обсуждение подобных незначительных деталей, однако впоследствии, зная эти нюансы, вы сможете избежать различных ошибок!

Чтение стандартного потока ввода

Чтение данных из стандартного потока ввода рассмотрим на примере программы `stdIN.go`, которую разделим на две части. Первая часть выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "os"
)
```


В данном коде нам впервые в этой книге встретился пакет `bufio`.



Подробнее о пакете `bufio` вы узнаете в главе 8.

Обычно для ввода и вывода файлов используется пакет `bufio`, но в этой книге мы будем использовать в основном пакет `os`, так как в нем содержится много удобных функций; одно из самых востребованных свойств этого пакета — то, что он предоставляет доступ к аргументам командной строки Go-программы (`os.Args`).

Согласно официальному описанию пакета `os` в нем представлены функции, которые выполняют операции ОС. Сюда входят функции создания, удаления, переименования файлов и каталогов, а также функции распознавания полномочий доступа UNIX и других характеристик файлов и каталогов. Основным преимуществом пакета `os` является то, что он не зависит от платформы. Проще говоря, его функции будут работать на компьютерах как с UNIX, так и с Microsoft Windows.

Вторая часть `stdIN.go` содержит следующий Go-код:

```
func main() {
    var f *os.File
    f = os.Stdin
    defer f.Close()

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        fmt.Println(">", scanner.Text())
    }
}
```

Во-первых, здесь вызывается функция `bufio.NewScanner()`, которой в качестве параметра передается стандартный поток ввода (`os.Stdin`). Этот вызов возвращает переменную `bufio.Scanner`, которая используется функцией `Scan()` для построчного чтения из `os.Stdin`. Каждая прочитанная строка выводится на экран, после чего считывается следующая строка. Обратите внимание, что каждая строка, которую выводит программа, начинается с символа `>`.

Запуск `stdIN.go` даст нам следующий вывод:

```
$ go run stdIN.go
This is number 21
> This is number 21
This is Mihalis
> This is Mihalis
Hello Go!
> Hello Go!
Press Control + D on a new line to end this program!
> Press Control + D on a new line to end this program!
```

Как принято в UNIX, вы можете прервать чтение программой данных из стандартного потока ввода, нажав **Ctrl+D**.



Go-программы `stdIN.go` и `stdOUT.go` очень просты, однако не стоит их недооценивать: они пригодятся, когда мы будем обсуждать UNIX-каналы в главе 8.

Работа с аргументами командной строки

Способ, описанный в этом разделе, рассмотрим на примере Go-кода `cla.go`, который разделим на три части. Эта программа находит минимальный и максимальный из своих аргументов командной строки.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)
```

Важно понимать, что для того, чтобы программа получала аргументы командной строки, необходимо использовать пакет `os`. Кроме того, нам нужен еще один пакет, `strconv`, чтобы была возможность преобразовывать аргументы командной строки, заданные в виде строк, в арифметический тип данных.

Вторая часть программы выглядит так:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    min, _ := strconv.ParseFloat(arguments[1], 64)
    max, _ := strconv.ParseFloat(arguments[1], 64)
}
```

Здесь `cla.go` проверяет, есть ли у нас аргументы командной строки, контролируя длину `os.Args`. Дело в том, что для работы программы необходим как минимум один аргумент командной строки. Обратите внимание, что `os.Args` — это срез Go, содержащий значения типа `string`. Первый элемент среза — это имя исполняемой программы. Поэтому, чтобы инициализировать переменные `min` и `max`, нужно использовать второй элемент среза `os.Args`, имеющий тип `string`, индекс которого равен 1.

Важный момент: если вы ожидаете одно или несколько значений с плавающей точкой, это вовсе не значит, что пользователь предоставит вам именно корректные значения с плавающей точкой. Он может этого и не сделать — случайно или преднамеренно. Однако, поскольку мы пока еще не обсуждали обработку ошибок в Go, `cla.go` предполагает, что все аргументы командной строки имеют правильный формат и, следовательно, будут приемлемыми. Поэтому `cla.go` игнорирует значение типа `error`, возвращаемое функцией `strconv.ParseFloat()`, используя следующую инструкцию:

```
n, _ := strconv.ParseFloat(arguments[i], 64)
```

Предыдущий оператор говорит Go, что мы хотим получить только первое значение, возвращаемое `strconv.ParseFloat()`, и что нас не интересует второе значение, которое в данном случае является переменной типа `error`, — мы присваиваем это значение символу подчеркивания. Символ подчеркивания, который называют *пустым идентификатором*, является средством сброса значения в Go. Если Go-функция возвращает несколько значений, пустой идентификатор можно использовать несколько раз.



Игнорирование всех или некоторых значений, возвращаемых Go-функцией, особенно значений типа `error`, — очень опасный метод, который не следует использовать в продуктивном коде!

Третья часть содержит следующий Go-код:

```
for i := 2; i < len(arguments); i++ {
    n, _ := strconv.ParseFloat(arguments[i], 64)

    if n < min {
        min = n
    }
    if n > max {
        max = n
    }
}

fmt.Println("Min:", min)
fmt.Println("Max:", max)
}
```

Здесь используется цикл `for`. Он позволяет перебрать все элементы среза `os.Args`, который перед этим присвоен переменной `arguments`.

Выполнение `cla.go` приведет к следующему выводу:

```
$ go run cla.go -10 0 1
Min: -10
Max: 1
```

```
$ go run cla.go -10
Min: -10
Max: -10
```

Как и следовало ожидать, программа ведет себя некорректно, когда на вход подаются ошибочные данные; хуже всего то, что она не выводит никаких предупреждений, сообщающих пользователю об ошибке (или нескольких ошибках), возникших во время обработки аргументов командной строки:

```
$ go run cla.go a b c 10
Min: 0
Max: 10
```

Вывод ошибок

В этом разделе представлен метод передачи данных в *стандартный поток ошибок* UNIX, который помогает различить в UNIX реальные значения и вывод сообщений об ошибках.

Go-код для иллюстрации использования стандартного потока ошибок в Go содержится в файле `stdERR.go`. Мы его рассмотрим, разделив на две части. Поскольку запись в стандартный поток требует использования дескриптора файла, связанного со стандартным потоком ошибок, Go-код `stdERR.go` будет основан на Go-коде из `stdOUT.go`.

Первая часть программы выглядит так:

```
package main

import (
    "io"
    "os"
)

func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
}
```

До сих пор `stdERR.go` практически не отличается от `stdOUT.go`.

Вторая часть `stdERR.go` выглядит так:

```
io.WriteString(os.Stdout, "This is Standard output\n")
io.WriteString(os.Stderr, myString)
io.WriteString(os.Stderr, "\n")
}
```

Функция `io.WriteString()` вызывается два раза для записи в стандартный поток ошибок (`os.Stderr`) и еще один раз — для записи в стандартный поток вывода (`os.Stdout`).

Выполнение `stdERR.go` даст следующий результат:

```
$ go run stdERR.go
This is Standard output
Please give me one argument!
```

Этот вывод не поможет нам отличать данные, записанные в стандартный поток вывода, от данных, записанных в стандартный поток ошибок, что иногда очень полезно. Но если использовать оболочку `bash(1)`, то можно прибегнуть к одному приему, позволяющему отличать данные из стандартного потока вывода от данных из стандартного потока ошибок. Почти в каждой UNIX-оболочке эта функциональность реализована по-своему.

При применении `bash(1)` можно перенаправить стандартный поток ошибок в файл:

```
$ go run stdERR.go 2>/tmp/stdError
This is Standard output
$ cat /tmp/stdError
Please give me one argument!
```



Число, стоящее после имени UNIX-программы или после системного вызова, соответствует номеру раздела руководства, к которому относится страница этой программы или вызова. Большинство имен встречается только на одной странице справочника, следовательно, указывать номер раздела не обязательно. Однако есть имена, которые встречаются в нескольких разделах, поскольку имеют несколько значений, такие как `crontab(1)` и `crontab(5)`. Поэтому если вы попытаетесь получить справочную страницу имени с несколькими значениями, не указав номер раздела, то получите запись с наименьшим номером раздела.

Подобным образом можно отменить вывод ошибок, перенаправив их на устройство `/dev/null`, что соответствует указанию UNIX полностью игнорировать этот вывод:

```
$ go run stdERR.go 2>/dev/null
This is Standard output
```

В двух примерах мы перенаправили файловый дескриптор стандартного потока ошибок в определенный файл и в `/dev/null` соответственно. Если вы хотите сохранить данные стандартного потока вывода и стандартного потока ошибок в одном файле, то можете перенаправить файловый дескриптор стандартного потока

ошибок (2) в файловый дескриптор стандартного потока вывода (1). В следующей команде показан довольно распространенный в UNIX-системах способ:

```
$ go run stdERR.go >/tmp/output 2>&1
$ cat /tmp/output
This is Standard output
Please give me one argument!
```

Наконец, можно перенаправить и стандартный поток вывода, и стандартный поток ошибок на `/dev/null`:

```
$ go run stdERR.go >/dev/null 2>&1
```

Запись в журнальные файлы

Пакет `log` позволяет отправлять журнальные сообщения в системную службу журналирования UNIX-машины, а Go-пакет `syslog`, который является частью пакета `log`, позволяет определить *уровень журналирования* и *средство журналирования*, которое будет использовать ваша Go-программа.

Обычно большинство системных журнальных файлов в UNIX размещаются в каталоге `/var/log`. Однако журнальные файлы многих популярных сервисов, таких как Apache и Nginx, находятся в других местах, в зависимости от их конфигурации.

Вообще, использование журнальных файлов для записи некоторой информации считается более эффективной практикой, чем вывод тех же данных на экран, по двум причинам: во-первых, потому, что выходные данные, сохраняемые в файле, не теряются, а во-вторых, потому, что вы можете вести поиск и обрабатывать файлы журналов с помощью инструментов UNIX, таких как `grep(1)`, `awk(1)` и `sed(1)`; если же сообщения выводятся в окне терминала, это невозможно.

Пакет `log` предлагает множество функций для перенаправления вывода в системную службу журналирования UNIX-машины. В число этих функций входят `log.Printf()`, `log.Print()`, `log.Println()`, `log.Fatalf()`, `log.Fatalln()`, `log.Panic()`, `log.Panicln()` и `log.Panicf()`.



Обратите внимание, что функции журналирования могут быть очень полезны при отладке программ, особенно написанных на Go серверных процессах. Не стоит недооценивать их возможности.

Уровни журналирования

Уровень журналирования — это значение, которое определяет степень серьезности журнального сообщения. Существуют следующие уровни журналирования (в порядке возрастания серьезности): `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert` и `emerg`.

Средства журналирования

Средство журналирования подобно категории, используемой при регистрации информации. Средство журналирования принимает значения `auth`, `authpriv`, `cron`, `daemon`, `kern`, `lpr`, `mail`, `mark`, `news`, `syslog`, `user`, `UUCP`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6` или `local7` и определяется в файле `/etc/syslog.conf`, `/etc/rsyslog.conf` или другом соответствующем файле, в зависимости от процесса сервера, используемого для ведения журнала в операционной системе на данной UNIX-машине.

Это означает, что, если средство журналирования не определено и, соответственно, не поддерживается, отправленные в него журнальные сообщения могут быть проигнорированы и, следовательно, потеряны.

Серверы журналирования

На каждом компьютере с UNIX есть особый серверный процесс, который отвечает за получение журнальных сообщений и запись их в лог-файлы. Существуют различные разновидности серверов журналирования. Но только два из них используются в большинстве вариантов UNIX: `syslogd(8)` и `rsyslogd(8)`.

На машинах с macOS процесс ведения журнала называется `syslogd(8)`. На большинстве машин с Linux, напротив, используется `rsyslogd(8)` — улучшенная и более надежная версия `syslogd(8)`, которая была первой системной утилитой UNIX для регистрации журнальных сообщений.

Впрочем, независимо от вашего варианта UNIX или серверного процесса, используемого для ведения журнала, на всех компьютерах с UNIX журналирование работает одинаково и, следовательно, не влияет на Go-код, который вы напишете.

Файл конфигурации `rsyslogd(8)` обычно называется `rsyslog.conf` и находится в `/etc`. Содержимое файла конфигурации `rsyslog.conf`, без учета строк с комментариями и строк, начинающихся с `$`, может выглядеть следующим образом:

```
$ grep -v '^#' /etc/rsyslog.conf | grep -v '^$' | grep -v '^\\$'
auth,authpriv.*                /var/log/auth.log
*.*;auth,authpriv.none         -/var/log/syslog
daemon.*                       -/var/log/daemon.log
kern.*                         -/var/log/kern.log
lpr.*                          -/var/log/lpr.log
mail.*                         -/var/log/mail.log
user.*                         -/var/log/user.log
mail.info                     -/var/log/mail.info
mail.warn                     -/var/log/mail.warn
mail.err                      /var/log/mail.err
news.crit                     /var/log/news/news.crit
```

```

news.err                /var/log/news/news.err
news.notice              -/var/log/news/news.notice
*.=debug;\
    auth,authpriv.none;\
    news.none;mail.none    -/var/log/debug
*.=info;*.=notice;*.=warn;\
    auth,authpriv.none;\
    cron,daemon.none;\
    mail,news.none        -/var/log/messages
*.emerg                  :omusrmsg:*
daemon.*;mail.*;\
    news.err;\
    *.=debug;*.=info;\
    *.=notice;*.=warn    |/dev/xconsole
local7.* /var/log/cisco.log

```

Таким образом, для того чтобы передать журнальную информацию в `/var/log/cisco.log`, нужно использовать средство журналирования *local7*. Символ звездочки после имени средства журналирования дает серверу журналирования указание перехватывать все уровни журналирования, которые поступают в средство журналирования *local7*, и записывать их в `/var/log/cisco.log`.

Сервер `syslogd(8)` имеет довольно похожий файл конфигурации, обычно это `/etc/syslog.conf`. В macOS High Sierra файл `/etc/syslog.conf` практически пуст и заменен на `/etc/asl.conf`. Тем не менее, логика конфигурации у `/etc/syslog.conf`, `/etc/rsyslog.conf` и `/etc/asl.conf` одинакова.

Пример Go-программы, которая записывает информацию в журнальные файлы

Рассмотрим использование пакетов `log` и `log/syslog` для записи в файлы системного журнала на примере Go-кода, представленного в файле `logFiles.go`.



Обратите внимание: пакет `log/syslog` не реализован в версии Go для Microsoft Windows.

Первая часть `logFiles.go` выглядит так:

```

package main

import (
    "fmt"
    "log"
    "log/syslog"
    "os"

```



```

    "path/filepath"
)

func main() {
    programName := filepath.Base(os.Args[0])
    syslog, err := syslog.New(syslog.LOG_INFO|syslog.LOG_LOCAL7, programName)

```

Первым параметром функции `syslog.New()` является приоритет, который представляет собой соединение средства журналирования и уровня журналирования. Так, приоритет `LOG_NOTICE | LOG_MAIL`, который выбран в качестве примера, будет отправлять сообщения уровня журналирования `NOTICE` в средство журналирования `MAIL`.

В результате этот код устанавливает такой режим журналирования по умолчанию: средство журналирования `local7` и уровень журналирования `info`. Вторым параметром функции `syslog.New()` — это имя процесса, который будет отображаться в журналах в качестве отправителя сообщения. Вообще, рекомендуется использовать настоящее имя исполняемого файла, чтобы впоследствии можно было легко найти нужную информацию в файлах журналов.

Вторая часть программы содержит следующий Go-код:

```

    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }
    log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")

```

После вызова функции `syslog.New()` нужно проверить переменную типа `error`, которая возвращается этой функцией, чтобы убедиться, что все в порядке. Если это так, тогда значение переменной типа `error` равно `nil` и можно вызвать функцию `log.SetOutput()`, которая задает приемник вывода для записи в журнал по умолчанию, — в данном случае это созданный нами ранее регистратор журнала (`sysLog`). Затем можно использовать функцию `log.Println()` для отправки информации на сервер журнала.

Третья часть `logFiles.go` содержит следующий код:

```

    syslog, err = syslog.New(syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Println("LOG_MAIL: Logging in Go!")
    fmt.Println("Will you see this?")
}

```

Как видно из последней части кода, мы можем изменять конфигурацию ведения журнала в своих программах столько раз, сколько захотим, и при этом все равно можем использовать `fmt.Println()` для вывода данных на экран.

При выполнении `logFiles.go` на экран компьютера с Debian Linux будут выведены следующие данные:

```
$ go run logFiles.go
Broadcast message from systemd-journald@mail (Tue 2017-10-17 20:06:08 EEST):
logFiles[23688]: Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
Message from syslogd@mail at Oct 17 20:06:08 ...
Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
Will you see this?
```

Выполнение того же Go-кода на компьютере с macOS High Sierra приведет к следующим результатам:

```
$ go run logFiles.go
Will you see this?
```

Имейте в виду, что на большинстве UNIX-машин информация журналов хранится в нескольких файлах. Это также относится и к машине с Debian Linux, использованной в этом разделе. В результате `logFiles.go` отправляет выходные данные в несколько файлов журнала, в чем можно убедиться с помощью следующих команд оболочки:

```
$ grep LOG_MAIL /var/log/mail.log
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
$ grep LOG_LOCAL7 /var/log/cisco.log
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
$ grep LOG_ /var/log/syslog
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
```

Как видно из результатов, сообщение оператора `log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")` записано в два файла: `/var/log/cisco.log` и `/var/log/syslog`, тогда как сообщение оператора `log.Println("LOG_MAIL: Logging in Go!")` попало в `/var/log/syslog` и `/var/log/mail.log`.

Из этого раздела важно запомнить следующее: если сервер журналирования на UNIX-машине не настроен на перехват всех средств журналирования, то некоторые из отправляемых на него записей журнала могут быть удалены без каких-либо предупреждений.

Функция `log.Fatal()`

В этом разделе мы познакомимся с тем, как работает функция `log.Fatal()`. Она используется тогда, когда происходит что-то действительно очень плохое и вы просто хотите поскорее выйти из программы, предварительно отправив сообщение о сложной ситуации.

Использование `log.Fatal()` будет проиллюстрировано в программе `logFatal.go`, которая содержит следующий Go-код:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    syslog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(syslog)
    }
    log.Fatal(syslog)
    fmt.Println("Will you see this?")
}
```

Выполнение `logFatal.go` приведет к следующему выводу:

```
$ go run logFatal.go
exit status 1
```

Легко понять, что при использовании `log.Fatal()` Go-программа завершается в том месте, где была вызвана функция `log.Fatal()`. Именно поэтому вы не увидите вывод оператора `fmt.Println("Will you see this?")`.

Однако в соответствии с параметрами вызова `syslog.New()` при этом в файл журнала `/var/log/mail.log`, связанный с почтой, добавлена запись:

```
$ grep "Some program" /var/log/mail.log
Jan 10 21:29:34 iMac Some program![7123]: 2019/01/10 21:29:34 &{17 Some program!
iMac.local {0 0} 0xc00000c220}
```

Функция `log.Panic()`

Возможны ситуации, когда программа завершается сбоем и вы хотите получить как можно больше информации о нем.

В такие сложные моменты одним из решений станет использование `log.Panic()` — разновидности функций журналирования, работа которой будет продемонстрирована в этом разделе на примере Go-кода из файла `logPanic.go`.

Go-код `logPanic.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    syslog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(syslog)
    }

    log.Panic(syslog)
    fmt.Println("Will you see this?")
}
```

Выполнение `logPanic.go` в macOS Mojave приведет к следующим результатам:

```
$ go run logPanic.go
panic: &{17 Some program! iMac.local
{0 0} 0xc0000b21e0}
goroutine 1 [running]:
log.Panic(0xc00004ef68, 0x1, 0x1)
    /usr/local/Cellar/go/1.11.4/libexec/src/log/log.go:326 +0xc0
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch01/logPanic.go:17 +0xd6
exit status 2
```

Выполнение той же программы в Debian Linux с версией Go 1.3.3 приведет к следующему:

```
$ go run logPanic.go
panic: &{17 Some program! mail
{0 0} 0xc2080400e0}
goroutine 16 [running]:
runtime.panic(0x4ec360, 0xc208000320)
    /usr/lib/go/src/pkg/runtime/panic.c:279 +0xf5
log.Panic(0xc208055f20, 0x1, 0x1)
    /usr/lib/go/src/pkg/log/log.go:307 +0xb6
main.main()
    /usr/lib/go/src/pkg/log/log.go:307 +0xb6
```

```

/home/mtsouk/Desktop/masterGo/ch/ch1/code/logPanic.go:17 +0x169
goroutine 17 [runnable]:
runtime.MHeap_Scavenger()
/usr/lib/go/src/pkg/runtime/mheap.c:507
runtime.goexit()
/usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 18 [runnable]:
bgsweep()
/usr/lib/go/src/pkg/runtime/mgc0.c:1976
runtime.goexit()
/usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 19 [runnable]:
runfinq()
/usr/lib/go/src/pkg/runtime/mgc0.c:2606
runtime.goexit()
/usr/lib/go/src/pkg/runtime/proc.c:1445
exit status 2

```

Таким образом, `log.Panic()` выводит дополнительную низкоуровневую информацию, которая, по идее, должна помочь разрешить сложную ситуацию, возникшую в Go-коде.

Как и `log.Fatal()`, функция `log.Panic()` добавит запись в соответствующий файл журнала и немедленно прекратит работу Go-программы.

Запись в специальный журнальный файл

Иногда нужно просто записать журнальные сообщения в выбранный вами файл. Это может понадобиться по многим причинам. Например, для записи данных отладки, которые бывают слишком большими. Или если вы не хотите записывать их в системные журнальные файлы, а собираетесь хранить данные в собственных журналах, отдельно от системных данных, а потом, допустим, передать их куда-то или сохранить в базе данных. Или же вы хотите хранить свои данные в другом формате. В этом подразделе вы узнаете, как организовать запись в специальный журнальный файл.

Назовем нашу Go-утилиту `customLog.go` и будем записывать данные в журнальный файл `/tmp/mGo.log`.

Разделим Go-код для `customLog.go` на три части. Первая часть выглядит следующим образом:

```

package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

```

Путь к журнальному файлу в `customLog.go` задан жестко как значение глобальной переменной с именем `LOGFILE`. Для наших целей в примере журнальный файл находится в каталоге `/tmp`, что не является обычным местом для хранения данных, поскольку, как правило, каталог `/tmp` очищается после каждой перезагрузки системы. Однако на данном этапе это избавит нас от необходимости запускать `customLog.go` с полномочиями пользователя `root` и не потребует размещать лишние файлы в системных каталогах. Если позже вы решите использовать код `customLog.go` в реальном приложении, вам следует изменить путь на более правильный.

Вторая часть `customLog.go` выглядит следующим образом:

```
func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

Здесь мы создаем новый журнальный файл, используя функцию `os.OpenFile()` с необходимыми правами доступа к UNIX-файлам (`0644`).

Последняя часть `customLog.go` выглядит следующим образом:

```
    iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)

    iLog.SetFlags(log.LstdFlags)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")
}
```

Если вы посмотрите на страницу документации пакета `log`, которую можно найти по адресу <https://golang.org/pkg/log/>, то увидите, что функция `SetFlags` позволяет устанавливать выходные флаги (варианты) для текущего средства журналирования. По умолчанию функция предлагает значения `LstdFlags: Ldate` и `Ltime`, то есть в каждой записи журнала, которая записывается в журнальный файл, будут указаны текущая дата и время.

При выполнении `customLog.go` не будет генерироваться вывод на экран. Однако если дважды выполнить эту программу, получим следующее содержимое файла `/tmp/mGo.log`:

```
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
CustomLog 2019/01/10 18:16:09 Another log entry!
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
CustomLog 2019/01/10 18:16:09 Another log entry!
CustomLog 2019/01/10 18:16:17 Hello there!
CustomLog 2019/01/10 18:16:17 Another log entry!
```

Вывод номеров строк в записях журнала

В этом разделе вы узнаете, как указать в журнальном файле номер строки исходного файла, который выполнил инструкцию, сделавшую запись в журнал. Для этого рассмотрим Go-программу из файла `customLogLineNumber.go`, разделив ее на две части. Первая часть выглядит так:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

Как видим, здесь пока нет особых отличий от кода `customLog.go`.

Остальной Go-код из файла `customLogLineNumber.go` выглядит так:

```
iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)
iLog.SetFlags(log.LstdFlags | log.Lshortfile)
iLog.Println("Hello there!")
iLog.Println("Another log entry!")
}
```

Всю магию выполняет оператор `iLog.SetFlags(log.LstdFlags | log.Lshortfile)`, который, кроме `log.LstdFlags`, также активизирует флаг `log.Lshortfile`. Последний добавляет в строку записи журнала полное имя файла, а также номер строки Go-оператора, который создал эту запись.

Выполнение `customLogLineNumber.go` не выводит данные на экран. Но после двух запусков файла `customLogLineNumber.go` содержимое файла журнала `/tmp/mGo.log` будет примерно таким:

```
$ go run customLogLineNumber.go
$ cat /tmp/mGo.log
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:26: Hello there!
CustomLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:27: Another log entry!
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:26: Hello there!
```

```
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:27: Another log entry!  
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:26: Hello there!  
CustomLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:27: Another log entry!
```

Как видим, использование длинных имен для утилит командной строки затрудняет чтение журнальных файлов.



В главе 2 вы узнаете, как использовать ключевое слово `defer` для того, чтобы Go-функции генерировали более красивые сообщения журнала.

Обработка ошибок в Go

Ошибки и обработка ошибок — две очень важные темы для Go. Go так любит сообщения об ошибках, что имеет особый тип данных для ошибок — `error`. Это также означает, что вы можете легко создавать собственные сообщения об ошибках, если окажется, что то, что уже есть в Go, вам не подходит.

Скорее всего, при разработке Go-пакетов вам придется создавать и обрабатывать собственные сообщения об ошибках.

Обратите внимание, что само по себе ошибочное состояние — это одно, а решение о том, как реагировать на это состояние, — совсем другое. Проще говоря, не все ошибочные состояния одинаковы: для одних ошибочных состояний может потребоваться немедленная остановка программы, а в других ситуациях можно ограничиться выводом предупреждающего сообщения для пользователя и продолжить выполнение программы. Принимая решение о том, что делать с каждым значением типа `error`, которое может получить программа, разработчику следует руководствоваться здравым смыслом.



Ошибки в Go не похожи на исключения или ошибки в других языках программирования; это обычные объекты Go, которые возвращаются из функций или методов, подобно любым другим значениям.

Тип данных `error`

Существует множество сценариев, в которых возникает необходимость обработать новый случай ошибки при разработке собственных Go-приложений. Тип данных `error` для того и предназначен, чтобы помочь вам создавать собственные ошибки.

В этом разделе вы научитесь создавать переменные типа `error`. Как вы увидите, для создания переменной типа `error` необходимо вызвать функцию `New()` из стандартного Go-пакета `errors`.

Пример Go-кода для иллюстрации этого процесса представлен в файле `newError.go`. Разделим его на две части. Первая часть программы выглядит так:

```
package main

import (
    "errors"
    "fmt"
)

func returnError(a, b int) error {
    if a == b {
        err := errors.New("Error in returnError() function!")
        return err
    } else {
        return nil
    }
}
```

Здесь происходит много интересного. Прежде всего, впервые в этой книге встречается определение Go-функции, отличной от `main()`. Имя этой новой функции — `returnError()`. Кроме того, здесь показано, как работает функция `errors.New()`, которая принимает в качестве параметра значение типа `string`. Наконец, если функция должна вернуть переменную типа `error`, но ошибка не произошла, то возвращается `nil`.



Подробнее о типах Go-функций вы узнаете в главе 6.

Вторая часть `newError.go` выглядит так:

```
func main() {
    err := returnError(1, 2)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    err = returnError(10, 10)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    if err.Error() == "Error in returnError() function!" {
        fmt.Println("!!!")
    }
}
```

Как видно из кода, большая часть времени уходит на проверки того, равна ли переменная типа `error` со значением `nil`, после чего можно действовать соответственно. Здесь также продемонстрировано использование функции `errors.Error()`, которая позволяет преобразовать переменную типа `error` в тип `string`. Эта функция позволяет сравнивать переменную типа `error` с переменной типа `string`.



На компьютерах с UNIX рекомендуется передавать сообщения об ошибках в службу журналирования, особенно если Go-программа является сервером или другим критически важным приложением. Однако примеры кода, представленные в этой книге, не всегда будут следовать данному принципу, чтобы избежать засорения ваших журнальных файлов лишними данными.

Выполнение `newError.go` приведет к следующим результатам:

```
$ go run newError.go
returnError() ended normally!
Error in returnError() function!
!!
```

Если вы попытаетесь сравнить переменную типа `error` с переменной типа `string` без предварительного преобразования переменной типа `error` в `string`, компилятор Go выдаст следующее сообщение об ошибке:

```
# command-line-arguments
./newError.go:33:9: invalid operation: err == "Error in returnError()
function!" (mismatched types error and string)
```

Обработка ошибок

Обработка ошибок является очень важной частью Go. Поскольку почти все функции Go возвращают сообщение об ошибке или `nil`, у Go есть способ сообщить, возникло ли состояние ошибки при выполнении функции. Скорее всего, вам вскоре надоест встречать следующий Go-код не только в этой книге, но и в любой другой Go-программе, найденной в Интернете:

```
if err != nil {
    fmt.Println(err)
    os.Exit(10)
}
```



Пожалуйста, не путайте обработку ошибок, сопровождаемую выводом сообщений, с выводом сообщения в стандартный поток вывода ошибок: это совершенно разные вещи. Первая из них связана с Go-кодом, который обрабатывает ошибки, тогда как вторая — с записью чего-либо в стандартный дескриптор файла ошибок.

Представленный пример кода выводит сгенерированное сообщение об ошибке на экран и выходит из программы посредством функции `os.Exit()`. Обратите внимание: чтобы выйти из программы, можно также поставить ключевое слово `return` внутри функции `main()`. Вообще, вызов `os.Exit()` из функции, если только это не `main()`, считается плохой практикой. Функции, отличные от `main()`, как правило, перед выходом возвращают сообщение об ошибке, которое обрабатывается вызывающей функцией.

Если вы хотите отправить сообщение об ошибке не на экран, а в службу журналирования, нужно изменить предыдущий Go-код следующим образом:

```
if err != nil {
    log.Println(err)
    os.Exit(10)
}
```

Наконец, есть и другой вариант этого кода, который используется, когда случается что-то очень плохое и вы хотите срочно завершить программу:

```
if err != nil {
    panic(err)
    os.Exit(10)
}
```

Panic — это встроенная Go-функция, которая останавливает выполнение программы и начинает паниковать! Если вам приходится слишком часто использовать `panic`, возможно, стоит пересмотреть вашу реализацию Go-кода. Лучше избегать ситуаций паники, по возможности заменяя ее обработкой ошибок.

Как вы узнаете из следующей главы, в Go есть функция `recover`, которая выручит вас в некоторых неприятных ситуациях. Из главы 2 вы узнаете больше о возможностях этого дуэта функций — `panic` и `recover`.

А теперь пора познакомиться с Go-программой, которая не только обрабатывает сообщения об ошибках, генерируемые стандартными Go-функциями, но и создает собственное сообщение об ошибке. Эта программа называется `errors.go`. Разделим ее на пять частей. Как вы увидите, утилита `errors.go` пытается улучшить функциональность программы `cla.go`, о которой уже говорилось в данной главе. Теперь мы добавим проверку того, соответствуют ли ее аргументы командной строки формату чисел с плавающей точкой.

Первая часть программы выглядит так:

```
package main

import (
    "errors"
    "fmt"
    "os"
    "strconv"
)
```

Эта часть `error.go` содержит ожидаемые операторы `import`.

Вторая часть `error.go` содержит следующий Go-код:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    var err error = errors.New("An error")
    k := 1
    var n float64
```

Здесь создается новая переменная типа `error` с именем `err`, чтобы инициализировать ее собственным значением.

Третья часть программы выглядит так:

```
for err != nil {
    if k >= len(arguments) {
        fmt.Println("None of the arguments is a float!")
        return
    }
    n, err = strconv.ParseFloat(arguments[k], 64)
    k++
}

min, max := n, n
```

Это самая сложная часть программы: если первый аргумент командной строки не является корректным числом с плавающей точкой, нужно проверить следующий аргумент и продолжать проверку до тех пор, пока не встретится аргумент командной строки подходящего формата. Если ни один из аргументов командной строки не соответствует формату числа с плавающей точкой, `error.go` прекращает работу и выводит на экран сообщение об ошибке. Вся проверка выполняется путем исследования значения типа `error`, которое возвращается функцией `strconv.ParseFloat()`. Весь этот код предназначен лишь для точной инициализации переменных `min` и `max`.

Четвертая часть программы содержит следующий Go-код:

```
for i := 2; i < len(arguments); i++ {
    n, err := strconv.ParseFloat(arguments[i], 64)
    if err == nil {
        if n < min {
            min = n
        }
        if n > max {
            max = n
        }
    }
}
```

Здесь мы просто обрабатываем все аргументы командной строки правильного формата, чтобы найти среди них минимальное и максимальное значения с плавающей точкой.

Наконец, последняя часть кода программы выводит на экран текущие значения переменных `min` и `max`:

```
    fmt.Println("Min:", min)
    fmt.Println("Max:", max)
}
```

Как видим из Go-кода, представленного в файле `errors.go`, большая часть кода посвящена обработке ошибок, а не выполнению того, для чего, собственно, предназначена программа. К сожалению, это относится к большинству современных программ, разработанных на Go, а также на большинстве других языков программирования.

Если выполнить `error.go`, то получим следующий вывод:

```
$ go run errors.go a b c
None of the arguments is a float!
$ go run errors.go b c 1 2 3 c -1 100 -200 a
Min: -200
Max: 100
```

Использование Docker

В последнем разделе этой главы вы узнаете, как можно использовать образ Docker для компиляции и выполнения Go-кода внутри образа Docker.

Возможно, вы знаете, что в Docker все начинается с образа Docker; вы можете создать собственный образ Docker с нуля или начать с уже существующего. В этом разделе мы загрузим базовый образ Docker с Docker Hub и продолжим сборку Go-программы `Hello World!` внутри этого образа.

Содержимое `Dockerfile` будет использоваться следующим образом:

```
FROM golang:alpine

RUN mkdir /files
COPY hw.go /files
WORKDIR /files
RUN go build -o /files/hw hw.go
ENTRYPOINT ["/files/hw"]
```

Здесь в первой строке определяется образ Docker, который будет использоваться. Остальные три команды создают в образе Docker новый каталог, копируют в образ Docker файл (`hw.go`) из текущего пользовательского каталога и соответственно изменяют текущий рабочий каталог образа Docker. Последние две команды соз-

дают двоичный исполняемый файл из исходного Go-файла и задают путь к этому двоичному файлу, который будет выполняться при запуске образа Docker.

Итак, как использовать этот Dockerfile? Если файл с именем `hw.go` существует и находится в текущем рабочем каталоге, можно создать новый образ Docker следующим образом:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
alpine: Pulling from library/golang
cd784148e348: Pull complete
7e273b0dfc44: Pull complete
952c3806fd1a: Pull complete
ee1f873f86f9: Pull complete
7172cd197d12: Pull complete
Digest:
sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb018ad77d647
Status: Downloaded newer image for golang:alpine
----> f56365ec0638
Step 2/6 : RUN mkdir /files
----> Running in 18fa7784d82c
Removing intermediate container 18fa7784d82c
----> 9360e95d7cb4
Step 3/6 : COPY hw.go /files
----> 680517bc4aa3
Step 4/6 : WORKDIR /files
----> Running in f3f678fcc38d
Removing intermediate container f3f678fcc38d
----> 640117aea82f
Step 5/6 : RUN go build -o /files/hw hw.go
----> Running in 271cae1fa7f9
Removing intermediate container 271cae1fa7f9
----> dc7852b6aeeb
Step 6/6 : ENTRYPOINT ["/files/hw"]
----> Running in cdadf286f025
Removing intermediate container cdadf286f025
----> 9bec016712c4
Successfully built 9bec016712c4
Successfully tagged go_hw:v1
```

Имя нового образа Docker — `go_hw:v1`.

Если на вашем компьютере уже есть образ Docker `golang:alpine`, то результат этой команды будет таким:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
----> f56365ec0638
Step 2/6 : RUN mkdir /files
```

```

---> Running in 982e6883bb13
Removing intermediate container 982e6883bb13
---> 0632577d852c
Step 3/6 : COPY hw.go /files
---> 68a0feb2e7dc
Step 4/6 : WORKDIR /files
---> Running in d7d4d0c846c2
Removing intermediate container d7d4d0c846c2
---> 6597a7cb3882
Step 5/6 : RUN go build -o /files/hw hw.go
---> Running in 324400d532e0
Removing intermediate container 324400d532e0
---> 5496dd3d09d1
Step 6/6 : ENTRYPOINT ["/files/hw"]
---> Running in bbd24840d6d4
Removing intermediate container bbd24840d6d4
---> 5a0d2473aa96
    Successfully built 5a0d2473aa96
    Successfully tagged go_hw:v1

```

Чтобы убедиться, что образ Docker `go_hw:v1` действительно существует на вашем компьютере, нужно сделать следующее:

```

$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
go_hw	v1	9bec016712c4	About a minute ago	312MB
golang	alpine	f56365ec0638	11 days ago	310MB

В файле `hw.go` содержится следующий код:

```

package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World!")
}

```

Образ Docker, который находится на локальном компьютере, можно использовать следующим образом:

```

$ docker run go_hw:v1
Hello World!

```

Существуют и другие, более сложные способы выполнения образа Docker, но для столь наивного образа Docker это самый простой способ его использования.

При желании можно передать (push) образ Docker в реестр Docker, размещенный в Интернете, чтобы впоследствии иметь возможность извлекать его оттуда (pull).

Таким местом может быть Docker Hub — при условии, что у вас есть учетная запись Docker Hub, создать которую легко, и это бесплатно. Итак, для того чтобы отправить созданный нами образ в Docker Hub, нужно создать в Docker Hub учетную запись, затем выполнить на вашей UNIX-машине следующие команды:

```
$ docker login
Authenticating with existing credentials...
Login Succeeded
$ docker tag go_hw:v1 "mactsouk/go_hw:v1"
$ docker push "mactsouk/go_hw:v1"
The push refers to repository [docker.io/mactsouk/go_hw]
bdb6946938e3: Pushed
99e21c42e35d: Pushed
0257968d27b2: Pushed
e121936484eb: Pushed
61b145086eb8: Pushed
789935042c6f: Pushed
b14874cfef59: Pushed
7bff100f35cb: Pushed
v1: digest:
sha256:c179d5d48a51b74b0883e582d53bf861c6884743eb51d9b77855949b5d91dd
e1 size: 1988
```

Первая команда необходима для входа в Docker Hub, ее достаточно выполнить один раз. Команда `docker tag` нужна для указания имени, под которым локальный образ будет храниться в Docker Hub. Эту команду нужно выполнить перед командой `push docker`. Последняя команда отправляет желаемый образ Docker в Docker Hub и генерирует расширенное сообщение о результатах. Если вы сделаете свой образ Docker общедоступным, то любой желающий сможет получить его из Docker Hub и использовать.

Есть несколько способов удалить один или несколько образов Docker с локальной UNIX-машины. Один из них — воспользоваться `IMAGE ID` образа Docker:

```
$ docker rmi 5a0d2473aa96 f56365ec0638
Untagged: go_hw:v1
Deleted:
sha256:5a0d2473aa96bcdafbef92751a0e1c1bf146848966c8c971f462eb1eb242d2a6
Deleted:
sha256:5496dd3d09d13c63bf7a9ac52b90bb812690cdfd33cfc3340509f9bfe6215c48
Deleted:
sha256:598c4e474b123eccb84f41620d2568665b88a8f176a21342030917576b9d82a8
Deleted:
sha256:6597a7cb3882b73855d12111787bd956a9ec3abb11d9915d32f2bba4d0e92ec6
Deleted:
sha256:68a0feb2e7dc5a139eaa7ca04e54c20e34b7d06df30bcd4934ad6511361f2cb8
Deleted:
sha256:c04452ea9f45d85a999bdc54b55ca75b6b196320c021d777ec1f766d115aa514
Deleted:
sha256:0632577d852c4f9b66c0eff2481ba06c49437e447761d655073eb034fa0ac333
```



```
Deleted:
sha256:52efd0fa2950c8f3c3e2e44fbc4eb076c92c0f85fff46a07e060f5974c1007a9
Untagged: golang:alpine
Untagged:
golang@sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb018ad77d647
Deleted:
sha256:f56365ec0638b16b752af4bf17e6098f2fda027f8a71886d6849342266cc3ab7
Deleted:
sha256:d6a4b196ed79e7ff124b547431f77e92dce9650037e76da294b3b3aded709bdd
Deleted:
sha256:f509ec77b9b2390c745afd76cd8dd86977c86e9ff377d5663b42b664357c3522
Deleted:
sha256:1ee98fa99e925362ef980e651c5a685ad04cef41dd80df9be59f158cf9e52951
Deleted:
sha256:78c8e55f8cb4c661582af874153f88c2587a034ee32d21cb57ac1fef51c6109e
Deleted:
sha256:7bfff100f35cb359a368537bb07829b055fe8e0b1cb01085a3a628ae9c187c7b8
```



Работа с Docker — огромная и действительно важная тема, к которой мы вернемся еще не раз.

Упражнения и ссылки

- ☐ Посетите сайт Go: <https://golang.org/>.
- ☐ Посетите сайт Docker: <https://www.docker.com/>.
- ☐ Посетите сайт Docker Hub: <https://hub.docker.com/>.
- ☐ Посетите Go 2 Draft Designs: <https://blog.golang.org/go2draft>.
- ☐ Посетите сайт документации по Go: <https://golang.org/doc/>.
- ☐ Почитайте документацию пакета `log`: <https://golang.org/pkg/log/>.
- ☐ Почитайте документацию пакета `log/syslog`: <https://golang.org/pkg/log/syslog/>.
- ☐ Почитайте документацию пакета `os`: <https://golang.org/pkg/os/>.
- ☐ Посетите <https://golang.org/cmd/gofmt/> — страницу документации инструмента `gofmt`, который используется для форматирования Go-кода.
- ☐ Напишите Go-программу, которая вычисляет сумму всех аргументов командной строки, которые являются действительными числами.
- ☐ Напишите Go-программу, вычисляющую среднее значение всех чисел с плавающей запятой, переданных программе в качестве аргументов командной строки.
- ☐ Напишите Go-программу, которая считывает целые числа до тех пор, пока не встретит во входных данных слово `END`.

- ❑ Можете ли вы изменить программу `customLog.go` так, чтобы данные журнала записывались одновременно в два файла журнала? Возможно, для этого вам придется обратиться к главе 8.
- ❑ Если вы работаете на компьютере Mac, почитайте описание редактора TextMate по адресу <http://macromates.com/>, а также редактора BBEEdit по адресу <https://www.barebones.com/products/bbedit/>.
- ❑ Посетите страницу документации пакета `fmt`: <https://golang.org/pkg/fmt/>, чтобы больше узнать о «глаголах» форматирования и доступных функциях.
- ❑ Приглашаю вас посетить страницу <https://blog.Golang.Org/why-generics>, чтобы больше узнать о Go и Generics.

Резюме

В этой главе приведена общая информация о языке Go, а также раскрыты многие интересные темы: компиляция Go-кода, работа Go со стандартными потоками ввода, вывода и ошибок, обработка аргументов командной строки, вывод данных на экран и использование службы журналирования UNIX-систем, а также обработка ошибок. Все эти вопросы лежат в основе изучения Go.

Следующая глава посвящена деталям внутренней реализации Go, в том числе сборке мусора, компилятору Go, вызову C-кода из Go, ключевому слову `defer`, Go-ассемблеру и WebAssembly, а также функциям `panic` и `recover`.

2

Go изнутри

Все функции Go, которые вы изучили в предыдущей главе, очень удобны, и вы будете постоянно их использовать. Однако нет ничего более полезного, чем способность видеть и понимать, что при этом происходит внутри Go.

В этой главе вы познакомитесь со сборщиком мусора Go и узнаете, как он работает. Кроме того, вы научитесь вызывать код на C из программ на Go, что иногда незаменимо. Однако вам не придется обращаться к этой функции слишком часто, поскольку Go — язык программирования с большими возможностями.

Кроме того, вы узнаете, как вызывать код на Go из программ на C, научитесь использовать функции `panic()` и `recover()` и ключевое слово `defer`.

В этой главе рассмотрим следующие темы:

- ❑ компилятор Go;
- ❑ как работает сборка мусора в Go;
- ❑ как проверить работу сборщика мусора;
- ❑ вызов кода на C из программы на Go;
- ❑ вызов кода на Go из программы на C;
- ❑ функции `panic()` и `recover()`;
- ❑ пакет `unsafe`;
- ❑ удобное, но коварное ключевое слово `defer`;
- ❑ Linux-утилита `strace(1)`;
- ❑ утилита `dtrace(1)` из систем FreeBSD, включая macOS Mojave;
- ❑ где найти информацию о вашей среде Go;
- ❑ построение деревьев узлов (node trees) с помощью Go;
- ❑ генерация кода WebAssembly из Go;
- ❑ Go-ассемблер.

Компилятор Go

Компилятор Go запускается с помощью инструмента `go`. Этот инструмент делает гораздо больше, чем просто создает исполняемые файлы.



Используемый в этом разделе файл `unsafe.go` не содержит никакого особенного кода — представленные здесь команды будут работать с любым корректным исходным файлом Go.

Для того чтобы скомпилировать исходный файл Go, нужно воспользоваться командой¹ `go tool compile`. В результате вы получите *объектный файл* — файл с расширением `.o`. Пример компиляции показан ниже, в виде следующих команд, выполненных на компьютере с macOS Mojave:

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:39 unsafe.o
$ file unsafe.o
unsafe.o: current ar archive
```

Объектный файл — это файл, в котором содержится *объектный код*, то есть машинный код в переносимом формате, который в большинстве случаев не может быть непосредственно выполнен. Главным преимуществом переносимого формата является то, что на этапе компоновки ему требуется минимум памяти.

Если при выполнении компиляции `go tool` использовать флаг командной строки `-pack`, то вместо объектного файла получим *архивный файл*:

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:40 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

Архивный файл — это двоичный файл, внутри которого содержится один или несколько других файлов. Как правило, архивные файлы применяются для объединения нескольких файлов в один. В Go используются архивные файлы формата `ar`.

¹ Если вы используете компьютер с Windows, то `go tool compile` может выполняться с ошибкой `can't find import`. В этом случае при вызове команды нужно добавить флаг `-I`, который будет указывать на место расположения библиотек в GOPATH, например:

```
$ go tool compile -I C:\golang\pkg\windows_amd64 unsafe.go
```

Для того чтобы просмотреть содержимое архивного файла `.a`, нужно воспользоваться следующей командой:

```
$ ar t unsafe.a
___.PKGDEF
_go_.o
```

Обратите внимание еще на один полезный флаг командной строки для команды `go tool compile`: `-race`. Он позволяет распознавать *состояние гонки*. Подробнее о том, что такое состояние гонки и почему его следует избегать, вы прочитаете в главе 10.

В конце этой главы вы найдете дополнительные примеры применения команды `go tool compile`, когда речь пойдет о языке ассемблера и деревьях узлов (node trees). Сейчас для проверки попробуйте выполнить следующую команду:

```
$ go tool compile -S unsafe.go
```

Эта команда генерирует много выходных данных, которые могут показаться непонятными. Это означает, что Go довольно хорошо скрывает излишние сложности, если только специально не попросить его показать их!

Сборка мусора

Сборка мусора (Garbage Collection, GC) — это процесс освобождения места в памяти, которое больше не используется. Другими словами, сборщик мусора определяет объекты, находящиеся вне области видимости, на которые нельзя больше ссылаться (недостижимые объекты), и освобождает занимаемую ими память. Этот процесс выполняется конкурентно, не до и не после, а во время работы Go-программы. Документация реализации сборщика мусора в Go гласит следующее:

«GC выполняется конкурентно (concurrent), одновременно с потоками мутатора (mutator), в точном соответствии с типом (этот принцип также известен как чувствительность к типу), допускается параллельное выполнение нескольких потоков GC. Это конкурентная пометка и очистка (mark-sweep), при которой используется барьер записи (write barrier). При этом процессе ничего не генерируется и не сжимается. Освобождение памяти выполняется на основе размера, выделенного для каждой программы P, чтобы в общем случае минимизировать фрагментацию и избежать блокировок».

В этом фрагменте документации много терминов, которые станут понятными позже. Но сначала разберемся, как просмотреть некоторые параметры сборки мусора.

К счастью, стандартная библиотека Go включает в себя функции, которые позволяют узнать, какая именно скрытая деятельность происходит внутри сборщика мусора. Необходимый для этого код хранится в файле `gColl.go` и состоит из трех частей.

Первая часть кода `gColl.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("-----")
}
```

Обратите внимание: каждый раз, когда мы хотим получить свежую статистику о сборке мусора, нам нужно заново вызывать функцию `runtime.ReadMemStats()`. Функция `printStats()` нужна для того, чтобы не писать многократно один и тот же код Go.

Вторая часть программы выглядит так:

```
func main() {
    var mem runtime.MemStats
    printStats(mem)

    for i := 0; i < 10; i++ {
        s := make([]byte, 50000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
}
```

Цикл `for` создает много больших *срезов* Go, чтобы под них выделялись большие объемы памяти и запускался сборщик мусора.

Последняя часть `gColl.go` содержит следующий код Go, который вызывает дополнительное выделение памяти для срезов Go:

```
    for i := 0; i < 10; i++ {
        s := make([]byte, 100000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
        time.Sleep(5 * time.Second)
    }
    printStats(mem)
}
```

Выполнение `gColl.go` на macOS Mojave дает следующий результат:

```
$ go run gColl.go
mem.Alloc: 66024
mem.TotalAlloc: 66024
mem.HeapAlloc: 66024
mem.NumGC: 0
-----
mem.Alloc: 50078496
mem.TotalAlloc: 500117056
mem.HeapAlloc: 50078496
mem.NumGC: 10
-----
mem.Alloc: 76712
mem.TotalAlloc: 1500199904
mem.HeapAlloc: 76712
mem.NumGC: 20
-----
```

Вряд ли вы будете постоянно проверять работу сборщика мусора Go, однако возможность наблюдать за тем, как он работает в медленном приложении, в долгосрочной перспективе способна экономить уйму времени. Поверьте, вы не пожалеете о том времени, которое потратите на изучение сборки мусора вообще и сборщика мусора Go в частности.

В следующей команде показан прием, позволяющий получить еще более подробный отчет о том, как работает сборщик мусора Go:

```
$ GODEBUG=gctrace=1 go run gColl.go
```

Если перед любой командой `go run` поставить `GODEBUG=gctrace=1`, то Go выводит аналитические данные о работе сборщика мусора. Данные представлены в такой форме:

```
gc 4 @0.025s 0%: 0.002+0.065+0.018 ms clock,
    0.021+0.040/0.057/0.003+0.14 ms cpu, 47->47->0 MB, 48 MB goal, 8 P
gc 17 @30.103s 0%: 0.004+0.080+0.019 ms clock,
    0.033+0/0.076/0.071+0.15 ms cpu, 95->95->0 MB, 96 MB goal, 8 P
```

Здесь приводится подробная информация о размерах кучи в процессе сборки мусора. Возьмем для примера тройку значений `47->47->0 MB`. Первое число — размер кучи перед запуском сборщика мусора; второе значение — размер кучи, когда сборщик завершает работу; последнее значение — актуальный размер кучи.

Трехцветный алгоритм

В основе работы сборщика мусора Go лежит трехцветный алгоритм.



Обратите внимание на то, что трехцветный алгоритм не уникален для Go; он может использоваться и в других языках программирования.

Официальное название алгоритма, используемого в Go, — *трехцветный алгоритм пометки и очистки*. Он может работать конкурентно, одновременно с программой, и использует *барьер записи*. Это означает, что при запуске Go-программы планировщик Go составляет график работы собственно приложения и сборщика мусора — как если бы планировщик Go имел дело с обычным приложением, состоящим из нескольких *горутин*! Подробнее о горутинах и планировщике Go вы узнаете из главы 9.

Основную идею трехцветного алгоритма сформулировали Эдсгер В. Дайкстра (Edsger W. Dijkstra), Лесли Лэмпорт (Leslie Lamport), Эй. Дж. Мартин (A. J. Martin), К. С. Шолтен (C. S. Scholten) и И. Ф. М. Стеффенс (E. F. M. Steffens). Этот алгоритм впервые описан в статье *On-the-Fly Garbage Collection: An Exercise in Cooperation*.

Главный принцип алгоритма трехцветной пометки и очистки состоит в разделении объектов, находящихся в куче, на три набора, в соответствии с цветом, который назначается им алгоритмом. Пора обсудить значение каждого из этих цветов. Объекты *черного цвета* гарантированно не имеют указателей ни на один объект *белого цвета*. Однако объект белого цвета может иметь указатель на объект черного цвета, поскольку это не влияет на работу сборщика мусора. Объекты *серого цвета* могут иметь указатели на некоторые объекты белого цвета. Именно объекты белого цвета являются претендентами на удаление.

Обратите внимание, что ни один объект не может перейти непосредственно из черного множества в белое — именно это обеспечивает работу алгоритма и позволяет освобождать память из-под объектов белого множества. Кроме того, ни один объект из черного множества не может напрямую указывать на объект из белого множества.

Итак, когда начинается сборка мусора, все объекты становятся белыми. Сборщик мусора перебирает все корневые объекты и окрашивает их в серый цвет. *Корневые объекты* — это объекты, к которым приложение может обращаться напрямую, включая глобальные переменные и другие элементы, находящиеся в стеке. Большинство этих объектов зависят от Go-кода конкретной программы.

После этого сборщик мусора выбирает серый объект, пометает его черным и проверяет, есть ли у него указатели на другие объекты из белого множества. Это означает, что при проверке серого объекта на предмет указателей на другие объекты он окрашивается в черный цвет. Если проверка обнаружит, что у данного объекта есть один или несколько указателей на белые объекты, алгоритм поменяет цвет этих белых объектов на серый. Процесс продолжается до тех пор, пока не будут перебраны все объекты серого множества. Затем объекты белого множества считаются недостижимыми, и занимаемая ими память может использоваться повторно. Таким образом, считается, что в этот момент элементы белого множества попали в корзину.



Обратите внимание: если в какой-то момент сборки мусора объект из серого множества станет недостижимым, он будет обработан не в этом, а в следующем цикле сборки мусора! Это не оптимальный, но и не такой уж плохой вариант.

Приложение, работающее во время выполнения сборки мусора, называется *мутатором*. Мутатор запускает небольшую функцию, называемую *барьером записи*. Эта функция выполняется всякий раз, когда изменяется указатель в куче. Если указатель объекта в куче изменился, это означает, что данный объект теперь достижим. Барьер записи пометает этот объект в серый цвет и помещает в серое множество.



Мутатор отвечает за то, чтобы ни один элемент из черного множества не имел указателя на элемент из белого множества. Это достигается с помощью функции барьера записи. Невыполнение этого условия разрушило бы процесс сборки мусора и, скорее всего, привело бы к аварийному завершению работы программы.

В итоге кучу можно представить как граф, состоящий из связанных объектов. Такой граф показан на рис. 2.1, где также продемонстрирован один из этапов сборки мусора.

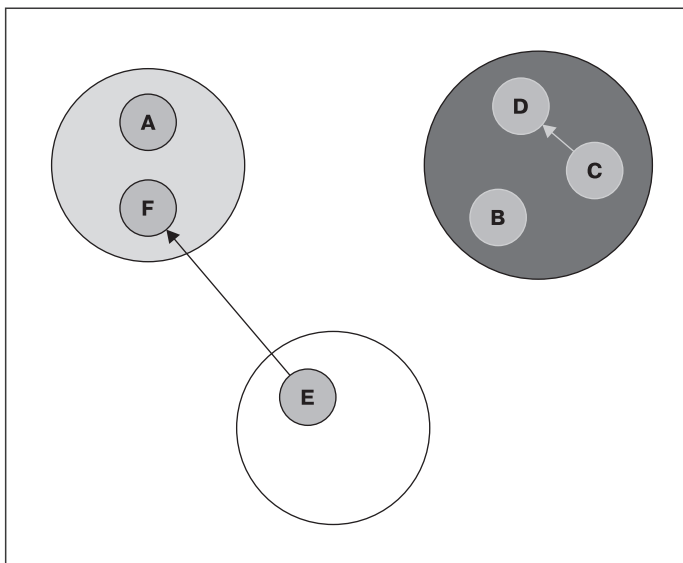


Рис. 2.1. В Go при сборке мусора в корзину область кучи программы можно представить в виде графа

Итак, у нас есть три цвета: черный, белый и серый. Когда алгоритм начинает работу, все объекты окрашены в белый цвет. По мере работы алгоритма белые объекты перемещаются в одно из двух остальных множеств: серое или черное.

Те объекты, которые останутся в белом множестве, в какой-то момент будут удалены.

На представленном графе видно, что, хотя объект E, который относится к белому множеству, может получить доступ к объекту F, он сам недостижим из какого-либо другого объекта, потому что никакой другой объект не указывает на объект E. Это делает объект E идеальным претендентом для сборки мусора! Кроме того, объекты A, B и C являются корневыми объектами и всегда достижимы; следовательно, они не могут быть собраны как мусор.

Угадаете ли вы, что будет дальше на этом графе? Несложно понять, что алгоритм должен будет обработать оставшиеся элементы из серого множества и, следовательно, объекты A и F перейдут в черное множество: объект A — потому что является корневым элементом, а F — потому что не указывает ни на какой другой объект, пока так же будет находиться в сером множестве.

После того как объект A будет удален сборщиком мусора, объект F станет недостижимым и попадет в мусор в следующем цикле сборщика мусора, поскольку недостижимый объект не может волшебным образом стать достижимым на следующей итерации цикла сборки мусора.

Сборщик мусора Go может также применяться к таким переменным, как *каналы*. Когда сборщик мусора обнаруживает, что канал недостижим, то есть недостижима переменная канала, сборщик мусора освобождает ресурсы этой переменной, даже если сам канал не закрыт. Подробнее о каналах вы узнаете в главе 9.

Go позволяет запускать сборку мусора вручную. Для этого нужно вставить в код Go оператор `runtime.GC()`. Однако помните, что `runtime.GC()` заблокирует вызывающую функцию и может даже заблокировать всю программу, особенно если это очень большая Go-программа с огромным количеством объектов. Так происходит главным образом потому, что невозможно выполнять сборку мусора, когда остальное окружение быстро меняется, поскольку тогда сборщик мусора не сможет четко идентифицировать элементы белого, черного и серого множеств. Такой статус сборки мусора также называется *начальной точкой сборки мусора*.

По адресу <https://github.com/golang/go/blob/master/src/runtime/mgc.go> вы найдете полный исходный код сборщика мусора Go, который стоит изучить, если вы хотите получить больше информации об операции сборки мусора. Вы даже можете внести изменения в этот код, если хватит смелости!



Обратите внимание: команда Go постоянно совершенствует сборщик мусора. Создатели Go пытаются ускорить сборку мусора, уменьшая количество сканирований, необходимых для трех цветовых множеств. Однако, несмотря на все оптимизации, общая идея алгоритма остается прежней.

Подробнее о работе сборщика мусора Go

В этом разделе вы более подробно узнаете о сборщике мусора Go и его деятельности. Основной проблемой сборщика мусора Go является низкая латентность, что означает, в сущности, короткие паузы, чтобы обеспечить его работу в реальном времени. При этом программа создает новые объекты и постоянно манипулирует существующими объектами посредством указателей. Такой процесс может завершиться созданием объектов, доступ к которым больше невозможен, потому что не существует указателей, ссылающихся на эти объекты. Затем эти объекты становятся мусором и ждут, когда сборщик мусора очистит их и освободит занимаемую ими память. После этого освободившаяся память снова готова к использованию.

Алгоритм пометки и очистки является самым простым из используемых алгоритмов. Этот алгоритм останавливает выполнение программы (*сборщик мусора stop-the-world*), чтобы перебрать все доступные объекты из кучи и **пометить** их. После этого алгоритм **удаляет** все недостижимые объекты. На этапе пометки каждый объект отмечается как белый, серый или черный. Дочерним объектам серых объектов назначается серый цвет, а их родительский серый объект теперь окрашивается в черный цвет. Когда все серые объекты будут перебраны, начинается этап очистки. Эта методика работает потому, что не существует указателей от черных объектов на белые, что является необходимым условием алгоритма.

Несмотря на свою простоту, алгоритм пометки и очистки приостанавливает выполнение работающей программы, следовательно, увеличивает задержку реальных процессов. Go пытается уменьшить эту задержку, запуская сборщик мусора как параллельный процесс и используя трехцветный алгоритм, описанный в предыдущем разделе. Однако другие процессы могут перемещать указатели или создавать новые объекты одновременно с работой сборщика мусора. Для сборщика мусора это может сильно усложнить жизнь. В результате трехцветный алгоритм сможет работать конкурентно при условии сохранения главного требования алгоритма отслеживания и очистки: ни один объект из черного множества не должен указывать на объект белого множества.

Решением этой проблемы является корректная обработка всех ситуаций, которые могут вызвать проблему для алгоритма сборки мусора. В частности, все новые объекты должны заноситься в серое множество, поскольку таким образом сохранится необходимое условие алгоритма пометки и очистки. В дополнение, при перемещении указателя объект, на который ссылается этот указатель, должен помечаться серым цветом. Серое множество играет роль барьера между белым и черным множествами. Наконец, при каждом перемещении указателя автоматически должен выполняться некий Go-код, которым является ранее упомянутый барьер записи, изменяющий цвет объекта. Задержка, вызванная выполнением кода барьера записи, — это та цена, которую приходится платить за возможность конкурентной сборки мусора.

Обратите внимание, что в языке программирования Java есть много сборщиков мусора, которые легко настраиваются с помощью ряда параметров. Один из них — *G1*. Его рекомендуется применять для приложений с низкой задержкой.



Важно помнить, что сборщик мусора Go работает в реальном времени, конкурентно с другими горутинами и оптимизируется только для обеспечения низкой латентности своего запуска.

В главе 11 вы узнаете, как отобразить на графике тайминг работы Go-программы, включая информацию о запусках сборщика мусора в Go.

Хеш-таблицы, срезы и сборщик мусора Go

В этом разделе на нескольких примерах я продемонстрирую, почему следует быть осторожным при сборке мусора. Цель этого раздела — показать, что способ хранения указателей сильно влияет на производительность сборщика мусора, особенно когда мы имеем дело с очень большим количеством указателей.

В представленных примерах используются указатели, срезы (slice) и хеш-таблицы (map), которые являются собственными типами данных Go. Подробнее об указателях, срезах и хеш-таблицах в Go вы узнаете из главы 3.

Использование срезов

Рассмотрим пример использования *среза* для хранения большого количества структур. Каждая структура состоит из двух целочисленных значений. Go-код `sliceGC.go` выглядит следующим образом:

```
package main

import (
    "runtime"
)

type data struct {
    i, j int
}

func main() {
    var N = 40000000
    var structure []data
    for i := 0; i < N; i++ {
        value := int(i)
        structure = append(structure, data{value, value})
    }

    runtime.GC()
    _ = structure[0]
}
```

Последний оператор (`_ = structure[0]`) используется для предотвращения преждевременной очистки сборщиком мусора переменной `structure`, поскольку на нее нет указателей и она не задействована за пределами цикла `for`. Тот же прием будет применяться в следующих трех Go-программах. Кроме этой важной детали, цикл `for` здесь необходим для помещения всех значений в структуры, которые хранятся в срезе.

Использование хеш-таблиц с указателями

В этом я подразделе покажу, как использовать хеш-таблицу для хранения всех указателей в виде целых чисел. Программа называется `mapStar.go` и содержит следующий Go-код:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]*int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = &value
    }
    runtime.GC()
    _ = myMap[0]
}
```

Хеш-таблица, в которой хранятся целочисленные указатели, носит имя `myMap`. Цикл `for` здесь используется для помещения целочисленных значений в хеш-таблицу.

Использование хеш-таблиц без указателей

Рассмотрим, как использовать хеш-таблицу, в которой хранятся простые значения без указателей. Go-код `mapNoStar.go` выглядит так:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]int)
```

```

    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = value
    }
    runtime.GC()
    _ = myMap[0]
}

```

Как и раньше, цикл `for` здесь используется для помещения в хеш-таблицу целочисленных значений.

Разделение хеш-таблицы

Программа, представленная в этом подразделе, выполняет преобразование одной хеш-таблицы в хеш-таблицу нескольких хеш-таблиц. Такой процесс называется *шардингом* (sharding). Программа этого подраздела хранится в файле `mapSplit.go`, и мы ее рассмотрим по частям. Первая часть `mapSplit.go` содержит следующий Go-код:

```

package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    split := make([]map[int]int, 200)

```

В этой части программы определяется хэш хешей.

Вторая часть выглядит так:

```

    for i := range split {
        split[i] = make(map[int]int)
    }
    for i := 0; i < N; i++ {
        value := int(i)
        split[i%200][value] = value
    }
    runtime.GC()
    _ = split[0][0]
}

```

На этот раз мы использовали два цикла `for`: один для создания хеша хешей, а второй — для хранения данных в этом хеше хешей.

Сравнение производительности описанных методик

Поскольку во всех четырех программах используются огромные структуры данных, они потребляют большие объемы памяти. Программы, которые занимают много

места в памяти, чаще запускают сборщик мусора Go. Сравним между собой производительность этих четырех реализаций с помощью команды `time(1)`.

В представленном результате важны не конкретные цифры, а разница во времени между четырьмя подходами. Результат сравнения выглядит так:

```
$ time go run sliceGC.go
real    1.50s
user    1.72s
sys     0.71s
$ time go run mapStar.go
real    13.62s
user    23.74s
sys     1.89s
$ time go run mapNoStar.go
real    11.41s
user    10.35s
sys     1.15s
$ time go run mapSplit.go
real    10.60s
user    10.01s
sys     0.74s
```

Оказывается, хеш-таблицы замедляют сборщик мусора Go, в то время как срезы работают с ним гораздо лучше. Следует отметить, что это не проблема хеш-таблиц, а результат работы сборщика мусора Go. Однако, если не использовать хеш-таблицы, в которых хранятся огромные объемы данных, в ваших программах эта проблема не будет столь очевидной.



Подробнее о тестировании в Go вы прочитаете в главе 11. Кроме того, в главе 3 вы познакомитесь с более профессиональным способом измерения времени, необходимого для выполнения команды или программы в Go.

Пока достаточно о сборщике мусора и его особенностях; тема следующего раздела — небезопасный код и стандартный Go-пакет `unsafe`.

Небезопасный код

Небезопасный код — это Go-код, который обходит безопасность типов и безопасность памяти Go. В большинстве случаев небезопасный код связан с указателями. Однако следует помнить, что использование небезопасного кода может быть опасным для ваших программ, поэтому, если вы не вполне уверены, что вам необходим небезопасный код в вашей программе, не используйте его!

Мы продемонстрируем использование небезопасного кода в программе `unsafe.go`, которую разделим на три части.

Первая часть `unsafe.go` выглядит так:

```
package main
```

```
import (
    "fmt"
    "unsafe"
)
```

Как видим, для использования небезопасного кода нужно импортировать стандартный пакет Go `unsafe`.

Вторая часть программы содержит следующий Go-код:

```
func main() {
    var value int64 = 5
    var p1 = &value
    var p2 = (*int32)(unsafe.Pointer(p1))
}
```

Обратите внимание на то, как здесь используется функция `unsafe.Pointer()`: она позволяет нам на свой страх и риск создать указатель `int32` с именем `p2`. Этот указатель ссылается на переменную `int64` с именем `value`, доступ к которой осуществляется с помощью указателя `p1`. Любой указатель Go можно преобразовать в `unsafe.Pointer`.



Указатель типа `unsafe.Pointer` позволяет переопределять систему типов Go. Это позволяет существенно повысить производительность, но может быть опасно, если использовать указатели неправильно или небрежно. Кроме того, так разработчики получают больший контроль над данными.

Последняя часть `unsafe.go` содержит следующий Go-код:

```
fmt.Println("p1: ", *p1)
fmt.Println("p2: ", *p2)
*p1 = 5434123412312431212
fmt.Println(value)
fmt.Println("p2: ", *p2)
*p1 = 54341234
fmt.Println(value)
fmt.Println("p2: ", *p2)
}
```



Вы можете разыменовывать указатель и получать, использовать или изменять его значение с помощью символа звездочки (*).

Если запустить `unsafe.go`, получим следующие результаты:

```
$ go run unsafe.go
*p1: 5
*p2: 5
5434123412312431212
*p2: -930866580
54341234
*p2: 54341234
```

О чем говорит нам этот вывод? О том, что указатель на 32-разрядное целое число не может предоставить доступ к 64-разрядному целому числу.

Как будет показано в следующем разделе, функции пакета `unsafe` позволяют делать с памятью еще много интересных вещей.

Пакет `unsafe`

Теперь, когда вы увидели, как работает пакет `unsafe`, самое время поговорить о том, почему этот пакет такой особенный. Если вы посмотрите на исходный код пакета `unsafe`, то несколько удивитесь. В системе macOS Mojave с версией Go 1.11.4, установленной с сайта Homebrew (<https://brew.sh/>), исходный код пакета `unsafe` расположен по адресу `/usr/local/Cellar/go/1.11.4/libexec/src/unsafe/unsafe.go`, и его содержимое без учета комментариев выглядит так:

```
$ cd /usr/local/Cellar/go/1.11.4/libexec/src/unsafe/
$ grep -v '^//' unsafe.go | grep -v '^$'
package unsafe
type ArbitraryType int
type Pointer *ArbitraryType
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

Где же находится остальной код Go пакета `unsafe`? Ответ на этот вопрос сравнительно прост: компилятор Go генерирует пакет `unsafe`, когда вы импортируете его в программу.



Многие низкоуровневые пакеты, такие как `runtime`, `syscall` и `os`, интенсивно используют пакет `unsafe`.

Еще один пример использования пакета `unsafe`

В этом подразделе вы ближе познакомитесь с пакетом `unsafe` и его возможностями на примере небольшой Go-программы с именем `moreUnsafe.go`. Мы разделим ее на три части. Программа `moreUnsafe.go` делает следующее: получает доступ ко всем элементам массива с помощью указателей.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
    "unsafe"
)
```

Вторая часть `moreUnsafe.go` содержит следующий Go-код:

```
func main() {
    array := [...]int{0, 1, -2, 3, 4}
    pointer := &array[0]
    fmt.Print(*pointer, " ")
    memoryAddress := uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])

    for i := 0; i < len(array)-1; i++ {
        pointer = (*int)(unsafe.Pointer(memoryAddress))
        fmt.Print(*pointer, " ")
        memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])
    }
}
```

Сначала переменная `pointer` указывает на адрес в памяти `array[0]` — первого элемента целочисленного массива. Затем переменная `pointer`, которая указывала на целочисленное значение, преобразуется в `unsafe.Pointer()`, потом — в `uintptr`. Результат сохраняется в `memoryAddress`.

Значение `unsafe.Sizeof(array[0])` переводит нас к следующему элементу массива, а именно — сколько памяти занимает каждый элемент массива. Поэтому данное значение прибавляется к переменной `memoryAddress` на каждой итерации цикла `for`, что позволяет получить адрес в памяти следующего элемента массива. Запись `*pointer` разыменовывает указатель и возвращает сохраненное целочисленное значение.

Третья часть программы выглядит так:

```
    fmt.Println()
    pointer = (*int)(unsafe.Pointer(memoryAddress))
    fmt.Print("One more: ", *pointer, " ")
    memoryAddress = uintptr(unsafe.Pointer(pointer)) + unsafe.Sizeof(array[0])
    fmt.Println()
}
```

В последней части программы мы пытаемся получить доступ к несуществующему элементу массива, используя указатели и адреса ячеек памяти. Вследствие использования пакета `unsafe` компилятор Go не может отследить эту логическую ошибку, в результате которой будет возвращено нечто некорректное.

При выполнении `moreUnsafe.go` генерируются следующие результаты:

```
$ go run moreUnsafe.go
0 1 -2 3 4
One more: 824634208008
```

Мы только что получили доступ ко всем элементам массива Go с помощью указателей. Однако настоящая проблема заключается в том, что при попытке доступа к недопустимому элементу массива программа, не генерируя ошибку, возвращает случайное число.

Вызов C-кода из Go

Цель Go — сделать программирование приятным, избавив вас от особенностей C. Однако C остается языком программирования с очень большими возможностями и все еще полезен. Это означает, что существуют ситуации, например, при использовании базы данных или драйверов устройств, написанных на C, в которых все еще требуется использование C. Это говорит о том, что вам придется работать с C-кодом в Go-проектах.



Если вы обнаружите, что слишком часто используете эту возможность в одном и том же проекте, вам, вероятно, придется пересмотреть выбранную концепцию или выбор языка программирования.

Вызов C-кода из Go в одном файле

Самый простой способ вызвать C-код из Go-программы — включить код на C в исходный Go-файл. Тут нужен особый подход, но это довольно быстро и не очень сложно.

Рассмотрим исходный Go-файл с именем `cgo.go`. В этом файле содержится код на C и на Go, который мы разделим на три части.

Первая часть выглядит так:

```
package main

// #include <stdio.h>
// void callC() {
//     printf("Calling C code!\n");
// }
import "C"
```



Как видим, C-код включен в Go-программу в виде комментариев. Однако благодаря импорту пакета `C` `go tool` знает, что делать с подобными комментариями.

Вторая часть программы содержит следующий Go-код:

```
import "fmt"

func main() {
```

Таким образом, все остальные пакеты должны импортироваться отдельно. В последней части `cGo.go` содержится следующий код:

```
fmt.Println("A Go statement!")
C.callC()
fmt.Println("Another Go statement!")
}
```

Для того чтобы выполнить С-функцию `callC()`, ее нужно вызвать как `C.callC()`. Выполнение `cGo.go` приведет к следующим результатам:

```
$ go run cGo.go
A Go statement!
Calling C code!
Another Go statement!
```

Вызов из Go С-кода в отдельных файлах

Теперь попробуем вызывать С-код из Go-программы, если этот С-код находится в отдельном файле.

Рассмотрим следующую задачу, которую мы решим с помощью нашей программы. Будем использовать две С-функции, которые мы якобы реализовали ранее и теперь не хотим или не можем переписать на Go.

С-код

В этом подразделе представлен С-код нашего примера. Он содержится в двух файлах: `callC.h` и `callC.c`. Подключаемый файл (`callC.h`) хранит следующий код:

```
#ifndef CALLC_H
#define CALLC_H

void cHello();
void printMessage(char* message);

#endif
```

Исходный С-файл (`callC.c`) содержит следующий С-код:

```
#include <stdio.h>
#include "callC.h"

void cHello() {
    printf("Hello from C!\n");
}

void printMessage(char* message) {
    printf("Go send me %s\n", message);
}
```

Файлы `callC.c` и `callC.h` хранятся в отдельном каталоге, который в данном случае называется `callClib`. Но вы можете выбрать любое имя каталога по своему желанию.



В С-коде нет ничего, что бы указывало на то, что этот код будет вызываться из Go-программы. Главное — вызывать правильные С-функции с указанием верного количества параметров правильного типа. Посмотрите на Go-код — и тогда все станет ясно.

Go-код

В этом подразделе представлен исходный Go-код для нашего примера. Код находится в файле `callC.go`, и мы его рассмотрим по частям.

Первая часть `callC.go` содержит следующий Go-код:

```
package main

// #cgo CFLAGS: -I${SRCDIR}/callClib
// #cgo LDFLAGS: ${SRCDIR}/callC.a
// #include <stdlib.h>
// #include <callC.h>
import "C"
```

Самый важный оператор Go всего исходного Go-файла — это отдельный оператор `import`, используемый для подключения пакета `C`. Но `C` — это виртуальный Go-пакет, который просто дает инструкцию `go build` сначала обработать входной файл с помощью инструмента `cgo`, и только потом передать файл компилятору Go. Как видим, здесь тоже использованы комментарии, которые сообщают Go-программе о наличии в ней С-кода. В данном случае они указывают программе `callC.go`, где найти файл `callC.h`, а также файл библиотеки `callC.a`, который мы вскоре создадим. Такие строки начинаются с `#cgo`.

Вторая часть программы выглядит так:

```
import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println("Going to call a C function!")
    C.hello()
```

Последняя часть `callC.go` выглядит следующим образом:

```
    fmt.Println("Going to call another C function!")
    myMessage := C.CString("This is Mihalis!")
    defer C.free(unsafe.Pointer(myMessage))
```

```

    C.printMessage(myMessage)

    fmt.Println("All perfectly done!")
}

```

Чтобы передать строку из Go в С-функцию, нужно создать строку в формате С с помощью `C.CString()`. Кроме того, нам понадобится оператор `defer`, чтобы освободить память, занимаемую строкой С, после того как она будет использована. Оператор `defer` включает в себя вызов функции `C.free()` и затем `unsafe.Pointer()`.

В следующем разделе вы узнаете, как скомпилировать и выполнить программу `callC.go`.

Сочетание кода на Go и С

Теперь, когда у нас есть код на С и на Go, пора узнать, что нужно сделать, чтобы выполнить файл на Go, из которого вызывается код на С.

Хорошая новость: нам не придется делать ничего сверхсложного, потому что вся важная информация содержится в Go-файле. Единственное — нужно скомпилировать С-код, чтобы создать библиотеку, для чего следует выполнить такие команды:

```

$ ls -l callClib/
total 16
-rw-r--r--@ 1 mtsouk  staff  162 Jan 10 09:17 callC.c
-rw-r--r--@ 1 mtsouk  staff   89 Jan 10 09:17 callC.h
$ gcc -c callClib/*.c
$ ls -l callC.o
-rw-r--r-- 1 mtsouk  staff  952 Jan 22 22:03 callC.o
$ file callC.o
callC.o: Mach-O 64-bit object x86_64
$ /usr/bin/ar rs callC.a *.o
ar: creating archive callC.a
$ ls -l callC.a
-rw-r--r-- 1 mtsouk  staff 4024 Jan 22 22:03 callC.a
$ file callC.a
callC.a: current ar archive
$ rm callC.o

```

У нас появится файл с именем `callC.a`, расположенный в том же каталоге, что и `callC.go`. Исполняемый файл `gcc` — это имя *С-компилятора*.

Теперь мы готовы скомпилировать файл с Go-кодом и создать новый исполняемый файл:

```

$ go build callC.go
$ ls -l callC
-rwxr-xr-x 1 mtsouk  staff 2403184 Jan 22 22:10 callC
$ file callC
callC: Mach-O 64-bit executable x86_64

```

Запустив исполняемый файл `callC`, получим следующий результат:

```
$ ./callC
Going to call a C function!
Hello from C!
Going to call another C function!
Go send me This is Mihalis!
All perfectly done!
```



Если вам нужно использовать небольшое количество С-кода, то весьма желательно использовать общий Go-файл для кода на С и на Go — так проще. Но если вы собираетесь сделать что-то большое и сложное, лучше создать статическую С-библиотеку.

Вызов Go-функций из С-кода

Аналогичным образом можно вызвать Go-функции из С-кода. В этом разделе представлен небольшой пример, в котором две Go-функции вызываются из С-программы. Для этого Go-пакет преобразуем в библиотеку С (shared library), которая будет использоваться в С-программе.

Go-пакет

Представим код Go-пакета, который будет использоваться в С-программе. Именем Go-пакета должно быть `main`, но имя файла может быть любым; в данном случае это `вс.go`. Мы рассмотрим код пакета по частям.



Подробнее о Go-пакетах вы прочитаете в главе 6.

Первая часть кода Go-пакета выглядит так:

```
package main

import "C"

import (
    "fmt"
)
```

Как вам известно, Go-пакет обязательно должен называться `main`. Нам также нужно импортировать в Go-код пакет `C`.

Вторая часть пакета содержит следующий Go-код:

```
// export PrintMessage
func PrintMessage() {
    fmt.Println("A Go function!")
}
```

Каждую функцию Go, которая будет вызываться из C-кода, необходимо сначала экспортировать. Это означает, что перед ее реализацией нужно поставить строку комментария, начинающуюся с `//export`. После `//export` следует указать имя функции, которую будет использовать C-код.

Последняя часть файла `usedByC.go` выглядит так:

```
// export Multiply
func Multiply(a, b int) int {
    return a * b
}

func main() {
}
```

Функция `main()` файла `usedByC.go` не нуждается в коде, поскольку она не будет экспортироваться и, следовательно, не будет использоваться C-программой. Кроме того, поскольку мы хотим экспортировать функцию `Multiply()`, необходимо поставить перед реализацией этой функции строку `//export Multiply`.

Теперь нужно сгенерировать из Go-кода общую библиотеку C, выполнив следующую команду:

```
$ go build -o usedByC.o -buildmode=c-shared usedByC.go
```

Эта команда создаст два файла с именами `usedByC.h` и `usedByC.o`:

```
$ ls -l usedByC.*
-rw-r--r--@ 1 mtsouk  staff      204    Jan 10 09:17 usedByC.go
-rw-r--r--  1 mtsouk  staff     1365    Jan 22 22:14 usedByC.h
-rw-r--r--  1 mtsouk  staff    2329472    Jan 22 22:14 usedByC.o
$ file usedByC.o
usedByC.o: Mach-O 64-bit dynamically linked shared library x86_64
```

После этого не следует вносить никаких изменений в `usedByC.h`.

C-код

Соответствующий C-код содержится в исходном файле `willUseGo.c`, который мы рассмотрим по частям. Первая часть `willUseGo.c` выглядит так:

```
#include <stdio.h>
#include "usedByC.h"

int main(int argc, char **argv) {
    GoInt x = 12;
    GoInt y = 23;

    printf("About to call a Go function!\n");
    PrintMessage();
}
```


Если вам знаком язык C, то вы поймете, зачем нужно подключить файл `usedByC.h`: именно благодаря этому файлу C-код узнает о доступных функциях библиотеки.

Вторая часть C-программы выглядит так:

```
GoInt p = Multiply(x,y);
printf("Product: %d\n", (int)p);
printf("It worked!\n");
return 0;
}
```

Переменная `GoInt p` необходима для получения целочисленного значения из функции `Go`; затем посредством операции `(int)` `p` это значение преобразуется в целое число формата C.

Компиляция и выполнение `willUseGo.c` на машине с macOS Mojave приведет к следующим результатам:

```
$ gcc -o willUseGo willUseGo.c ./usedByC.o
$ ./willUseGo
About to call a Go function!
A Go function!
Product: 276
It worked!
```

Ключевое слово `defer`

Ключевое слово `defer` позволяет отложить выполнение функции до тех пор, пока не потребуется вернуть значение внешней функции. Этот прием широко используется в операциях ввода и вывода файлов, поскольку избавляет от необходимости запоминать, когда следует закрыть открытый файл. Ключевое слово `defer` позволяет поместить вызов функции, которая закрывает открытый файл, рядом с вызовом функции, которая его открыла. Подробнее об использовании `defer` в операциях, связанных с файлами, вы узнаете из главы 8. В этом же разделе будут представлены два варианта использования `defer`. Как практически использовать `defer`, вы увидите в разделе о встроенных Go-функциях `panic()` и `recover()`, а также в разделе о журналировании.

Очень важно учитывать, что после возвращения из внешней функции *отложенные функции* выполняются в порядке «последним пришел — первым вышел» (Last In, First Out, LIFO). Это означает, что если сначала отложить функцию `f1()`, затем функцию `f2()` и после нее — функцию `f3()` в одной и той же внешней функции, то, когда эта внешняя функция будет возвращать значение, функция `f3()` выполнится первой, функция `f2()` — второй, а функция `f1()` — последней.

Поскольку такое определение `defer` не очень понятно, возможно, изучив Go-код и результаты программы `defer.go`, которые мы рассмотрим по частям, вы лучше разберетесь в применении `defer`.

Первая часть программы `defer.go` выглядит так:

```
package main

import (
    "fmt"
)

func d1() {
    for i := 3; i > 0; i-- {
        defer fmt.Print(i, " ")
    }
}
```

Помимо блока `import`, в этом коде Go реализована функция с именем `d1()`. Эта функция содержит цикл `for` с оператором `defer`, который будет выполнен три раза.

Вторая часть `defer.go` состоит из следующего Go-кода:

```
func d2() {
    for i := 3; i > 0; i-- {
        defer func() {
            fmt.Print(i, " ")
        }()
    }
    fmt.Println()
}
```

В этой части кода мы видим реализацию второй функции с именем `d2()`. Функция `d2()` содержит цикл `for` с оператором `defer`, который также будет выполнен три раза. Однако здесь ключевое слово `defer` применяется не к отдельному выражению `fmt.Print()`, а к *анонимной функции*. Кроме того, эта анонимная функция не принимает параметров.

Последняя часть программы содержит следующий Go-код:

```
func d3() {
    for i := 3; i > 0; i-- {
        defer func(n int) {
            fmt.Print(n, " ")
        }(i)
    }
}

func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

Кроме функции `main()`, которая вызывает функции `d1()`, `d2()` и `d3()`, мы видим здесь реализацию функции `d3()`. Данная функция включает в себя цикл `for`, в котором ключевое слово `defer` используется для анонимной функции. Однако на этот раз у анонимной функции есть целочисленный параметр с именем `n`. Из Go-кода видно, что параметр `n` принимает значение от переменной `i`, используемой в цикле `for`.

Выполнение `defer.go` приводит к следующим результатам:

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

Скорее всего, вы посчитаете полученные результаты сложными для понимания, что доказывает: работа и результат применения `defer` могут быть весьма странными, если не сделать сам код понятным и недвусмысленным. Позвольте мне разъяснить эти результаты, чтобы вы имели представление о том, сколько путаницы может внести `defer`, если не уделить коду достаточно внимания.

Начнем с первой строки (1 2 3), сгенерированной функцией `d1()`. Значения `i` в `d1()` равны 3, 2 и 1 — именно в таком порядке. Отложенной функцией в `d1()` является `fmt.Print()`; в результате, когда функция `d1()` готова завершить работу, мы получаем три значения переменной `i` из цикла `for` в обратном порядке, поскольку отложенные функции выполняются в порядке LIFO.

Теперь о второй строке результатов, которая создается функцией `d2()`. Странно, что вместо 1 2 3 мы получили на выходе три нуля; однако причина этого относительно проста.

После того как завершился цикл `for`, значение `i` равно 0, поскольку именно это значение `i` привело к завершению цикла `for`. Но весь фокус в том, что отложенная анонимная функция выполняется после завершения цикла `for`. А поскольку у нее нет параметров, то она трижды выполняется со значением `i`, равным 0, и эти результаты выводятся на экран. Такой сбивающий с толку код приводит к появлению в проектах неприятных ошибок, поэтому постарайтесь избежать подобных вещей.

Наконец, рассмотрим третью строку результатов, которую генерирует функция `d3()`. Благодаря тому что анонимная функция имеет параметр, всякий раз, когда эта анонимная функция откладывается, она получает и использует текущее значение `i`. В результате при каждом запуске анонимная функция получает разные значения для обработки, что видно по сгенерированным результатам.

Сейчас вам должно быть ясно, что наилучший вариант использования `defer` — третий, который представлен в функции `d3()`, потому что здесь мы намеренно передаем в анонимную функцию нужную переменную простым и понятным способом.

Использование defer для журналирования

Рассмотрим способ применения `defer`, связанный с журналированием. Цель этого метода — упростить журналирование информации о выполнении функции, сделать ее более заметной. Go-программа, в которой показано использование ключевого слова `defer` для журналирования, называется `logDefer.go`. Исследуя ее, мы разделим эту программу на три части.

Первая часть `logDefer.go` выглядит так:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

func one(aLog *log.Logger) {
    aLog.Println("-- FUNCTION one -----")
    defer aLog.Println("-- FUNCTION one -----")

    for i := 0; i < 10; i++ {
        aLog.Println(i)
    }
}
```

В функции `one()` ключевое слово `defer` используется для того, чтобы гарантировать, что второй вызов `aLog.Println()` будет выполнен непосредственно перед завершением работы функции `one()`. Поэтому все журнальные сообщения от функции `one()` будут размещаться между открывающим и закрывающим вызовами `aLog.Println()`. Так намного легче обнаружить журнальные сообщения данной функции в журнальных файлах.

Вторая часть `logDefer.go` выглядит так:

```
func two(aLog *log.Logger) {
    aLog.Println("---- FUNCTION two")
    defer aLog.Println("FUNCTION two -----")

    for i := 10; i > 0; i-- {
        aLog.Println(i)
    }
}
```

Для удобства группировки журнальных сообщений функции `two()` также применяется функция `defer`. Однако в функции `two()` используются сообщения,

немного отличающиеся от тех, что выводятся функцией `one()`. Пользователь сам выбирает формат сообщений.

Последняя часть `logDefer.go` содержит следующий Go-код:

```
func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

    iLog := log.New(f, "logDefer ", log.LstdFlags)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")

    one(iLog)
    two(iLog)
}
```

При выполнении `logDefer.go` ничего не будет выводиться на экран. Однако просмотр содержимого `/tmp/mGo.log` — журнального файла, который используется программой, — покажет, насколько удобно в данном случае применять `defer`:

```
$ cat /tmp/mGo.log
logDefer 2019/01/19 21:15:11 Hello there!
logDefer 2019/01/19 21:15:11 Another log entry!
logDefer 2019/01/19 21:15:11 -- FUNCTION one -----
logDefer 2019/01/19 21:15:11 0
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 -- FUNCTION one -----
logDefer 2019/01/19 21:15:11 ---- FUNCTION two
logDefer 2019/01/19 21:15:11 10
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 FUNCTION two -----
```

Функции `panic()` и `recover()`

В этом разделе вы познакомитесь с хитроумной техникой, которая уже упоминалась в предыдущей главе. Эта техника, предполагающая использование функций `panic()` и `recover()`, представлена в файле `panicRecover.go`, который мы рассмотрим по частям.

Строго говоря, `panic()` — это встроенная функция Go, которая завершает текущий поток Go-программы и начинает бить тревогу. Что же касается функции `recover()`, которая также является встроенной функцией Go, то она позволяет вернуть контроль над программой, которая только что запаниковала, запустив `panic()`.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
)

func a() {
    fmt.Println("Inside a()")
    defer func() {
        if c := recover(); c != nil {
            fmt.Println("Recover inside a()!")
        }
    }()
    fmt.Println("About to call b()")
    b()
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
}
```

Помимо блока `import`, эта часть содержит реализацию функции `a()`. Самой важной частью функции `a()` является блок кода `defer`. В нем реализована анонимная функция, которая будет вызываться при вызове `panic()`.

Второй фрагмент кода `panicRecover.go` выглядит так:

```
func b() {
    fmt.Println("Inside b()")
    panic("Panic in b()!")
    fmt.Println("Exiting b()")
}
```

Последняя часть программы, которая иллюстрирует функции `panic()` и `recover()`, выглядит следующим образом:

```
func main() {
    a()
    fmt.Println("main() ended!")
}
```

Выполнение `panicRecover.go` приводит к следующим результатам:

```
$ go run panicRecover.go
Inside a()
About to call b()
Inside b()
Recover inside a()!
main() ended!
```

То, что сейчас произошло, поистине впечатляет. Однако, как видно из вывода программы, функция `a()` не завершилась нормально, потому что два ее последних оператора не были выполнены:

```
fmt.Println("b() exited!")
fmt.Println("Exiting a()")
```

Все же, к счастью, `panicRecover.go` завершилась по нашей воле и без паники благодаря тому, что анонимная функция, используемая в `defer`, взяла ситуацию под контроль. Обратите внимание: функция `b()` ничего не знает о функции `a()`, но функция `a()` содержит Go-код, который обрабатывает состояние паники, возникающее в функции `b()`.

Самостоятельное использование функции `panic()`

Мы также можем использовать функцию `panic()` самостоятельно, без каких-либо попыток восстановления. В этом подразделе покажем, как это сделать, используя Go-код из файла `justPanic.go`, который мы разделим на две части.

Первая часть `justPanic.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
)
```

Как видим, использование `panic()` не требует импорта дополнительных Go-пакетов.

Вторая часть `justPanic.go` содержит следующий Go-код:

```
func main() {
    if len(os.Args) == 1 {
        panic("Not enough arguments!")
    }

    fmt.Println("Thanks for the argument(s)!")
}
```

Если Go-программа не имеет хотя бы одного аргумента командной строки, то она вызовет функцию `panic()`. Функция `panic()` принимает один параметр — сообщение об ошибке, которое мы хотим вывести на экран.

Выполнение `justPanic.go` на машине с macOS Mojave приводит к следующим результатам:

```
$ go run justPanic.go
panic: Not enough arguments!
goroutine 1 [running]:
main.main()
/Users/mtsouk/ch2/code/justPanic.go:10 +0x91
exit status 2
```

Как видим, применение функции `panic()` само по себе приводит к завершению работы Go-программы, не давая возможности восстановить ее. Таким образом, использование пары `panic()` и `recover()` гораздо более практично и профессионально, чем использование одной только функции `panic()`.



Вывод функции `panic()` похож на вывод функции `Panic()` из пакета `log`. Однако `panic()` ничего не передает в службу журналирования UNIX-компьютера.

Две полезные UNIX-утилиты

Бывают случаи, когда UNIX-программа по неизвестной причине дает сбой или не работает должным образом. Вы хотите выяснить, почему так происходит, но при этом не хотите переписывать код и добавлять кучу операторов отладки.

В этом разделе представлены две утилиты командной строки, которые позволяют увидеть системные вызовы C, генерируемые исполняемым файлом. Эти два инструмента называются `strace(1)` и `dtrace(1)` и позволяют проверять работу программы.



Имейте в виду, что по большому счету любая программа, работающая на компьютере с UNIX, для связи с ядром UNIX и выполнения большинства задач в итоге использует системные вызовы C.

Обе утилиты работают с командой `go run`, но вы получите меньше несвязанных выходных данных, если сначала создадите исполняемый файл, используя `go build`, а затем используете этот файл. В основном так происходит потому, что, как вы уже знаете, `go run` создает несколько временных файлов, прежде чем фактически запустить Go-код. Обе утилиты увидят это и попытаются отобразить информацию о временных файлах, а это не то, что нам нужно.

Утилита strace

Утилита командной строки `strace(1)` позволяет отслеживать системные вызовы и сигналы. Поскольку `strace(1)` работает только на компьютерах с Linux, в этом разделе для демонстрации `strace(1)` будет использоваться компьютер с Debian Linux.

Утилита `strace(1)` выводит на экран следующую информацию:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 15 vars */]) = 0
brk(0)                                = 0x186c000
fstat(3, {st_mode=S_IFREG|0644, st_size=35288, ...}) = 0
```

В выходных данных `strace(1)` отображаются все системные вызовы вместе с их параметрами, а также возвращаемыми значениями. Обратите внимание: в мире UNIX, если функция возвращает 0 — это хорошо.

Для того чтобы обработать исполняемый файл, следует поместить команду `strace(1)` перед файлом, который вы хотите обработать. Однако, чтобы сделать полезные выводы из результатов, вам придется их самостоятельно интерпретировать. К счастью, такие утилиты, как `grep(1)`, позволяют получить результат, который нам действительно нужен:

```
$ strace find /usr 2>&1 | grep ioctl
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59c50) = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59be0) = -1 ENOTTY (Inappropriate ioctl for device)
```

При использовании параметра командной строки `-c` утилита `strace(1)` позволяет выводить счетчик времени, вызовы и ошибки для каждого системного вызова:

```
$ strace -c find /usr 1>/dev/null
```

% time	seconds	usecs/call	calls	errors	syscall
82.88	0.063223	2	39228		getdents
16.60	0.012664	1	19587		newfstatat
0.16	0.000119	0	19618	13	open

Поскольку обычно результаты программы попадают в стандартный поток вывода, а результаты `strace(1)` — в стандартный поток ошибок, выполненная выше команда отменяет вывод результатов проверяемой команды `find` и показывает результаты команды `strace(1)`. Как видно из последней строки выходных данных, системный вызов `open(2)` сделан 19 618 раз, при этом возникло 13 ошибок и все это заняло примерно 0,16 % времени выполнения всей команды, или 0,000119 секунды.

Утилита dtrace

Утилиты отладки, такие как `strace(1)` и `truss(1)`, позволяют отслеживать системные вызовы, производимые процессом, однако иногда они бывают медленными и поэтому не подходят для решения проблем производительности в высоконагруженных UNIX-системах. Еще одна утилита, называемая *DTrace*, позволяет увидеть, что происходит внутри всей системы, без необходимости что-либо изменять или перекомпилировать. DTrace также позволяет работать в системах, находящихся в процессе эксплуатации, и динамически наблюдать за работающими программами и серверными процессами, не создавая больших дополнительных нагрузок.



Несмотря на наличие версии `dtrace(1)` для Linux, утилита `dtrace(1)` лучше всего работает в macOS и других вариантах FreeBSD.

В этом подразделе используется утилита командной строки `dtruss(1)`, поставляемая с macOS. Это просто сценарий для `dtrace(1)`, который показывает системные вызовы процесса и избавляет нас от необходимости самостоятельно писать скрипт для `dtrace(1)`. Обратите внимание, что для выполнения `dtrace(1)` и `dtruss(1)` необходимы права пользователя `root`.

Утилита `dtruss(1)` выводит на экран следующие данные:

```
$ sudo dtruss godoc
ioctl(0x3, 0x80086804, 0x7FFEEFBFEC20)      = 0 0
close(0x3)                                = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0)    = -1 Err#2
thread_selfid(0x0, 0x0, 0x0)              = 1895378 0
geteuid(0x0, 0x0, 0x0)                    = 0 0
getegid(0x0, 0x0, 0x0)                    = 0 0
```

Как видим, `dtruss(1)` работает так же, как утилита `strace(1)`. Подобно `strace(1)`, `dtruss(1)` при использовании с параметром `-c` выводит счетчик системных вызовов:

```
$ sudo dtruss -c go run unsafe.go 2>&1
```

CALL	COUNT
access	1
bsdthread_register	1
getuid	1
ioctl	1
issetugid	1
kqueue	1
write	1
mkdir	2
read	244
kevent	474

<code>fcntl</code>	479
<code>lstat64</code>	553
<code>psynch_cvsignal</code>	649
<code>psynch_cvwait</code>	654

Эти данные сразу сообщат вам о возможных узких местах вашего Go-кода или позволят сравнить производительность двух утилит командной строки.



К утилитам, подобным `strace(1)`, `dtrace(1)` и `dtruss(1)`, нужно привыкнуть. Однако такие инструменты способны сделать нашу жизнь намного проще и комфортнее, поэтому я настоятельно рекомендую вам прямо сейчас начать изучение хотя бы одного такого инструмента.

Если хотите больше узнать об утилите `dtrace(1)`, обратитесь к публикации *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* Брендана Грегга (Brendan Gregg) и Джима Мауро (Jim Mauro) или посетите сайт <http://dtrace.org/>.

Учтите, что `dtrace(1)` намного функциональнее, чем `strace(1)`, поскольку у `dtrace(1)` есть собственный язык программирования. Однако `strace(1)` более универсальна, если вам нужно только проследить за системными вызовами исполняемого файла.

Среда Go

В этом разделе речь пойдет о том, как узнать информацию о вашей текущей среде Go с помощью функций и свойств пакета `runtime`. Программа, которая будет разработана в этом разделе, называется `goEnv.go`. Рассмотрим ее по частям.

Первая часть `goEnv.go` выглядит так:

```
package main

import (
    "fmt"
    "runtime"
)
```

Как вы вскоре увидите, в пакет `runtime` входят функции и свойства, которые предоставят нужную информацию. Во второй части кода `goEnv.go` содержится реализация функции `main()`:

```
func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("Using Go version", runtime.Version())
    fmt.Println("Number of CPUs:", runtime.NumCPU())
    fmt.Println("Number of Goroutines:", runtime.NumGoroutine())
}
```

Выполнение `goEnv.go` на компьютере с macOS Mojave с версией Go 1.11.4 приводит к следующим результатам:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.11.4
Number of CPUs: 8
Number of Goroutines: 1
```

На компьютере с Debian Linux и версией Go 1.3.3 эта же программа генерирует следующие выходные данные:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.3.3
Number of CPUs: 1
Number of Goroutines: 4
```

Однако, чтобы получить по-настоящему полезную информацию о среде Go, мы воспользуемся следующей программой с именем `requiredVersion.go`. Эта программа сообщает, какую версию Go вы используете — 1.8 или выше:

```
package main

import (
    "fmt"
    "runtime"
    "strconv"
    "strings"
)

func main() {
    myVersion := runtime.Version()
    major := strings.Split(myVersion, ".")[0][2]
    minor := strings.Split(myVersion, ".")[1]
    m1, _ := strconv.Atoi(string(major))
    m2, _ := strconv.Atoi(minor)

    if m1 == 1 && m2 < 8 {
        fmt.Println("Need Go version 1.8 or higher!")
        return
    }

    fmt.Println("You are using Go version 1.8 or higher!")
}
```

Стандартный Go-пакет `strings` здесь используется для разбиения строки с версией Go, которую возвращает функция `runtime.Version()`. Из этой строки мы извлекаем две первые части, а функция `strconv.Atoi()` используется для преобразования строки в целое число.

Выполнение `requiredVersion.go` на компьютере с macOS Mojave дает следующий результат:

```
$ go run requiredVersion.go
You are using Go version 1.8 or higher!
```

Если же запустить `requiredVersion.go` на компьютере с Debian Linux, который мы уже использовали ранее в этом разделе, то программа выдаст следующий результат:

```
$ go run requiredVersion.go
Need Go version 1.8 or higher!
```

Таким образом, используя Go-код из файла `requiredVersion.go`, мы можем определить, установлена ли на вашем UNIX-компьютере требуемая версия Go.

Команда `go env`

Если вам нужно получить список всех переменных среды, поддерживаемых языком и компилятором Go, а также текущие значения этих переменных, то следует выполнить команду `go env`.

На моем компьютере с macOS Mojave с установленной версией Go 1.11.4 `go env` выдает весьма обширные результаты:

```
$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/mtsouk/Library/Caches/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/mtsouk/go"
GOPROXY=""
GORACE=""
GOROOT="/usr/local/Cellar/go/1.11.4/libexec"
GOTMPDIR=""
GOTOOLDIR="/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
CXX="clang++"
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
```

```
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-
arguments -fmessage-length=0 -fdebug-prefix-
map=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build790367620=/tmp/go-build -gno-record-gcc-switches -fno-common"
```

Обратите внимание, что некоторые из этих переменных среды могут иметь другие значения, если вы используете другую версию Go, или ваше имя пользователя не `mtsouk`, или вы используете другой вариант UNIX на ином оборудовании, или вы используете Go-модули по умолчанию (`GOMOD`).

Go-ассемблер

В этом разделе вкратце изложена информация о языке ассемблера и Go-ассемблере, который является инструментом Go, позволяющим увидеть ассемблерный код, используемый компилятором Go.

В качестве примера рассмотрим код ассемблера для программы `goEnv.go`, описанной в предыдущем разделе этой главы. Для этого мы выполним следующую команду:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S goEnv.go
```

Значение переменной `GOOS` определяет имя операционной системы, для которой создается программа, а значение переменной `GOARCH` — архитектуру компиляции. Эта команда выполнена на компьютере с macOS Mojave, поэтому переменной `GOOS` было присвоено значение `darwin`.

Даже для такой небольшой программы, как `goEnv.go`, эта команда выводит довольно много данных. Часть ее результатов выглядит так:

```
"".main TEXT size=859 args=0x0 locals=0x118
  0x0000 00000 (goEnv.go:8)      TEXT    "".main(SB), $280-0
  0x00be 00190 (goEnv.go:9)      PCDATA  $0, $1
  0x0308 00776 (goEnv.go:13)    PCDATA  $0, $5
  0x0308 00776 (goEnv.go:13)    CALL    runtime.convT2E64(SB)
"".init TEXT size=96 args=0x0 locals=0x8
  0x0000 00000 (<autogenerated>:1) TEXT    "".init(SB), $8-0
  0x0000 00000 (<autogenerated>:1) MOVQ    (TLS), CX
  0x001d 00029 (<autogenerated>:1) FUNCDATA $0,
gclocals d4dc2f11db048877dbc0f60a22b4adb3(SB)
  0x001d 00029 (<autogenerated>:1) FUNCDATA $1,
gclocals 33cdeccccebe80329f1fdbee7f5874cb(SB)
```

Строки, содержащие директивы `FUNCDATA` и `PCDATA`, автоматически генерируются компилятором Go, затем используются сборщиком мусора Go.

У предыдущей команды есть другой вариант:

```
$ GOOS=darwin GOARCH=amd64 go build -gcflags -S goEnv.go
```

Переменная `G00S` может принимать значения `android`, `darwin`, `dragonfly`, `freebsd`, `linux`, `nacl`, `netbsd`, `openbsd`, `plan9`, `solaris`, `windows` и `zos`; переменная `GOARCH` — значения `386`, `amd64`, `amd64p32`, `arm`, `armbe`, `arm64`, `arm64be`, `ppc64`, `ppc64le`, `mips`, `mipsle`, `mips64`, `mips64le`, `mips64p32`, `mips64p32le`, `ppc`, `s390`, `s390x`, `spar`, `spar` и `sparc64`.



Если вас действительно заинтересовал Go-ассемблер и вы хотите получить дополнительную информацию, посетите сайт <https://golang.org/doc/asm>.

Узловые деревья

Узлы Go (nodes) — это объекты типа `struct` с большим количеством свойств. Подробнее об определении и использовании *структур* в Go вы узнаете из главы 4. Все элементы Go-программы подвергаются синтаксическому разбору и анализируются модулями компилятора Go в соответствии с грамматикой языка. Конечным результатом анализа Go-кода являются соответствующее ему *дерево*. Дерево — это альтернативный способ представления программы, предназначенный не для разработчика, а для компилятора.



Обратите внимание, что команда `go tool 6g -W test.go` не будет работать в новых версиях Go. Вместо нее следует использовать команду `go tool compile -W test.go`.

В этом разделе мы сначала используем в качестве примера следующий Go-код, который хранится в файле `nodeTree.go`. Это позволит нам увидеть информацию низкого уровня, предоставляемую `go tool`:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello there!")
}
```

Go-код `nodeTree.go` довольно понятен, поэтому вы не удивитесь тому, что он выводит следующие данные:

```
$ go run nodeTree.go
Hello there!
```

Теперь пора взглянуть на некоторые внутренние операции Go, для чего выполним следующую команду:

```
$ go tool compile -W nodeTree.go
before walk main
. CALLFUNC 1(8) tc(1) STRUCT-(int, error)
. . NAME-fmt.Println a(true) 1(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
. . DDDARG 1(8) esc(no) PTR64-*[1]interface {}
. CALLFUNC-list
. . CONVIFACE 1(8) esc(h) tc(1) implicit(true) INTER-interface {}
. . . NAME-main.statictmp_0 a(true) 1(8) x(0) class(PEXTERN) tc(1) used string
. VARKILL 1(8) tc(1)
. . NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N) used
ARRAY-[1]interface {}
after walk main
. CALLFUNC-init
. . AS 1(8) tc(1)
. . . NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
. . AS 1(8) tc(1)
. . . NAME-main..autotmp_2 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used PTR64-*[1]interface {}
. . . ADDR 1(8) tc(1) PTR64-*[1]interface {}
. . . . NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
. . BLOCK 1(8)
. . BLOCK-list
. . . AS 1(8) tc(1) hascall
. . . . INDEX 1(8) tc(1) assigned bounded hascall INTER-interface {}
. . . . IND 1(8) tc(1) implicit(true) assigned hascall ARRAY-[1]
interface {}
. . . . . NAME-main..autotmp_2 a(true) 1(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
. . . . . LITERAL-0 1(8) tc(1) int
. . . . . EFACE 1(8) tc(1) INTER-interface {}
. . . . . ADDR a(true) 1(8) tc(1) PTR64-*uint8
. . . . . NAME-type.string a(true) x(0) class(PEXTERN) tc(1) uint8
. . . . . ADDR 1(8) tc(1) PTR64-*string
. . . . . NAME-main.statictmp_0 a(true) 1(8) x(0)
class(PEXTERN) tc(1) addrtaken used string
. . BLOCK 1(8)
. . BLOCK-list
. . . AS 1(8) tc(1) hascall
. . . . NAME-main..autotmp_1 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[1]interface {}
. . . . SLICEARR 1(8) tc(1) hascall SLICE-[1]interface {}
. . . . . NAME-main..autotmp_2 a(true) 1(8) x(0) class(PAUTO)
```



```

esc(N) tc(1) assigned used PTR64-*[1]interface {}
. CALLFUNC l(8) tc(1) hascall STRUCT-(int, error)
. . NAME-fmt.Println a(true) l(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
. . DDDARG l(8) esc(no) PTR64-*[1]interface {}
. CALLFUNC-list
. . AS l(8) tc(1)
. . . INDREGSP-SP a(true) l(8) x(0) tc(1) addrtaken main.__ SLICE-[]interface {}
. . . NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
. VARKILL l(8) tc(1)
. . NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
addrtaken assigned used ARRAY-[1]interface {}
before walk init
. IF l(1) tc(1)
. . GT l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . RETURN l(1) tc(1)
. IF l(1) tc(1)
. . EQ l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . CALLFUNC l(1) tc(1)
. . . NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
. . LITERAL-1 l(1) tc(1) uint8
. CALLFUNC l(1) tc(1)
. . NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
. . LITERAL-2 l(1) tc(1) uint8
. RETURN l(1) tc(1)
after walk init
. IF l(1) tc(1)
. . GT l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . RETURN l(1) tc(1)

```

```

.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
.   .   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1) hascall
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-1 l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1) hascall
.   .   NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned used uint8
.   .   LITERAL-2 l(1) tc(1) uint8
.   RETURN l(1) tc(1)

```

Как можно понять, компилятор Go и его инструменты делают многое незаметно для нас, даже в случае столь небольшой программы, как `nodeTree.go`.



Параметр `-W` дает команде компиляции `go tool` инструкцию вывести на экран дерево синтаксического анализа отладки (debug parse tree) после проверки соответствия типов.

Посмотрим на результаты следующих двух команд:

```

$ go tool compile -W nodeTree.go | grep before
before walk main
before walk init
$ go tool compile -W nodeTree.go | grep after
after walk main
after walk init

```

Как вы видите, ключевое слово `before` означает начало выполнения функции. Если бы у нашей программы было больше функций, то мы бы получили больше выходных данных, как показано в следующем примере:

```

$ go tool compile -W defer.go | grep before
before d1
before d2
before d3
before main
before d2.func1
before d3.func1
before init
before type..hash.[2]interface {}
before type..eq.[2]interface {}

```

В этом примере использован Go-код из файла `defer.go`. Он намного сложнее, чем `nodeTree.go`. Однако очевидно, что функция `init()` автоматически генерируется Go, поскольку она существует в обоих результатах выполнения команды `go tool compile -W` (для `nodeTree.go` и `defer.go`). Сейчас вы увидите более интересную версию `nodeTree.go` с именем `nodeTreeMore.go`:

```
package main

import (
    "fmt"
)

func functionOne(x int) {
    fmt.Println(x)
}

func main() {
    varOne := 1
    varTwo := 2
    fmt.Println("Hello there!")
    functionOne(varOne)
    functionOne(varTwo)
}
```

В программе `nodeTreeMore.go` определены две переменные с именами `varOne` и `varTwo` и дополнительная функция с именем `functionOne`. Поиск имен `varOne`, `varTwo` и `functionOne` в результатах выполнения команды `go tool compile -W` даст нам следующее:

```
$ go tool compile -W nodeTreeMore.go | grep functionOne | uniq
before walk functionOne
after walk functionOne
. . . NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used FUNC-func(int)
$ go tool compile -W nodeTreeMore.go | grep varTwo | uniq
. . . NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used int
. . . NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used int
$ go tool compile -W nodeTreeMore.go | grep varOne | uniq
. . . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
. . . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
```

Как видим, переменная `varOne` представлена здесь как `NAME-main.varOne`, `varTwo` — как `NAME-main.varTwo`, а функция `functionOne()` — как `NAME-main.functionOne`. Соответственно, функция `main()` здесь называется `NAME-main`.

Теперь рассмотрим следующий код дерева синтаксического анализа отладки для программы `nodeTreeMore.go`:

```
before walk functionOne
. AS l(8) tc(1)
```

```
. . NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1) assigned
used int
. . NAME-main.x a(true) g(1) l(7) x(0) class(PPARAM) tc(1) used int
```

Эти данные соответствуют определению функции `functionOne()`. Запись `l(8)` говорит о том, что определение этого узла находится в строке 8, то есть после чтения строки 7. Целочисленная переменная `NAME-main..autotmp_2` генерируется компилятором автоматически.

Рассмотрим следующий фрагмент данных дерева синтаксического анализа отладки:

```
. CALLFUNC l(15) tc(1)
. . NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used FUNC-func(int)
. CALLFUNC-list
. . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used int
```

Первая строка сообщает о том, что в строке 15 программы, на что указывает запись `l(15)`, вызывается функция `NAME-main.functionOne`, которая, судя по записи `l(7)`, определена в строке 7 программы. Запись `FUNC-func(int)` говорит о том, что эта функция принимает один целочисленный параметр. Список параметров функции, который указывается после `CALLFUNC-list`, включает в себя переменную `NAME-main.varOne`, которая, согласно записи `l(12)`, определена в строке 12 программы.

Хотите знать больше о go build?

Если вы хотите больше узнать о том, что происходит внутри компилятора, когда выполняется команда `go build`, добавьте к ней флаг `-x`:

```
$ go build -x defer.go
WORK=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-build254573394
mkdir -p $WORK/b001/
cat >$WORK/b001/importcfg.link << 'EOF' # internal
packagefile command-line-arguments=/Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
packagefile fmt=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/fmt.a
packagefile
runtime=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime.a
packagefile
errors=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/errors.a
packagefile io=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/io.a
packagefile
math=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math.a
packagefile os=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/os.a
```

```

packagefile
reflect=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/reflect.a
packagefile
strconv=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/strconv.a
packagefile
sync=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync.a
packagefile
unicode/utf8=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode/utf8.a
packagefile
internal/bytealg=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/
bytealg.a
packagefile
internal/cpu=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/cpu.a
packagefile
runtime/internal/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/
runtime/internal/atomic.a
packagefile
runtime/internal/sys=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime/
internal/sys.a
packagefile
sync/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync/atomic.a
packagefile
internal/poll=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/poll.a
packagefile
internal/syscall/unix=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/
internal/syscall/unix.a
packagefile
internal/testlog=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/
testlog.a
packagefile
syscall=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/syscall.a
packagefile
time=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/time.a
packagefile
unicode=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode.a
packagefile
math/bits=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math/bits.a
packagefile
internal/race=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/race.a
EOF
mkdir -p $WORK/b001/exe/
cd .
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/link -o
$WORK/b001/exe/a.out -importcfg $WORK/b001/importcfg.link -buildmode=exe
-buildid=nkFdi6n3HGYZXDdCOju1/VfKOjehfe3PSzik3cZom/OthUDj9rTh0tZPf-2627/
nkFdi6n3HGYZXDdCOju1 -extld=clang /Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d

```

```
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/buildid -w  
$WORK/b001/exe/a.out # internal  
mv $WORK/b001/exe/a.out defer  
rm -r $WORK/b001/
```

Повторю: очень многое из происходящего скрыто от нас, и хорошо бы об этом помнить. Однако в большинстве случаев вам не придется иметь дело с детальными командами процесса компиляции.

Создание кода WebAssembly

Go позволяет создавать код WebAssembly с помощью инструмента `go`. Прежде чем показать этот процесс, приведу дополнительную информацию о WebAssembly.

Краткое введение в WebAssembly

WebAssembly (Wasm) — это машинная модель и исполняемый формат для виртуальной машины, разработанный для повышения скорости и сокращения размера файла. Это означает, что двоичный файл WebAssembly можно использовать на любой платформе без каких-либо изменений.

WebAssembly поставляется в двух форматах: простом текстовом и двоичном. Файлы WebAssembly в простом текстовом формате имеют расширение `.wat`, а двоичные файлы — `.wasm`. Обратите внимание, что для загрузки и использования двоичного файла WebAssembly необходимо использовать JavaScript API.

Кроме Go, код WebAssembly можно генерировать из других языков программирования, которые поддерживают статическую типизацию, таких как Rust, C и C++.

Почему WebAssembly так важен

WebAssembly важен по следующим причинам:

- ❑ код WebAssembly работает со скоростью, очень близкой к «родной» скорости машины, то есть быстро;
- ❑ код WebAssembly можно генерировать из многих языков программирования, в том числе, возможно, и тех, которые вы уже знаете;
- ❑ большинство современных браузеров изначально поддерживают WebAssembly, не требуя подключения плагина или какого-либо другого программного обеспечения;
- ❑ код WebAssembly работает намного быстрее, чем код JavaScript.

Go и WebAssembly

Для Go WebAssembly — это просто еще одна архитектура. Поэтому для создания кода WebAssembly допускается использовать возможности кросс-компиляции Go.

Подробнее о возможностях кросс-компиляции Go вы прочитаете в главе 11. А пока обратите внимание на значения переменных среды `GOOS` и `GOARCH`, которые используются при компиляции Go-кода в WebAssembly, — именно здесь происходит вся магия.

Пример

В этом разделе будет показано, как скомпилировать Go-программу в код WebAssembly. Go-код программы `toWasm.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Creating WebAssembly code from Go!")
}
```

Здесь важно отметить, что в данном коде нет признаков WebAssembly, и `toWasm.go` можно скомпилировать и выполнить как отдельную программу. Другими словами, у данной программы нет внешних зависимостей, связанных с WebAssembly.

Последний шаг, который нужно будет сделать для создания кода WebAssembly, — выполнить следующую команду:

```
$ GOOS=js GOARCH=wasm go build -o main.wasm toWasm.go
$ ls -l
total 4760
-rwxr-xr-x  1 mtsouk  staff   2430633  Jan 19 21:00 main.wasm
-rw-r--r--@ 1 mtsouk  staff      100   Jan 19 20:53 toWasm.go
$ file main.wasm
main.wasm: , created: Thu Oct 25 20:41:08 2007, modified: Fri May 28 13:51:43 2032
```

Как видим, значения `GOOS` и `GOARCH`, указанные в первой команде, дают инструкцию Go сгенерировать код WebAssembly. Если не указать правильные значения `GOOS` и `GOARCH`, то при компиляции не будет создан код WebAssembly или же возникнет сбой.

Использование сгенерированного кода WebAssembly

Пока мы создали только двоичный файл WebAssembly. Но это еще не все. Нужно сделать еще кое-что, прежде чем мы сможем использовать этот двоичный файл WebAssembly и увидеть его результаты в окне браузера.



Если вы используете браузер Google Chrome, то можно применить флаг, который позволяет включить Liftoff — компилятор для WebAssembly, что теоретически улучшает время выполнения кода WebAssembly. Попробуйте — это не больно! Чтобы активизировать этот флаг, откройте страницу <chrome://flags/#enable-webassembly-baseline>.

Прежде всего нужно скопировать `main.wasm` в каталог веб-сервера, а затем выполнить следующую команду:

```
$ cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

Эта команда скопирует файл `wasm_exec.js` из установочного каталога Go в текущий каталог. Поместите этот файл в тот же каталог веб-сервера, что и файл `main.wasm`.

Мы не будем приводить здесь код JavaScript, который содержится в `wasm_exec.js`. Представим HTML-код файла `index.html`:

```
<HTML>

<head>
  <meta charset="utf-8">
  <title>Go and WebAssembly</title>
</head>

<body>
  <script src="wasm_exec.js"></script>
  <script>
    if (!WebAssembly.instantiateStreaming) { // polyfill
      WebAssembly.instantiateStreaming = async (resp, importObject) => {
        const source = await (await resp).arrayBuffer();
        return await WebAssembly.instantiate(source, importObject);
      };
    }

    const go = new Go();
    let mod, inst;
    WebAssembly.instantiateStreaming(fetch("main.wasm"),
      go.importObject).then((result) => {
```



```

    mod = result.module;
    inst = result.instance;
    document.getElementById("runButton").disabled = false;
  }).catch((err) => {
    console.error(err);
  });

  async function run() {
    console.clear();
    await go.run(inst);
    inst = await WebAssembly.instantiate(mod, go.importObject);
  }
</script>

<button onClick="run();" id="runButton" disabled>Run</button>
</body>
</HTML>

```

Обратите внимание, что кнопка Run, созданная этим HTML-кодом, не будет активизирована, пока не загрузится код WebAssembly.

На рис. 2.2 показан результат работы кода WebAssembly, представленный в консоли JavaScript браузера Google Chrome. В других браузерах результаты будут аналогичными.

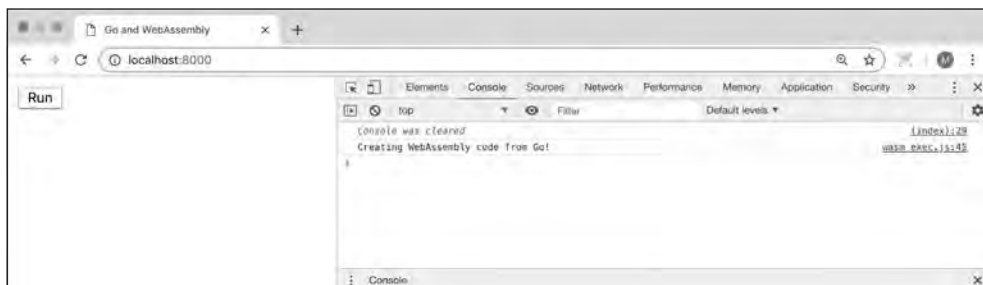


Рис. 2.2. Результат выполнения кода WebAssembly, сгенерированного в Go



В главе 12 вы узнаете, как написать на Go собственный веб-сервер.

Однако я считаю, что есть гораздо более простой и удобный способ тестирования приложений WebAssembly — с помощью *Node.js*. При использовании *Node.js* отпадает необходимость в веб-сервере, потому что *Node.js* — это среда выполнения JavaScript, построенная на движке Chrome V8 JavaScript.

Если на вашем локальном компьютере установлен Node.js, то вы можете выполнить следующую команду:

```
$ export PATH="$PATH:$(go env GOROOT)/misc/wasm"
$ GOOS=js GOARCH=wasm go run .
Creating WebAssembly code from Go!
```

Вторая команда показывает, правильно ли был выполнен код WebAssembly, и генерирует желаемое сообщение. Обратите внимание, что первая команда не является обязательной, она лишь меняет текущее значение переменной среды `PATH` на тот каталог, в котором хранятся файлы Go, связанные с WebAssembly.

Общие рекомендации по программированию на Go

Вот список практических советов, которые помогут вам написать более эффективный Go-код.

- ❑ Если в Go-функции произошла ошибка, журналируйте ее или передайте ее на уровень выше; не делайте того и другого одновременно, если только у вас нет на это действительно веской причины.
- ❑ Интерфейсы Go определяют поведение, а не данные и не структуру данных.
- ❑ По возможности используйте интерфейсы `io.Reader` и `io.Writer` — благодаря им получается более расширяемый код.
- ❑ Не передавайте в функцию указатель на простую переменную, если этого можно избежать. Лучше передайте значение переменной.
- ❑ Переменные ошибок (errors) не то же самое, что строковые переменные; это переменные ошибок!
- ❑ Не проверяйте Go-код на компьютерах, относящихся к среде эксплуатации, если только на это нет веских причин.
- ❑ Если вы недостаточно хорошо знаете какие-либо особенности Go, протестируйте их, прежде чем использовать, особенно если вы разрабатываете приложение или утилиту, которой будет пользоваться много людей.
- ❑ Если вы боитесь делать ошибки, то, скорее всего, в итоге вам не удастся сделать ничего по-настоящему интересного. Экспериментируйте, насколько хватит сил!

Упражнения и ссылки

- ❑ Познакомьтесь ближе со стандартным Go-пакетом `unsafe`, посетив страницу его документации <https://golang.org/pkg/unsafe/>.
- ❑ Посетите сайт DTrace по адресу <http://dtrace.org/>.
- ❑ Используя `strace(1)`, проверьте, как на вашем Linux-компьютере работают стандартные UNIX-утилиты, такие как `cp(1)` и `ls(1)`. Каковы оказались результаты проверки?
- ❑ Если у вас машина с macOS, воспользуйтесь `dtruss(1)`, чтобы проверить, как работает утилита `sync(8)`.
- ❑ Напишите собственный пример использования вашего C-кода из Go-программы.
- ❑ Напишите Go-функцию и используйте ее в программе на C.
- ❑ Узнайте больше о функциях пакета `runtime`, посетив страницу <https://golang.org/pkg/runtime/>.
- ❑ Читать научные труды бывает непросто, но это очень полезно. Скачайте статью *On-the-Fly Garbage Collection: An Exercise in Cooperation* и изучите ее. Эту статью можно найти на многих ресурсах, включая <https://dl.acm.org/citation.cfm?id=359655>.
- ❑ Посетите страницу <https://github.com/gasche/gc-latency-experiment>, где вы найдете код для сравнительного анализа производительности сборщиков мусора в различных языках программирования.
- ❑ Чтобы больше узнать о Node.js, посетите веб-сайт <https://nodejs.org/en/>.
- ❑ Подробнее о WebAssembly вы можете узнать на сайте <https://webassembly.org/>.
- ❑ Если хотите узнать больше о сборке мусора, обязательно посетите сайт <http://gchandbook.org/>.
- ❑ Посетите страницу документации `cgo` по адресу <https://golang.org/cmd/cgo/>.

Резюме

В этой главе рассмотрено много интересных тем, касающихся Go, в том числе теоретическая и практическая информация о сборщике мусора в Go. Вы узнали, как вызывать C-код из Go-программ; познакомились с удобным, хоть иногда и непростым, ключевым словом `defer`, функциями `panic()` и `restore()`; UNIX-инструментами `strace(1)`, `dtrace(1)` и `dtruss(1)`; узнали об использовании стандартного Go-пакета `unsafe`; научились генерировать из Go код WebAssembly и читать генерируемый Go ассемблерный код. С помощью пакета `runtime` вы получили информацию о своей Go-среде и узнали, как выявить и интерпретировать дерево узлов Go-программы. И в конце — несколько полезных советов по написанию программ на Go.

Из этой главы вам следует запомнить, что такие инструменты, как Go-пакет `unsafe` и возможность вызова C-кода из Go-программ, обычно используются в одном из трех случаев: если вы хотите добиться максимальной производительности и готовы пожертвовать для этого некоторой безопасностью Go, если вы хотите обмениваться данными с другим языком программирования и если хотите реализовать то, что невозможно реализовать в Go.

Следующая глава посвящена изучению основных типов данных, существующих в Go, включая *массивы*, *срезы* и *хеш-таблицы*. Несмотря на свою простоту, эти типы данных являются строительными блоками практически любого Go-приложения, поскольку служат основой для более сложных структур данных, которые позволяют хранить данные и перемещать информацию внутри Go-проектов.

Кроме того, вы познакомитесь с *указателями*, которые встречаются и в других языках программирования, *циклами* Go, а также с уникальным способом работы Go с датами и временем.

3

Работа с основными типами данных Go

В предыдущей главе рассмотрено много интересных тем, в том числе работа сборщика мусора Go, функции `panic()` и `recover()`, пакет `unsafe`, способы вызова C-кода из программы Go и, наоборот, Go-кода из программы на C, а также деревья узлов, создаваемые компилятором Go при компиляции Go-программы.

Главная тема этой главы — основные типы данных Go. Сюда входят *числовые типы, массивы, срезы и хеш-таблицы*. Несмотря на простоту, эти типы данных помогут вам при выполнении числовых расчетов. Они также предоставляют удобный и быстрый способ хранения, извлечения и изменения данных в Go-программе. Кроме того, вы узнаете, что такое *указатели, константы, циклы*, а также как работать с датами и временем в Go.

В данной главе рассмотрены следующие темы:

- ☐ числовые типы данных;
- ☐ массивы в Go;
- ☐ срезы Go, и почему они намного лучше, чем массивы;
- ☐ как добавить массив в существующий срез;
- ☐ хеш-таблицы Go;
- ☐ указатели в Go;
- ☐ циклы в Go;
- ☐ константы в Go;
- ☐ работа с временем;
- ☐ измерение времени выполнения команд и функций;
- ☐ работа с датами.

Числовые типы данных

В Go реализована встроенная поддержка целых чисел, чисел с плавающей точкой, а также комплексных чисел. В следующих подразделах мы ближе познакомимся со всеми числовыми типами, поддерживаемыми в Go.

Целые числа

Go поддерживает четыре размера целых чисел *со знаком* и *без знака*. Эти целочисленные типы со знаком называются `int8`, `int16`, `int32`, `int64`, а без знака — `uint8`, `uint16`, `uint32` и `uint64`. Число в конце каждого названия типа соответствует количеству битов, выделяемых для представления соответствующего числа.

Кроме того, есть типы `int` и `uint`, которые являются наиболее эффективным представлением целых чисел со знаком и без знака соответственно для текущей платформы. Поэтому в случае сомнений используйте `int` и `uint`, однако имейте в виду, что размер этих типов изменяется в зависимости от архитектуры компьютера.

Разница между целыми числами со знаком и без знака заключается в следующем: если целое число занимает 8 бит и не имеет знака, то оно может принимать значения в диапазоне от двоичного 00000000 (0) до двоичного 11111111 (255). Если же у числа есть знак, то оно может принимать значения от -127 до 127 . Таким образом, для хранения числа выделяется семь двоичных разрядов, так как восьмой бит используется для хранения знака целого числа. Это же правило применяется и к другим размерам целых чисел без знака.

Числа с плавающей точкой

Go поддерживает всего два типа чисел с плавающей точкой: `float32` и `float64`. Первый из них обеспечивает точность примерно до шестого знака после запятой, а второй — до 15-го знака.

Комплексные числа

Как и в случае с числами с плавающей запятой, Go предлагает два типа комплексных чисел с именами `complex64` и `complex128`. В первом из них используется два типа `float32`: один для вещественной части комплексного числа, а второй — для мнимой, тогда как в `complex128` используется два типа `float64`.

Комплексные числа выражаются в форме $a + bi$, где a и b — действительные числа, а i — решение уравнения $x^2 = -1$.

Все эти числовые типы данных продемонстрированы в программе `numbers.go`, которую мы рассмотрим, разделив на три части.

Первая часть `numbers.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    c1 := 12 + 1i
```

```

c2 := complex(5, 7)
fmt.Printf("Type of c1: %T\n", c1)
fmt.Printf("Type of c2: %T\n", c2)

var c3 complex64 = complex64(c1 + c2)
fmt.Println("c3:", c3)
fmt.Printf("Type of c3: %T\n", c3)

cZero := c3 - c3
fmt.Println("cZero:", cZero)

```

В этой части программы мы имеем дело с комплексными числами и выполняем над ними некоторые вычисления. Есть два способа создания комплексного числа: напрямую, как в случае с `c1` и `c2`, или косвенно, путем выполнения расчетов с существующими комплексными числами, как в случае с `c3` и `cZero`.



Если вы по ошибке попытаетесь создать комплексное число как `aComplex := 12 + 2 * i`, то возможны два варианта, поскольку Go воспринимает данный оператор как сложение и умножение. Если в текущей области видимости нет числовой переменной с именем `i`, то такой оператор приведет к появлению синтаксической ошибки и код Go не будет скомпилирован. Если же числовая переменная с именем `i` определена, то вычисление будет успешным, но в результате вы не получите желаемое комплексное число (дефект программы).

Вторая часть `numbers.go` выглядит так:

```

x := 12
k := 5
fmt.Println(x)
fmt.Printf("Type of x: %T\n", x)

div := x / k
fmt.Println("div", div)

```

В этой части мы имеем дело с целыми числами со знаком. Обратите внимание, что если разделить одно целое число на другое, то Go решит, что вы хотите получить результат целочисленного деления, и вычислит и вернет результат в виде *неполного частного от целочисленного деления*. Таким образом, попытка разделить целое число 11 на целое число 2 даст результат 5, а не 5,5, и это может удивить.



При преобразовании числа с плавающей запятой в целое число дробь отбрасывается путем усечения числа с плавающей запятой до нуля знаков после запятой. Это означает, что в процессе такой операции некоторые данные могут быть потеряны.

Последняя часть `numbers.go` содержит следующий код:

```
var m, n float64
m = 1.223
fmt.Println("m, n:", m, n)

y := 4 / 2.3
fmt.Println("y:", y)

divFloat := float64(x) / float64(k)
fmt.Println("divFloat", divFloat)
fmt.Printf("Type of divFloat: %T\n", divFloat)
}
```

В этой части программы обрабатываются числа с плавающей запятой. В представленном коде показано, как использовать функцию `float64()`, чтобы при делении одного целого числа на другое Go создавал число с плавающей точкой. Если просто ввести `divFloat := float64(x) / k`, то при выполнении кода получим следующее сообщение об ошибке:

```
$ go run numbers.go
# command-line-arguments
./numbers.go:35:25: invalid operation: float64(x) / k (mismatched types
float64 and int)
```

Выполнение `numbers.go` дает такой результат:

```
Type of c1: complex128
Type of c2: complex128
c3: (17+8i)
Type of c3: complex64
cZero: (0+0i)
12
Type of x: int
div 2
m, n: 1.223 0
y: 1.7391304347826086
divFloat 2.4
Type of divFloat: float64
```

Числовые литералы в Go 2

На момент написания этой книги поступило предложение внести изменения в поддержку Go *числовых литералов*. С помощью числовых литералов можно определять и использовать числа в языке программирования. Предложение связано с представлением двоичных и восьмеричных целочисленных литералов, разделителя между цифрами, а также с поддержкой шестнадцатеричных чисел с плавающей точкой.

Более подробную информацию о предложении, касающемся Go 2 и числовых литералов, вы найдете по адресу <https://golang.org/design/19308-number-literals>.



У разработчиков Go есть подробная информационная панель управления выпусками, которую вы можете просмотреть по адресу <https://dev.golang.org/release>.

Циклы Go

В любом языке программирования есть циклы, и Go не исключение. В Go есть цикл `for`, который позволяет осуществлять итерации по различным типам данных.



В Go нет ключевого слова `while`, но циклы `for` в Go вполне заменяют циклы `while`.

Цикл `for`

Цикл `for` позволяет выполнить определенный код заранее заданное число раз, пока действительно некое условие или в соответствии со значением, которое вычисляется в начале цикла `for`. Таким значением может быть размер среза или массива либо количество ключей в хеш-таблице. Итак, цикл `for` является самым распространенным способом перебрать все элементы массива, среза или хеш-таблицы.

Далее представлена простейшая форма цикла `for`. Переменная `i` принимает все значения из предопределенного диапазона:

```
for i := 0; i < 100; i++ {  
}
```

Вообще, цикл `for` состоит из трех частей: первая называется *инициализацией*, вторая — *условием*, а последняя — *изменением счетчика*. Ни одна из этих частей не является обязательной.

В приведенном выше цикле переменная `i` принимает значения в диапазоне от 0 до 99. Как только `i` станет равной 100, выполнение цикла `for` остановится. В данном случае `i` — локальная и временная переменная, то есть после завершения цикла `for` в какой-то момент `i` попадет в сборку мусора и исчезнет. Однако, если бы `i` была определена вне цикла `for`, она сохранила бы свое значение после завершения цикла `for`. В таком случае значение `i` после завершения цикла `for` будет равно 100, так как это последнее значение `i` в этой точке программы.

Для того чтобы выйти из цикла `for` до его завершения, можно использовать ключевое слово `break`. Ключевое слово `break` также позволяет создать цикл `for` без

условия выхода, такого как `i < 100` в предыдущем примере, поскольку тогда условие выхода может быть включено в блок кода цикла `for`. Также разрешается использовать в цикле `for` несколько условий выхода. Кроме того, можно пропустить одну итерацию цикла `for`, используя ключевое слово `continue`.

Цикл `while`

Как уже отмечалось, в Go нет ключевого слова `while` для реализации циклов `while`, однако язык позволяет использовать вместо цикла `while` цикл `for`. В этом разделе представлены два примера, где цикл `for` выполняет работу цикла `while`.

Сначала рассмотрим типичный случай того, как можно написать нечто аналогичное `while (true)`:

```
for {  
}
```

Задача разработчика заключается в том, чтобы поставить в нужном месте ключевое слово `break` для выхода из этого цикла `for`.

Однако цикл `for` также может эмулировать цикл `do ... while`, который встречается в других языках программирования.

Например, следующий код Go эквивалентен циклу `do ... while (anExpression)`:

```
for ok := true; ok; ok = anExpression {  
}
```

Как только переменная `ok` примет значение `false`, цикл `for` прекратит работу.

В Go также есть цикл `for condition {}`, который выполняется до тех пор, пока соблюдается условие `condition`.

Ключевое слово `range`

В Go есть еще ключевое слово `range`, которое используется в циклах `for` и позволяет писать легкочитаемый код для перебора любых типов данных, поддерживаемых в Go, включая *каналы*. Главным преимуществом ключевого слова `range` является то, что с ним не нужно заранее знать *емкость* (количество элементов) среза, хеш-таблицы или канала, чтобы перебрать по одному все его элементы. Далее вы узнаете, как использовать ключевое слово `range`.

Пример применения нескольких циклов Go

В этом разделе показаны несколько примеров использования циклов `for`. Имя файла с программой — `loops.go`, и для его изучения разделим программу на четыре части. Первый фрагмент кода `loops.go` выглядит так:

```

package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 100; i++ {
        if i%20 == 0 {
            continue
        }
        if i == 95 {
            break
        }

        fmt.Print(i, " ")
    }
}

```

В этом коде показан типичный цикл `for`, а также продемонстрировано использование ключевых слов `continue` и `break`.

Следующий фрагмент кода выглядит следующим образом:

```

fmt.Println()
i := 10
for {
    if i < 0 {
        break
    }
    fmt.Print(i, " ")
    i--
}
fmt.Println()

```

Этот код эмулирует типичный цикл `while`. Обратите внимание на использование ключевого слова `break` для выхода из цикла `for`.

Третья часть `loops.go` выглядит так:

```

i = 0
anExpression := true
for ok := true; ok; ok = anExpression {
    if i > 10 {
        anExpression = false
    }

    fmt.Print(i, " ")
    i++
}
fmt.Println()

```

Здесь вы видите, как цикл `for` выполняет работу цикла `do ... while`, упоминавшегося ранее в этой главе. Заметьте, что этот цикл `for` трудно читается.

Последняя часть `loops.go` содержит следующий код Go:

```
anArray := [5]int{0, 1, -1, 2, -2}
for i, value := range anArray {
    fmt.Println("index:", i, "value: ", value)
}
}
```

Применение ключевого слова `range` к переменной массива возвращает два значения: индекс массива и значение элемента, соответствующее этому индексу.

Вы можете использовать оба этих значения, только одно из них или ни одного, если просто хотите сосчитать элементы массива или выполнить какую-либо другую задачу столько раз, сколько элементов содержится в массиве.

Выполнение `loops.go` даст следующий результат:

```
$ go run loops.go
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 81 82
83 84 85 86 87 88 89 90 91 92 93 94
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11
index: 0 value: 0
index: 1 value: 1
index: 2 value: -1
index: 3 value: 2
index: 4 value: -2
```

Массивы в Go

Массивы являются одной из самых популярных структур данных по двум причинам. Первая состоит в том, что массивы просты и понятны, а вторая — в том, что они универсальны и позволяют хранить самые разные виды данных.

Ниже показано, как можно объявить массив, в котором хранятся четыре целых числа:

```
anArray := [4]int{1, 2, 4, -4}
```

Сначала указывается размер массива, затем его тип, а в конце — список элементов. Узнать длину массива можно с помощью функции `len()`: `len(anArray)`.

Индекс первого элемента любого измерения массива равен нулю, индекс второго элемента — единице и т. д. Это означает, что для одномерного массива с именем `a` допустимы индексы от 0 до `len(a) - 1`.

Возможно, вы знакомы со способами доступа к элементам массива в других языках программирования и использованием цикла `for` с одной или несколькими числовыми переменными, однако в Go есть более характерные для данного языка

способы перебора всех элементов массива. Они предполагают использование ключевого слова `range` и позволяют обойти использование функции `len()` в цикле `for`. Мы уже рассмотрели в качестве такого примера Go-код в файле `loops.go`.

Многомерные массивы

Массивы могут быть многомерными. Однако использование более трех измерений без серьезных причин может затруднить чтение программы и привести к ошибкам.



Массивы позволяют хранить элементы любых типов; мы используем целые числа для простоты, поскольку их легче понять и выводить на экран.

В следующем коде Go показано, как создать двумерный массив `twoD` и трехмерный массив `threeD`:

```
twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}
threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Мы можем легко прочитать, изменить или вывести на экран отдельный элемент любого из этих двух массивов. Например, первый элемент массива `twoD` — это `twoD[0][0]`, и его значение равно 1.

Таким образом, доступ ко всем элементам массива `threeD` можно организовать с помощью нескольких циклов `for` следующим образом:

```
for i := 0; i < len(threeD); i++ {
    for j := 0; j < len(v); j++ {
        for k := 0; k < len(m); k++ {
            }
        }
    }
}
```

Как вы видите, для того чтобы получить доступ ко всем элементам массива, нам нужно столько же циклов `for`, сколько измерений у этого массива. Это же правило применяется к срезам, которые будут рассмотрены в следующем разделе. Использование `x`, `y` и `z` в качестве имен переменных вместо `i`, `j` и `k` может быть хорошей идеей.

Полный пример того, как обращаться с массивами в Go, представлен в коде `usingArrays.go`, который мы рассмотрим, разделив на три части.

Первая часть кода выглядит следующим образом:

```
package main

import (
```

```

    "fmt"
)

func main() {
    anArray := [4]int{1, 2, 4, -4}
    twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14, 15, 16}}
    threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}

```

Здесь мы определяем три переменные массива с именами `anArray`, `twoD` и `threeD` соответственно.

Вторая часть `Arrays.go` выглядит так:

```

fmt.Println("The length of", anArray, "is", len(anArray))
fmt.Println("The first element of", twoD, "is", twoD[0][0])
fmt.Println("The length of", threeD, "is", len(threeD))

for i := 0; i < len(threeD); i++ {
    v := threeD[i]
    for j := 0; j < len(v); j++ {
        m := v[j]
        for k := 0; k < len(m); k++ {
            fmt.Print(m[k], " ")
        }
    }
    fmt.Println()
}

```

Из первого цикла `for` получаем двумерный массив `threeD[i]`, а из второго цикла `for` — одномерный массив `v[j]`. Последний цикл `for` перебирает элементы этого одномерного массива.

Последняя часть кода содержит следующий код Go:

```

for _, v := range threeD {
    for _, m := range v {
        for _, s := range m {
            fmt.Print(s, " ")
        }
    }
    fmt.Println()
}

```

Ключевое слово `range` выполняет ту же работу, что и переменные итераций, использованные в циклах `for` предыдущего фрагмента кода, но только более элегантно и понятно. Однако если вы хотите заранее узнать количество итераций, которые будут выполнены, то использовать ключевое слово `range` нельзя.



Ключевое слово `range` также работает с хеш-таблицами Go, что делает его весьма удобным и предпочтительным способом перебора значений. Как вы увидите в главе 9, ключевое слово `range` также работает с каналами.

Выполнение `usingArrays.go` приведет к следующим результатам:

```
$ go run usingArrays.go
The length of [1 2 4 -4] is 4
The first element of [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16]] is 1
The length of [[[1 0] [-2 4]] [[5 -1] [7 0]]] is 2
1 0 -2 4
5 -1 7 0
1 0 -2 4
5 -1 7 0
```

Одна из самых больших проблем при работе с массивами — это *ошибки выхода за пределы размерности массива*, что означает попытку доступа к несуществующему элементу. Например, это может быть попытка получить доступ к шестому элементу массива, состоящего всего из пяти элементов. Компилятор Go распознает такие проблемы и сообщает о них как об ошибках компиляции, потому что это помогает в процессе разработки. Таким образом, компилятор Go может обнаруживать ошибки выхода за пределы массива:

```
./a.go:10: invalid array index -1 (index must be non-negative)
./a.go:10: invalid array index 20 (out of bounds for 2-element array)
```

Недостатки массивов Go

У массивов Go есть много недостатков, которые, возможно, заставят вас подумать дважды, прежде чем использовать массивы в своих проектах Go. Прежде всего, после того как массив определен, вы не можете изменить его размер — другими словами, массивы Go не являются динамическими. Таким образом, для того, чтобы добавить элемент в существующий массив, в котором больше нет свободного места, нужно создать массив большего размера и скопировать в него все элементы старого массива. Кроме того, передавая массив функции в качестве параметра, вы на самом деле передаете копию массива, так что любые изменения, внесенные в массив внутри функции, после выхода из этой функции будут потеряны. Наконец, передача большого массива в функцию — довольно медленная операция, главным образом потому, что при этом Go создает копию массива. Решением всех этих проблем является использование срезов Go, которые представлены в следующем разделе.



Из-за своих недостатков массивы используются в Go очень редко!

Срезы в Go

Срезы Go — очень мощные структуры данных. Не будет преувеличением сказать, что срезы способны полностью заменить массивы в Go. Есть лишь несколько случаев, когда приходится использовать массив вместо среза. Наиболее очевидным является случай, когда вы абсолютно уверены в том, что нужно будет хранить фиксированное количество элементов.



Срезы в Go реализованы на основе массивов, следовательно, внутри каждого среза Go лежит массив.

Срезы передаются в функции *по ссылке* — это означает, что фактически в функцию передается адрес памяти переменной среза, и любые изменения, внесенные в срез внутри функции, не будут потеряны после выхода из нее. Кроме того, большие срезы передаются в функцию значительно быстрее, чем массивы с тем же количеством элементов, поскольку в этом случае Go не делает копию среза, а лишь передает адрес памяти переменной среза.

Выполнение основных операций со срезами

Для того чтобы создать *литерал среза*, нужно сделать следующее:

```
aSliceLiteral := []int{1, 2, 3, 4, 5}
```

Таким образом, литералы срезов определяются так же, как и массивы, но без указания количества элементов. Если добавить в это определение количество элементов, то вместо среза получим массив.

Однако, кроме этого, еще есть функция `make()`, которая позволяет создавать пустые срезы желаемой *длины* и *емкости* на основе параметров, передаваемых в `make()`. Параметр емкости можно пропустить, и тогда емкость среза будет равна его длине. Таким образом можно создать пустой срез на 20 элементов, который при необходимости автоматически расширится:

```
integer := make([]int, 20)
```

Обратите внимание, что Go автоматически инициализирует все элементы нового среза нулевыми значениями заданного типа. Это означает, что значение инициализации зависит от типа объектов, хранимых в срезе. К счастью, Go инициализирует элементы любого среза, создаваемого с помощью `make`.

Получить доступ ко всем элементам среза можно следующим образом:

```
for i := 0; i < len(integer); i++ {
    fmt.Println(integer[i])
}
```


Если необходимо очистить существующий срез, то нулевым значением для переменной среза будет `nil`:

```
asliceLiteral = nil
```

Чтобы добавить элемент к срезу, нужно воспользоваться функцией `append()`, что при необходимости автоматически увеличит размер среза:

```
integer = append(integer, 12345)
```

Чтобы получить доступ к первому элементу среза `integer`, следует обратиться к нему как `integer[0]`, а к последнему элементу среза `integer` — как к `integer[len(integer)-1]`.

Наконец, с помощью нотации `[:]` можно получить доступ к нескольким последовательным элементам среза. Этот оператор выбирает второй и третий элементы среза:

```
integer[1:3]
```

Кроме того, нотация `[:]` позволяет создать новый срез из уже существующего среза или массива:

```
s2 := integer[1:3]
```

Обратите внимание, что этот процесс, называемый *созданием вторичных срезов*, в некоторых случаях может вызвать проблемы. Рассмотрим следующую программу:

```
package main
```

```
import "fmt"
```

```
func main() {
    s1 := make([]int, 5)
    reSlice := s1[1:3]
    fmt.Println(s1)
    fmt.Println(reSlice)
    reSlice[0] = -100
    reSlice[1] = 123456
    fmt.Println(s1)
    fmt.Println(reSlice)
}
```

Следует отметить, что для выбора второго и третьего элементов среза с использованием нотации `[:]` необходимо использовать запись `[1: 3]`, которая означает, что надо выбрать из среза все элементы, начиная с индекса 1 и заканчивая индексом 3, не включая индекс 3.



Для некоторого массива `a1` можно создать срез `s1`, который бы ссылался на этот массив, с помощью оператора `s1 := a1[:]`.

Выполнение предыдущего кода, сохраненного как `reslice.go`, приведет к следующим результатам:

```
$ go run reslice.go
[0 0 0 0]
[0 0]
[0 -100 123456 0 0]
[-100 123456]
```

Таким образом, в конце программы содержимым среза `s1` является `[0 -100 123456 0 0]`, несмотря на то что мы не меняли его напрямую! Это означает, что при изменении элементов вторичного среза изменяются и элементы исходного среза, поскольку оба они указывают на один и тот же базовый массив. Проще говоря, при создании вторичного среза не создается копия исходного среза.

Вторая проблема создания вторичных срезов состоит в том, что базовый массив исходного среза будет сохраняться в памяти до тех пор, пока существует меньший, вторичный срез. Потому что вторичный срез ссылается на исходный срез. Так будет, даже если повторный срез создан для того, чтобы использовать только небольшую часть исходного среза. Для небольших срезов это не очень важно, однако при чтении больших файлов и сохранении данных в виде срезов, из которых вы хотите использовать лишь небольшую часть, это может вызвать проблемы.

Автоматическое расширение срезов

Срезы характеризуются двумя основными свойствами: *емкостью* (*capacity*) и *длиной* (*length*). Фокус в том, что обычно эти два свойства имеют разные значения. Длина среза равна длине массива с таким же количеством элементов и может быть получена с помощью функции `len()`. Емкость среза — это то, сколько места в памяти занимает срез в данный момент. Емкость можно узнать с помощью функции `cap()`. Поскольку размер срезов является динамической величиной, то, если длина среза выходит за пределы выделенной для него памяти, Go автоматически удваивает текущую емкость среза, чтобы выделить место для большего количества элементов.

Проще говоря, если длина и емкость среза имеют одинаковые значения, то попытка добавить в срез еще один элемент приводит к тому, что емкость среза будет удвоена, тогда как его длина увеличится только на единицу.

В случае небольших срезов такой прием может быть достаточно эффективным, однако добавление еще одного элемента в действительно огромный срез может потребовать больше памяти, чем вы ожидаете.

Представленный далее код `lenCap.go` более подробно иллюстрирует понятия емкости и длины среза. Мы рассмотрим его, разделив на три части. Первая часть программы выглядит так:

```
package main

import (
    "fmt"
)
```

```
func printSlice(x []int) {
    for _, number := range x {
        fmt.Print(number, " ")
    }
    fmt.Println()
}
```

Функция `printSlice()` помогает вывести на экран одномерный срез без необходимости постоянно повторять один и тот же код Go.

Вторая часть `lenCap.go` содержит следующий код:

```
func main() {
    aSlice := []int{-1, 0, 4}
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
    aSlice = append(aSlice, -100)
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}
```

В этой, а также в следующей части программы мы добавим в срез `aSlice` новые элементы, чтобы изменить его длину и емкость.

Последняя часть кода Go выглядит так:

```
    aSlice = append(aSlice, -2)
    aSlice = append(aSlice, -3)
    aSlice = append(aSlice, -4)
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}
```

Выполнение `lenCap.go` приведет к следующим результатам:

```
$ go run lenCap.go
aSlice: -1 0 4
Cap: 3, Length: 3
aSlice: -1 0 4 -100
Cap: 6, Length: 4
-1 0 4 -100 -2 -3 -4
Cap: 12, Length: 7
```

Как видно, начальная длина среза была равна 3 и его первоначальная емкость тоже равнялась 3. После добавления в срез нового элемента его длина стала равна 4, а емкость — 6. После добавления в срез еще трех элементов его длина стала равна 7, тогда как емкость еще раз удвоилась¹ и стала равной 12.

¹ Нужно заметить, что емкость среза удваивается, только пока его длина не превысит 1024 элементов. Далее она растет не более чем на 25 %.

Байтовые срезы

Байтовый срез — это срез, типом которого является `byte`. Чтобы создать байтовый срез с именем `s`, нужно выполнить следующий код:

```
s := make([]byte, 5)
```

Go знает, что в большинстве случаев байтовые срезы используются для хранения строк, и поэтому позволяет легко переключаться между этим типом и типом `string`. По способу доступа байтовый срез ничем не отличается от других типов срезов. Просто байтовые срезы используются в операциях ввода и вывода файлов. Как используются байтовые срезы, вы прочитаете в главе 8.

Функция `copy()`

С помощью функции `copy()` можно создать срез из элементов существующего массива или как копию существующего среза. Однако, поскольку функция `copy()` может быть очень запутанной, в этом подразделе разъясняется ее использование с помощью кода Go `copySlice.go`, который разделим на четыре части.



При использовании функции `copy()` для срезов следует быть очень осторожным, поскольку количество элементов, копируемых встроенной функцией `copy(dst, src)`, составляет минимум из `len(dst)` и `len(src)`.

Первая часть программы содержит следующий код Go:

```
package main

import (
    "fmt"
)

func main() {
    a6 := []int{-10, 1, 2, 3, 4, 5}
    a4 := []int{-1, -2, -3, -4}
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)

    copy(a6, a4)
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)
    fmt.Println()
}
```

Как видите, в этом коде созданы два среза с именами `a6` и `a4`, их содержимое выводится на экран, а затем предпринимается попытка скопировать `a4` в `a6`.

Поскольку в `a6` больше элементов, чем в `a4`, все элементы `a4` будут скопированы в `a6`. Однако, поскольку у `a4` только четыре элемента, а у `a6` — шесть элементов, последние два элемента `a6` останутся прежними.

Вторая часть `copySlice.go` выглядит так:

```
b6 := []int{-10, 1, 2, 3, 4, 5}
b4 := []int{-1, -2, -3, -4}
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
copy(b4, b6)
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
```

В этом случае в `b4` будут скопированы только первые четыре элемента `b6`, так как емкость `b4` составляет всего четыре элемента.

Третий фрагмент кода `copySlice.go` содержит следующий код Go:

```
fmt.Println()
array4 := [4]int{4, -4, 4, -4}
s6 := []int{1, 1, -1, -1, 5, -5}
copy(s6, array4[0:])
fmt.Println("array4:", array4[0:])
fmt.Println("s6:", s6)
fmt.Println()
```

Здесь массив из четырех элементов копируется в срез из шести элементов. Обратите внимание, что массив преобразуется в срез с помощью нотации `[ : ]` (`array4[0:]`).

Последняя часть кода `copySlice.go` выглядит следующим образом:

```
array5 := [5]int{5, -5, 5, -5, 5}
s7 := []int{7, 7, -7, -7, 7, -7, 7}
copy(array5[0:], s7)
fmt.Println("array5:", array5)
fmt.Println("s7:", s7)
}
```

Здесь показано, как скопировать срез в массив, в котором есть место для пяти элементов. Поскольку `copy()` принимает только аргументы среза, необходимо использовать нотацию `[ : ]` для преобразования массива в срез.

Если вы попытаетесь скопировать массив в срез или наоборот, не используя нотацию `[ : ]`, программа не будет скомпилирована и вы получите одно из следующих сообщений об ошибке:

```
# command-line-arguments
./a.go:42:6: first argument to copy should be slice; have [5]int
./a.go:43:6: second argument to copy should be slice or string; have [5]int
./a.go:44:6: arguments to copy must be slices; have [5]int, [5]int
```

Выполнение `copySlice.go` приведет к следующим результатам:

```
$ go run copySlice.go
a6: [-10 1 2 3 4 5]
a4: [-1 -2 -3 -4]
a6: [-1 -2 -3 -4 4 5]
a4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-10 1 2 3]
array4: [4 -4 4 -4]
s6: [4 -4 4 -4 5 -5]
array5: [7 7 -7 -7 7]
s7: [7 7 -7 -7 7 -7 7]
```

Многомерные срезы

Срезы, как массивы, могут быть многомерными. Следующая инструкция создает двумерный срез:

```
s1 := make([][]int, 4)
```



Если вам постоянно приходится использовать многомерные срезы, возможно, имеет смысл пересмотреть подход и выбрать более простую структуру программы, которая бы не требовала многомерных срезов.

В следующем разделе рассмотрим пример кода с использованием многомерного среза.

Еще один пример использования срезов

Будем надеяться, что код Go в программе `slices.go` прояснит многие аспекты использования срезов. Рассмотрим этот код, разделив его на пять частей.

Первая часть программы содержит обычную преамбулу, а также определение двух срезов:

```
package main

import (
    "fmt"
)

func main() {
    aSlice := []int{1, 2, 3, 4, 5}
    fmt.Println(aSlice)
```

```

integers := make([]int, 2)
fmt.Println(integers)
integers = nil
fmt.Println(integers)

```

Во второй части показано, как использовать нотацию `[]` для создания нового среза, который ссылается на существующий массив. Помните, что мы создаем не копию массива, а лишь ссылку на него, в чем вы убедитесь при выводе данных программы:

```

anArray := [5]int{-1, -2, -3, -4, -5}
refAnArray := anArray[:]

fmt.Println(anArray)
fmt.Println(refAnArray)
anArray[4] = -100
fmt.Println(refAnArray)

```

В третьем фрагменте кода с помощью функции `make()` создаются два среза, одномерный и двумерный:

```

s := make([]byte, 5)
fmt.Println(s)
twoD := make([][]int, 3)
fmt.Println(twoD)
fmt.Println()

```

Поскольку срезы в Go инициализируются автоматически, все элементы этих двух срезов будут иметь нулевое значение, соответствующее типу данного среза, которое для целых чисел равно `0`, а для срезов — `nil`. Напомню, что элементы многомерного среза, в свою очередь, являются срезами.

В четвертой части файла `slices.go`, которая содержит следующий код Go, показано, как вручную инициализировать все элементы двумерного среза:

```

for i := 0; i < len(twoD); i++ {
    for j := 0; j < 2; j++ {
        twoD[i] = append(twoD[i], i*j)
    }
}

```

Как видно из этого кода Go, для того чтобы расширить существующий срез и увеличить его размер, недостаточно просто сослаться на несуществующий индекс — нужно явно использовать функцию `append()`! Обращение к несуществующему индексу приведет к ошибке `panic: runtime error: index out of range`. Обратите внимание, что значения элементов среза выбирались произвольно.

В последней части программы показано, как использовать ключевое слово `range` для перебора и вывода на экран всех элементов двумерного среза:

```

for _, x := range twoD {
    for i, y := range x {

```

```

        fmt.Println("i:", i, "value:", y)
    }
    fmt.Println()
}
}

```

Если выполнить `slices.go`, то получим следующий результат:

```

$ go run slices.go
[1 2 3 4 5]
[0 0]
[]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -100]
[0 0 0 0 0]
[[] [] []]
i: 0 value: 0
i: 1 value: 0
i: 0 value: 0
i: 1 value: 1
i: 0 value: 0
i: 1 value: 2

```

Пусть вас не удивляет, что объекты двумерного среза инициализируются значениями `nil` и поэтому выводятся как пустые; это происходит потому, что нулевое значение для типа среза равно `nil`.

Сортировка срезов с помощью `sort.Slice()`

В этом подразделе вы увидите, как использовать функцию `sort.Slice()`, которая впервые появилась в Go версии 1.8. Это означает, что представленный здесь код, который хранится в файле `sortSlice.go`, не будет работать в более старых версиях Go. Рассмотрим эту программу, разделив ее на три части. Первая часть выглядит так:

```

package main

import (
    "fmt"
    "sort"
)

type aStructure struct {
    person string
    height int
    weight int
}

```


Помимо ожидаемой преамбулы, в этой книге впервые приведено определение структуры Go. Структуры Go подробно рассматриваются в главе 4. Пока просто запомните, что структуры — это типы данных, в которых хранится несколько переменных разных типов.

Вторая часть `sortSlice.go` содержит следующий код Go:

```
func main() {
    mySlice := make([]aStructure, 0)
    mySlice = append(mySlice, aStructure{"Mihalis", 180, 90})
    mySlice = append(mySlice, aStructure{"Bill", 134, 45})
    mySlice = append(mySlice, aStructure{"Marietta", 155, 45})
    mySlice = append(mySlice, aStructure{"Epifanios", 144, 50})
    mySlice = append(mySlice, aStructure{"Athina", 134, 40})

    fmt.Println("0:", mySlice)
```

Здесь создается новый срез с именем `mySlice`, элементами которого являются элементы созданной ранее структуры `aStructure`.

Заключительная часть программы выглядит так:

```
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height < mySlice[j].height
    })
    fmt.Println("<:", mySlice)
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height > mySlice[j].height
    })
    fmt.Println(">:", mySlice)
}
```

Здесь дважды выполняется сортировка `mySlice`, для чего используются функция `sort.Slice()` и две анонимные функции. Анонимные функции вызываются последовательно в ходе сортировки, с использованием поля `height` структуры `aStructure`.



Обратите внимание, что `sort.Slice()` изменяет последовательность элементов в срезе в соответствии с функцией сортировки.

Выполнение `sortSlice.go` приведет к следующим результатам:

```
$ go run sortSlice.go
0: [{Mihalis 180 90} {Bill 134 45} {Marietta 155 45} {Epifanios 144 50}
{Athina 134 40}]
<: [{Bill 134 45} {Athina 134 40} {Epifanios 144 50} {Marietta 155 45}
{Mihalis 180 90}]
>: [{Mihalis 180 90} {Marietta 155 45} {Epifanios 144 50} {Bill 134 45}
{Athina 134 40}]
```

Если вы попытаете выполнить `sortSlice.go` на компьютере UNIX с версией Go ранее 1.8, то получите следующее сообщение об ошибке:

```
$ go version
o version go1.3.3 linux/amd64
$ go run sortSlice.go
# command-line-arguments
./sortSlice.go:24: undefined: sort.Slice
./sortSlice.go:28: undefined: sort.Slice
```

Добавление массива к срезу

В этом подразделе показано, как добавить существующий массив к существующему срезу, на примере способа, приведенного в программе `appendArrayToSlice.go`. Рассмотрим эту программу, разделив ее на две части. Первая часть выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    s := []int{1, 2, 3}
    a := [3]int{4, 5, 6}
```

Пока мы только создали и инициализировали срез с именем `s` и массив с именем `a`. Вторая часть `appendArrayToSlice.go` выглядит так:

```
    ref := a[:]
    fmt.Println("Existing array:\t", ref)
    t := append(s, ref...)
    fmt.Println("New slice:\t", t)
    s = append(s, ref...)
    fmt.Println("Existing slice:\t", s)
    s = append(s, s...)
    fmt.Println("s+s:\t\t", s)
}
```

Здесь два важных момента. Во-первых, мы создаем новый срез с именем `t`, который содержит элементы `a + s`, — добавляем массив `a` к срезу `s` и сохраняем результат в срезе `s`. Таким образом, появляется выбор: сохранить новый срез в существующей переменной среза или нет. Решение зависит главным образом от того, что вы хотите получить.

Второй важный момент: для того чтобы этот код работал, необходимо создать ссылку на существующий массив (`ref := a[:]`). Обратите внимание, как переменная `ref` используется в двух вызовах `append()`: многоточие (...) разбивает массив на элементы, которые добавляются к существующему срезу.

Последние два оператора программы показывают, как можно скопировать срез до конца. Для этого тоже следует использовать многоточие (...).

Выполнение `appendArrayToSlice.go` приведет к следующим результатам:

```
$ go run appendArrayToSlice.go
Existing array:   [4 5 6]
New slice:       [1 2 3 4 5 6]
Existing slice:  [1 2 3 4 5 6]
s+s:            [1 2 3 4 5 6 1 2 3 4 5 6]
```

Хеш-таблицы Go

Хеш-таблицы, или карты (map) Go — то же, что известные хеш-таблицы, существующие во многих других языках программирования. Основным их преимуществом является способность использовать любой тип данных в качестве индекса, который в этом случае называется *ключом карты* или просто *ключом*. Несмотря на то что хеш-таблицы Go допускают использование в качестве ключей любых типов данных, тип данных, используемый в качестве ключа, должен быть *сопоставимым*, то есть у компилятора Go должна быть возможность отличать один ключ от другого, или, проще говоря, ключи хеш-таблицы должны поддерживать оператор `==`.

К счастью, почти все типы данных сопоставимы. Однако, как можно догадаться, использование типа данных `bool` в качестве ключа хеш-таблицы определенно ограничило бы ее всего двумя значениями. Кроме того, применение в качестве ключей чисел с плавающей точкой может повлечь проблемы, вызванные точностью, принятой на разных машинах и в разных операционных системах.



Как уже отмечалось, карты Go — то же, что и хеш-таблицы. Хорошо, что Go скрывает реализацию хеш-таблицы и, следовательно, ее сложность. Подробнее о том, как создать собственную хеш-таблицу в Go, вы узнаете из главы 5.

Следующий код с использованием функции `make()` позволяет создать пустую хеш-таблицу с ключами типа `string` и значениями типа `int`:

```
iMap = make(map[string]int)
```

Кроме того, используя *литерал хеш-таблицы* следующего вида, можно создать хеш-таблицу, заполненную данными:

```
anotherMap := map[string]int {
    "k1": 12
    "k2": 13
}
```

Чтобы получить доступ к объектам `anotherMap`, к ним нужно обратиться как к `anotherMap["k1"]` и `anotherMap["k2"]`. Чтобы удалить объект хеш-таблицы, следует воспользоваться функцией `delete()`:

```
delete(anotherMap, "k1")
```

Чтобы перебрать все элементы хеш-таблицы, воспользуйтесь следующим способом:

```
for key, value := range iMap {
    fmt.Println(key, value)
}
```

Более подробно использование хеш-таблиц иллюстрирует код Go из файла `usingMaps.go`. Рассмотрим эту программу, разделив ее на три части. Первая часть содержит следующий код:

```
package main

import (
    "fmt"
)

func main() {

    iMap := make(map[string]int)
    iMap["k1"] = 12
    iMap["k2"] = 13
    fmt.Println("iMap:", iMap)

    anotherMap := map[string]int{
        "k1": 12,
        "k2": 13,
    }
```

Вторая часть `usingMaps.go` выглядит так:

```
fmt.Println("anotherMap:", anotherMap)
delete(anotherMap, "k1")
delete(anotherMap, "k1")
delete(anotherMap, "k1")
fmt.Println("anotherMap:", anotherMap)

_, ok := iMap["doesItExist"]
if ok {
    fmt.Println("Exists!")
} else {
    fmt.Println("Does NOT exist")
}
```

Здесь представлен метод, который позволяет определить, есть ли в хеш-таблице данный ключ. Это жизненно важный прием, поскольку без него невозможно узнать, содержит ли хеш-таблица требуемую информацию.



К сожалению, если попытаться получить значение несуществующего ключа хеш-таблицы, в итоге получим ноль, что не позволяет определить, был ли результат нулевым, потому что запрошенный ключ не существует или же потому, что элемент с соответствующим ключом существует и действительно равен нулю. Вот почему мы должны использовать запись `_`, ок для хеш-таблиц.

Кроме того, здесь продемонстрировано использование функции `delete()`. Многократный вызов одного и того же оператора `delete()` ни на что не влияет и не генерирует предупреждающих сообщений.

Последняя часть программы выглядит так:

```
for key, value := range iMap {
    fmt.Println(key, value)
}
```

Здесь показано, как удобно использовать ключевое слово `range` для хеш-таблиц.

Если выполнить `usingMaps.go`, то получим следующий результат:

```
$ go run usingMaps.go
iMap: map[k1:12 k2:13]
anotherMap: map[k1:12 k2:13]
anotherMap: map[k2:13]
Does NOT exist
k1 12
k2 13
```



Помните, что вы не можете и не должны делать какие-либо предположения относительно последовательности вывода на экран пар «ключ — значение», поскольку эта последовательность является абсолютно случайной.

Запись в хеш-таблицу со значением `nil`

Следующий код Go будет работать:

```
aMap := map[string]int{}
aMap["test"] = 1
```

Однако следующий код Go работать не будет, поскольку мы присвоили значение `nil` хеш-таблице, которую затем пытаемся использовать:

```
aMap := map[string]int{}
// var aMap map[string]int
aMap = nil
fmt.Println(aMap)
aMap["test"] = 1
```

Сохранив этот код в `failMap.go` и попытавшись его выполнить, вы получите следующее сообщение об ошибке:

```
$ go run failMap.go
map[]
panic: assignment to entry in nil map
...
```

Это означает, что вставить данные в хеш-таблицу со значением `nil` не получится. Однако поиск, удаление, определение длины и использование циклов `range` для хеш-таблицы со значением `nil` не приведут к краху кода.

Когда использовать хеш-таблицы

Хеш-таблицы более универсальны, чем срезы и массивы. Правда, за такую гибкость приходится платить дополнительными вычислительными затратами на реализацию хеш-таблиц Go. Однако встроенные структуры Go очень быстрые, поэтому не бойтесь использовать хеш-таблицы Go, когда это необходимо. Следует помнить, что хеш-таблицы Go очень удобны и позволяют хранить различные типы данных, при этом просты для понимания и быстро работают.

Константы Go

Go поддерживает *константы* — переменные, значения которых нельзя изменять. Константы в Go определяются с помощью ключевого слова `const`.



В общем случае константы являются глобальными переменными, поэтому, если вы обнаружите, что создаете слишком много констант в локальной области видимости, стоит переосмыслить свой подход к созданию программы.

Основным преимуществом использования констант в программах является гарантия того, что их значение не изменится во время выполнения программы.

Значение констант определяется не во время выполнения программы, а на этапе компиляции.

Внутри Go для хранения констант используются логический, строковый и числовой типы данных. Благодаря этому в Go обеспечивается большая гибкость при работе с константами.

Новые константы определяются так:

```
const HEIGHT = 200
```

Обратите внимание, что в Go имена констант не обязательно писать ПРОПИСНЫМИ БУКВАМИ; это лишь мое личное предпочтение.

Кроме того, если вы хотите объявить сразу несколько констант, например, потому, что они логически связаны между собой, можете использовать следующую запись:

```
const (  
    C1 = "C1C1C1"  
    C2 = "C2C2C2"  
    C3 = "C3C3C3"  
)
```

Обратите внимание, что результаты всех операций между константами компилятор Go также считает константами. Но если константа является частью другого выражения, это уже будет не так.

Теперь для разнообразия рассмотрим следующие три объявления переменных, которые в Go означают одно и то же:

```
s1 := "My String"  
var s2 = "My String"  
var s3 string = "My String"
```

Однако, поскольку ни в одном из них не используется ключевое слово `const`, ни одна из этих переменных не является константой. Тем не менее, вы вполне можете определить аналогичным образом две константы:

```
const s1 = "My String"  
const s2 string = "My String"
```

И `s1`, и `s2` являются константами, однако для `s2` объявлен тип (`string`), что делает это объявление более ограничивающим, чем объявление `s1`. Это связано с тем, что типизированная константа Go должна соответствовать всем строгим правилам типизированной переменной Go. Константа без типа, напротив, не обязательно должна следовать строгим правилам типизированной переменной — другими словами, ее можно более свободно использовать в смешанных выражениях. Кроме того, даже константы без типа имеют тип по умолчанию, который используется только в тех случаях, когда никакая другая информация о типе не доступна. Основное преимущество такого поведения заключается в том, что, если заранее неизвестно, как именно будет использоваться константа, то можно не использовать жесткие ограничения типизированной переменной Go.

Простым примером является определение числовой константы, такое как `const value = 123`. Поскольку вы можете использовать константу `value` во многих выражениях, то явное объявление типа значительно усложнило бы задачу. Рассмотрим следующий код Go:

```
const s1 = 123
const s2 float64 = 123

var v1 float32 = s1 * 12
var v2 float32 = s2 * 12
```

С определением `v1` у компилятора не возникнет проблем; однако код, используемый для определения `v2`, не будет компилироваться, потому что `s2` и `v2` имеют разные типы:

```
$ go run a.go
# command-line-arguments
./a.go:12:6: cannot use s2 * 12 (type float64) as type float32 in assignment
```

Общая рекомендация такова: если вы используете в программе много констант, хорошо бы объединить их в одном пакете или в структуре Go.

Генератор констант `iota`

Генератор констант `iota` применяется для объявления последовательности взаимосвязанных значений, в которых используются увеличивающиеся числа, так что не требуется четко указывать каждое из этих чисел.

Большинство понятий, связанных с ключевым словом `const`, в том числе генератор констант `iota`, проиллюстрированы в файле `constants.go`, который мы рассмотрим, разделив на четыре части.

Первый фрагмент кода `constants.go` выглядит так:

```
package main

import (
    "fmt"
)

type Digit int
type Power2 int

const PI = 3.1415926

const (
    C1 = "C1C1C1"
    C2 = "C2C2C2"
    C3 = "C3C3C3"
)
```


В этой части — два новых типа с именами `Digit` и `Power2` и четыре новые константы с именами `PI`, `C1`, `C2` и `C3`.



Ключевое слово `type` предоставляет способ определить новый именованный тип, который применяет тот же базовый тип, что и существующий тип. Это свойство языка в основном применяется для разделения типов, которые могут использовать один и тот же базовый тип данных.

Вторая часть файла `constants.go` содержит следующий код Go:

```
func main() {
    const s1 = 123
    var v1 float32 = s1 * 12
    fmt.Println(v1)
    fmt.Println(PI)
```

В этой части мы определяем еще одну константу (`s1`), которая используется в выражении (`v1`).

Третья часть программы выглядит так:

```
const (
    Zero Digit = iota
    One
    Two
    Three
    Four
)
fmt.Println(One)
fmt.Println(Two)
```

Здесь вы видите определение генератора констант `iota` на основе типа `Digit`, которое эквивалентно следующему объявлению четырех констант:

```
const (
    Zero = 0
    One = 1
    Two = 2
    Three = 3
    Four = 4
)
```

Последняя часть `constants.go` выглядит так:

```
const (
    p2_0 Power2 = 1 << iota
    -
    p2_2
    -
    p2_4
    -
)
```

```

    p2_6
)

fmt.Println("2^0:", p2_0)
fmt.Println("2^2:", p2_2)
fmt.Println("2^4:", p2_4)
fmt.Println("2^6:", p2_6)
}

```

Есть еще один генератор констант `iota`, который немного отличается от предыдущего. Во-первых, здесь показано использование символа подчеркивания в блоке `const` с генератором констант `iota`, что позволяет пропускать ненужные значения. Во-вторых, значение `iota` всегда увеличивается и может использоваться в выражениях, что и происходит в данном случае.

Теперь посмотрим, что на самом деле происходит внутри блока `const`. Для `p2_0` значение `iota` равно 0, а `p2_0` определяется как 1. Для `p2_2` значение `iota` равно 2, а `p2_2` определяется как результат выражения `1 << 2`, которое в двоичном представлении равно `00000100`. Десятичное значение `00000100` равно 4, что и является результатом и значением `p2_2`. Аналогично значение `p2_4` равно 16, а значение `p2_6` — 32.

Как видно, использование `iota` экономит время, если оно соответствует вашим потребностям.

Выполнение программы `constants.go` приводит к следующим результатам:

```

$ go run constants.go
1476
3.1415926
1
2
2^0: 1
2^2: 4
2^4: 16
2^6: 64

```

Указатели в Go

Go поддерживает *указатели* — адреса памяти, что обеспечивает повышение скорости в обмен на сложный для отладки код и неприятные ошибки. Хотите узнать об этом больше — спросите любого знакомого, кто программирует на C.

Пример использования указателей приведен в главе 2, когда речь шла о небезопасном коде и пакете `unsafe`, а также сборщике мусора Go. В данном разделе эта сложная тема раскрыта глубже. Кроме того, нативные указатели Go безопасны — при условии, что вы знаете, что делаете.

При работе с указателями, если вы хотите получить значение указателя, необходимо использовать операцию `*`, которая называется *разыменованием указателя*, и оператор `&`, чтобы получить адрес памяти переменной.



Разработчикам-любителям следует использовать указатели только тогда, когда этого требуют библиотеки, которыми они пользуются. При небрежном использовании указатели могут стать причиной ужасных и трудно-обнаруживаемых ошибок.

Чтобы функция принимала указатель в качестве параметра, нужно написать следующий код:

```
func getPointer(n *int) {
}
```

Аналогичным образом функция может возвращать указатель:

```
func returnPointer(n int) *int {
}
```

Использование безопасных указателей Go проиллюстрировано на примере программы `pointers.go`, которую мы рассмотрим, разделив на четыре части. Первый фрагмент кода `pointers.go` выглядит так:

```
package main

import (
    "fmt"
)

func getPointer(n *int) {
    *n = *n * *n
}

func returnPointer(n int) *int {
    v := n * n
    return &v
}
```

С одной стороны, хорошо, что функция `getPointer()` позволяет изменять переданную ей переменную без необходимости возвращать что-либо из функции. Это происходит потому, что указатель, переданный в качестве параметра, содержит адрес памяти этой переменной.

С другой стороны, `returnPointer()` получает целочисленный параметр и возвращает указатель на целое число, что обозначается как `return &v`. Это может показаться не очень полезным, однако вы по-настоящему оцените данную возможность в главе 4, где пойдет речь об указателях на структуры Go, а также в последующих главах, где будут рассматриваться более сложные структуры данных.

Обе функции, `getPointer()` и `returnPointer()`, вычисляют квадрат целого числа, однако используют совершенно разные подходы: `getPointer()` сохраняет результат в переданном параметре, а `returnPointer()` возвращает результат, который должен быть сохранен в другой переменной.

Вторая часть программы содержит следующий код Go:

```
func main() {
    i := -10
    j := 25

    pI := &i
    pJ := &j

    fmt.Println("pI memory:", pI)
    fmt.Println("pI memory:", pJ)
    fmt.Println("pI value:", *pI)
    fmt.Println("pI value:", *pJ)
```

Здесь `i` и `j` — обычные целочисленные переменные, а `pI` и `pJ` — указатели, ссылающиеся на `i` и `j` соответственно. `pI` — это адрес памяти, а `*pI` — значение, сохраненное в памяти по этому адресу.

Третья часть `pointers.go` выглядит так:

```
*pI = 123456
*pI--
fmt.Println("i:", i)
```

Здесь показано, как можно изменить переменную `i` с помощью указателя `pI`, который ссылается на `i`, двумя разными способами: во-первых, путем непосредственного присвоения нового значения `i`, во-вторых, с помощью оператора `--`.

Последняя часть кода `pointers.go` содержит следующий код Go:

```
getPointer(pJ)
fmt.Println("j:", j)
k := returnPointer(12)
fmt.Println(*k)
fmt.Println(k)
}
```

Здесь вызывается функция `getPointer()`, которой в качестве параметра передается `pJ`. Как мы уже говорили, любые изменения, внесенные в переменную, на которую указывает `pJ` внутри `getPointer()`, будут влиять на значение переменной `j`, что и доказывает вывод с помощью оператора `fmt.Println("j:", j)`. Вызов `returnPointer()` возвращает указатель, присвоенный переменной-указателю `k`.

Запуск `pointers.go` приводит к следующим результатам:

```
$ go run pointers.go
pI memory: 0xc0000160b0
pJ memory: 0xc0000160b8
pI value: -10
pJ value: 25
i: 123455
j: 625
144
0xc0000160f0
```

Я не удивлюсь, если у вас возникнут проблемы с пониманием кода Go из `pointers.go`: ведь мы еще не обсуждали функции и определения функций. Не поленитесь заглянуть в главу 6, где более подробно объясняются темы, связанные с функциями.



Обратите внимание, что строки в Go являются переменными, хранящими значение, а не указателями, как в C.

Зачем нужны указатели

Есть две основные причины использовать указатели в ваших программах:

- ❑ указатели позволяют обмениваться данными по ссылке, а не по значению, особенно между функциями Go;
- ❑ указатели бывают чрезвычайно полезны, если вы хотите отличить нулевое значение от неприсвоенного.

Время и дата

В этом разделе вы узнаете, как анализировать строки времени и даты в Go, как преобразовывать время и дату в различные форматы и как выводить время и дату в желаемом формате. На первый взгляд эта задача может показаться не важной, однако она может оказаться действительно критической, если требуется синхронизировать несколько задач или если приложение должно прочитать дату из одного или нескольких текстовых файлов либо получить ее непосредственно от пользователя.

Центром работы с временем и датами в Go является пакет `time`; в этом разделе будет продемонстрировано использование некоторых из его функций.

Прежде чем учиться анализировать текстовую строку и преобразовывать ее во время или дату, мы рассмотрим простую программу `usingTime.go`, которая даст нам общее представление о пакете `time`. Для изучения этой программы мы разделим ее на три части. Первая часть выглядит так:

```
package main
```

```
import (  
    "fmt"  
    "time"  
)
```

Второй фрагмент кода `usingTime.go` содержит следующий код Go:

```
func main() {  
    fmt.Println("Epoch time:", time.Now().Unix())  
    t := time.Now()
```

```

fmt.Println(t, t.Format(time.RFC3339))
fmt.Println(t.Weekday(), t.Day(), t.Month(), t.Year())

time.Sleep(time.Second)
t1 := time.Now()
fmt.Println("Time difference:", t1.Sub(t))

```

Функция `time.Now().Unix()` возвращает *время эпохи* UNIX — количество секунд, прошедших с 00:00:00 UTC 1 января 1970 года. Функция `Format()` позволяет преобразовывать переменную типа `time` в другой формат, в данном случае в формат RFC3339.

В этой книге нам много раз встретится функция `time.Sleep()`, позволяющая эмулировать задержку выполнения какой-либо другой функции. Константа `time.Second` в Go определяет интервал 1 секунда. Для того чтобы задать интервал 10 секунд, нужно умножить `time.Second` на 10. В Go есть еще несколько подобных констант, в том числе `time.Nanosecond`, `time.Microsecond`, `time.Millisecond`, `time.Minute` и `time.Hour`. Таким образом, минимальный временной интервал, который можно определить с помощью пакета `time`, — это наносекунда. Наконец, функция `time.Sub()` позволяет вычислить разность между двумя значениями времени.

Последняя часть программы выглядит так:

```

formatT := t.Format("01 January 2006")
fmt.Println(formatT)
loc, _ := time.LoadLocation("Europe/Paris")
londonTime := t.In(loc)
fmt.Println("Paris:", londonTime)
}

```

Здесь с помощью `time.Format()` определяется новый формат даты, который можно будет использовать для вывода переменной типа `time`.

Выполнение программы `usingTime.go` приведет к следующим результатам:

```

$ go run usingTime.go
Epoch time: 1548753515
2019-01-29 11:18:35.01478 +0200 EET m=+0.000339641
2019-01-29T11:18:35+02:00
Tuesday 29 January 2019
Time difference: 1.000374985s
01 January 2019
Paris: 2019-01-29 10:18:35.01478 +0100 CET

```

Теперь, когда мы познакомились с основами пакета `time`, пора углубиться в его функциональность, начиная с работы с временем.

Работа с временем

Если у нас есть переменная типа `time`, ее можно легко преобразовать в любой формат, связанный с временем или датой. Однако основная проблема возникает, когда у вас есть исходная строка и вы хотите проверить, является ли она допустимым значением

времени. Функция, используемая для синтаксического анализа строк, содержащих значения времени и даты, называется `time.Parse()` и принимает два параметра. Первый из них определяет ожидаемый формат анализируемой строки, а второй содержит саму строку, которая должна быть проанализирована. Первый параметр состоит из элементов, принадлежащих списку констант Go, связанных с анализом даты и времени.



Список констант, которые можно использовать для создания формата синтаксического анализа, находится по адресу <https://golang.org/src/time/format.go>. Go не определяет формат даты или времени в форме наподобие `DDYYYYMM` или `%D %Y %M`, как другие языки программирования, а использует собственный подход. Поначалу этот подход может показаться странным, однако впоследствии вы наверняка оцените его по достоинству, так как он предупреждает разработчика от глупых ошибок.

Константы Go для работы со временем равны `15` для часа, `04` для минут и `05` для секунд. Легко догадаться, что все эти числовые значения должны быть уникальными. Также можно использовать строки `PM` и `pm` (в верхнем или нижнем регистре) для указания на время после полудня.

Обратите внимание, что вы не обязаны использовать все доступные константы Go для работы со временем. Основная задача разработчика — разместить эти константы Go в желаемом порядке в соответствии с типом строк, которые будет обрабатывать программа. Окончательную версию строки из этих констант, которая будет передана функции `time.Parse()` в качестве первого параметра, можно рассматривать как *регулярное выражение*.

Синтаксический анализ времени

В этом разделе будет показано, как анализировать строку, которая передается в программу `parseTime.go` в качестве аргумента командной строки, чтобы преобразовать ее в переменную типа `time`. Однако это не всегда возможно, поскольку входная строка может иметь неправильный формат или содержать недопустимые символы. Мы рассмотрим утилиту `parseTime.go`, разделив ее на три части.

Первый фрагмент кода `parseTime.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "time"
)
```

Вторая часть содержит следующий код:

```
func main() {
    var myTime string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n", filepath.Base(os.Args[0]))
        os.Exit(1)
    }

    myTime = os.Args[1]
```

Последняя часть `parseTime.go`, в которой и происходит все волшебство, выглядит так:

```
    d, err := time.Parse("15:04", myTime)
    if err == nil {
        fmt.Println("Full:", d)
        fmt.Println("Time:", d.Hour(), d.Minute())
    } else {
        fmt.Println(err)
    }
}
```

Чтобы выполнить синтаксический анализ текстовой строки, содержащей часы и минуты, нужно использовать формат `"15:04"`. Значение переменной `err` говорит о том, был ли анализ успешным.

Выполнение `parseTime.go` приводит к следующим результатам:

```
$ go run parseTime.go
usage: parseTime string
exit status 1
$ go run parseTime.go 12:10
Full: 0000-01-01 12:10:00 +0000 UTC
Time: 12 10
```

Как мы видим, Go выводит строку даты и времени полностью, поскольку именно это хранится в переменной `time`. Если вас интересует только время, без даты, то нужно вывести соответствующие части переменной `time`.

Если при попытке проанализировать строку и преобразовать ее в тип `time` использовать неправильную константу Go, например `22:04`, то получим следующее сообщение об ошибке:

```
$ go run parseTime.go 12:10
parsing time "12:10" as "22:04": cannot parse ":10" as "2"
```

Но если использовать, например, константу Go `11`, предназначенную для анализа, где месяц указан как число, сообщение об ошибке будет немного другим:

```
$ go run parseTime.go 12:10
parsing time "12:10": month out of range
```


Работа с датами

В этом подразделе будет показано, как анализировать строки, обозначающие даты в Go, — для этого нам снова понадобится функция `time.Parse()`.

Константы Go для работы с датами — это `Jan` для анализа трехбуквенной аббревиатуры, используемой для описания месяца, `2006` для анализа года и `02` — для анализа дня месяца. Если вместо `Jan` использовать `January`, то, как и следовало ожидать, вместо трехбуквенного сокращения получим полное название месяца.

Кроме того, можно использовать константу `Monday` для анализа строк, которые содержат полное название дня недели, или `Mon` для сокращенного названия дня недели.

Синтаксический анализ дат

В этом подразделе мы разработаем программу Go под названием `parseDate.go`, которую рассмотрим, разделив на две части.

Первая часть `parseDate.go` содержит следующий код:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "time"
)

func main() {

    var myDate string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n", filepath.Base(os.Args[0]))
        return
    }
    myDate = os.Args[1]
```

Вторая часть `parseDate.go` содержит следующий код Go:

```
d, err := time.Parse("02 January 2006", myDate)
if err == nil {
    fmt.Println("Full:", d)
    fmt.Println("Time:", d.Day(), d.Month(), d.Year())
} else {
    fmt.Println(err)
}
}
```

Если между названием месяца и годом стоит символ -, то вместо "02 January-2006" в качестве первого параметра для `time.Parse()` можно использовать "02 January 2006".

Выполнение `parseDate.go` приведет к следующим результатам:

```
$ go run parseDate.go
usage: parseDate string
$ go run parseDate.go "20 July 2000"
Full: 2000-07-20 00:00:00 +0000 UTC
Time: 20 July 2000
```

Поскольку `parseDate.go` не ожидает данных о времени, в конец полной строки даты и времени автоматически добавляется строка `00:00:00 +0000 UTC`.

Изменение формата даты и времени

В этом разделе будет показано, как изменить формат строки, содержащей как дату, так и время. Чаще всего такие строки встречаются в файлах журналов веб-серверов, таких как Apache и Nginx. Поскольку мы пока не умеем построчно читать текстовые файлы, текст будет жестко закодирован в программе; однако это не повлияет на ее функциональность.

Мы рассмотрим код Go программы `timeDate.go`, разделив его на четыре части. Первая часть — ожидаемая преамбула:

```
package main
```

```
import (
    "fmt"
    "regexp"
    "time"
)
```

Стандартный пакет `Go regex` необходим для поддержки регулярных выражений.

Вторая часть кода `timeDate.go` выглядит следующим образом:

```
func main() {  
  
    logs := []string{"127.0.0.1 - - [16/Nov/2017:10:49:46 +0200] 325504",  
                     "127.0.0.1 - - [16/Nov/2017:10:16:41 +0200]  
                     \"GET /CVEEN HTTP/1.1\" 200 12531 \"-\"  
                     \"Mozilla/5.0 AppleWebKit/537.36\",  
                     "127.0.0.1 200 9412 - - [12/Nov/2017:06:26:05 +0200]  
                     \"GET \\\"http://www.mtsoukalos.eu/taxonomy/term/47\\\" 1507\",  
                     "[12/Nov/2017:16:27:21 +0300]", "[12/Nov/2017:20:88:21 +0200]",  
                     "[12/Nov/2017:20:21 +0200]",  
  
}
```

Поскольку мы не можем быть уверенными в данных и их формате, в этой программе представлены различные примеры данных, чтобы охватить множество различных случаев, включая неполные данные, такие как [12/Nov/2017:20:21 +0200], где в раз- деле времени нет секунд, и ошибочные данные, такие как [12/Nov/2017:20:88:21 +0200], где значение минут равно 88.

Третья часть `timeDate.go` содержит следующий код Go:

```
for _, logEntry := range logs {
    r := regexp.MustCompile(`.*\[([d\d\/\w+\/d\d\d\d:d\d:d\d\d\d:.*])\].*`)
    if r.MatchString(logEntry) {
        match := r.FindStringSubmatch(logEntry)
```

Основное преимущество, которое мы получаем от использования такого труд- ночеитаемого регулярного выражения, в данной программе заключается в том, что оно позволяет выяснить, есть где-то в исходной строке строка даты и времени или же нет.

После того как мы получим эту строку, мы передадим ее в `time.Parse()`, и пу- скай `time.Parse()` выполнит остальную часть работы.

Последняя часть программы содержит следующий код Go:

```
dt, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
if err == nil {
    newFormat := dt.Format(time.RFC850)
    fmt.Println(newFormat)
} else {
    fmt.Println("Not a valid date time format!")
}
} else {
    fmt.Println("Not a match!")
}
}
```

Как только мы найдем строку, которая соответствует регулярному выражению, мы проанализируем ее, используя `time.Parse()`, чтобы убедиться, что это допу- стимая строка даты и времени. Если это так, то `timeDate.go` выведет дату и время в *формате RFC850*.

Если выполнить `timeDate.go`, то получим следующий результат:

```
$ go run timeDate.go
Thursday, 16-Nov-17 10:49:46 EET
Thursday, 16-Nov-17 10:16:41 EET
Sunday, 12-Nov-17 06:26:05 EET
Sunday, 12-Nov-17 16:27:21 +0300
Not a valid date time format!
Not a match!
```

Измерение времени выполнения программы

В этом разделе будет показано, как измерить время выполнения одной или нескольких команд в Go. Та же методика может применяться для измерения времени выполнения функции или группы функций. В качестве примера мы рассмотрим программу `Go exesTime.go`, которую разделим на три части.



Это простой, но тем не менее очень мощный и удобный прием. Не стоит недооценивать простоту Go.

Первая часть `exesTime.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    time.Sleep(time.Second)
    duration := time.Since(start)
    fmt.Println("It took time.Sleep(1)", duration, "to finish.")
}
```

Чтобы измерить время выполнения команды, нам потребуется функциональность пакета `time`. Вся работа выполняется функцией `time.Since()`. Эта функция принимает один аргумент, который должен означать момент времени в прошлом. В данном случае будет измерено время, необходимое Go для выполнения `time.Sleep(time.Second)`, поскольку это единственный оператор, стоящий между `time.Now()` и `time.Since()`.

Вторая часть `exesTime.go` имеет следующий вид:

```
start = time.Now()
time.Sleep(2 * time.Second)
duration = time.Since(start)
fmt.Println("It took time.Sleep(2)", duration, "to finish.")
```

На этот раз мы измеряем время, необходимое Go для выполнения `time.Sleep(2 * time.Second)`. Это может быть очень полезно для определения точности функции `time.Sleep()`, точность которой главным образом определяется точностью внутренних часов Go.

Последняя часть `exesTime.go` выглядит следующим образом:

```
start = time.Now()
for i := 0; i < 200000000; i++ {
    _ = i
}
```

```

duration = time.Since(start)
fmt.Println("It took the for loop", duration, "to finish.")

sum := 0
start = time.Now()
for i := 0; i < 200000000; i++ {
    sum += i
}
duration = time.Since(start)
fmt.Println("It took the for loop", duration, "to finish.")

```

В этой последней части программы измеряется скорость выполнения двух циклов. Первый цикл не делает ничего, тогда как второй выполняет некоторые вычисления. Как мы увидим в результатах работы программы, второй цикл `for` выполняется быстрее первого.

При выполнении `execTime.go` получим следующие результаты:

```

$ go run execTime.go
It took time.Sleep(1) 1.000768881s to finish.
It took time.Sleep(2) 2.00062487s to finish.
It took the for loop 50.497931ms to finish.
It took the for loop 47.70599ms to finish.

```

Измерение скорости работы сборщика мусора Go

Теперь мы можем переписать программы `sliceGC.go`, `mapNoStar.go`, `mapStar.go` и `mapSplit.go` из предыдущей главы и получить более точные результаты без необходимости использовать утилиту командной строки UNIX `time(1)`. В сущности, единственное, что нужно сделать в каждом из этих файлов, — это вставить строки `time.Now()` и `time.Since()` с `runtime.GC()` между ними и вывести результаты. Обновленные версии `sliceGC.go`, `mapNoStar.go`, `mapStar.go` и `mapSplit.go` будут называться `sliceGCTime.go`, `mapNoStarTime.go`, `mapStarTime.go` и `mapSplitTime.go` соответственно.

Выполнение этих обновленных версий приведет к следующим результатам:

```

$ go run sliceGCTime.go
It took GC() 281.563μs to finish
$ go run mapNoStarTime.go
It took GC() 9.483966ms to finish
$ go run mapStarTime.go
It took GC() 651.704424ms to finish
$ go run mapSplitTime.go
It took GC() 12.743889ms to finish

```

Эти результаты гораздо точнее, чем предыдущие, так как они показывают только время, затраченное `runtime.GC()`, без учета времени, которое потребовалось программе для заполнения срезов и хеш-таблиц, используемых для хранения значений. Тем не менее, результаты по-прежнему подтверждают наши выводы о том, насколько медленно сборщик мусора Go работает с хеш-таблицами, в которых хранится большое количество данных.

Веб-ссылки и упражнения

- ❑ Напишите генератор константы `iota` для дней недели.
- ❑ Напишите программу Go, которая бы преобразовывала заданный массив в хеш-таблицу.
- ❑ Посетите страницу документации пакета `time` по адресу <https://golang.org/pkg/time/>.
- ❑ Сможете ли вы написать генератор константы `iota` для степеней числа четыре?

Советую также посетить страницу GitHub, где обсуждаются Go 2 и изменения числовых литералов. Это поможет вам понять, как происходят изменения в Go: <https://github.com/golang/proposal/blob/master/design/19308-number-literals.md>.

- ❑ Напишите собственную версию `parseDate.go`.
- ❑ Напишите собственную версию `parseTime.go`. Не забудьте протестировать вашу программу.
- ❑ Сможете ли вы создать версию `timeDate.go`, которая бы обрабатывала два формата даты и времени?

Резюме

В этой главе раскрыты многие интересные темы, касающиеся Go, в том числе числовые типы данных, хеш-таблицы, массивы и срезы, а также указатели Go, константы и циклы Go. Кроме того, вы узнали, как Go работает с датами и временем. Теперь вы должны понимать, чем срезы лучше массивов.

Следующая глава посвящена созданию и использованию составных типов в Go — сюда главным образом входят типы, которые создаются с помощью ключевого слова `struct` и называются *структурами*. Затем мы обсудим строковые переменные и *кортежи*.

Помимо этого в следующей главе вы узнаете о *регулярных выражениях* и *сопоставлении с образцом* — это сложные темы не только для Go, но и для других языков программирования. Однако при правильном и аккуратном использовании регулярные выражения и сопоставление с образцом настолько упрощают жизнь разработчика, что имеет смысл узнать о них больше.

JSON — это очень популярный текстовый формат, поэтому в следующей главе мы также обсудим, как создавать, импортировать и экспортировать в Go данные JSON.

Наконец, вы познакомитесь с ключевым словом `switch` и пакетом `strings`, который позволяет манипулировать строками в формате *UTF-8*.

4

Использование составных типов данных

В предыдущей главе рассмотрены многие основные темы Go, включая числовые типы данных, массивы, срезы, хеш-таблицы, указатели, константы, цикл `for`, ключевое слово `range`, а также способы работы с временем и датами.

В этой главе речь пойдет о расширенных свойствах Go, таких как *кортежи* и *строки*, стандартном пакете Go `strings` и операторе `switch`. Также, что наиболее важно, мы рассмотрим широко используемые в Go *структуры*.

Кроме того, в данной главе рассмотрены вопросы работы с текстовыми файлами форматов *JavaScript Object Notation (JSON)* и *Extensible Markup Language (XML)*, реализации простого хранилища данных типа «ключ — значение», создания *регулярных выражений* и выполнения сопоставления с образцом в Go.

В этой главе будут раскрыты следующие темы:

- ❑ структуры и ключевое слово `struct` в Go;
- ❑ кортежи в Go;
- ❑ строки, руны и строковые литералы;
- ❑ работа с текстовым форматом JSON;
- ❑ работа с текстовым форматом XML;
- ❑ регулярные выражения в Go;
- ❑ сопоставление с образцом в Go;
- ❑ оператор `switch`;
- ❑ возможности пакета `strings`;
- ❑ вычисление числа π с высокой точностью;
- ❑ разработка хранилища типа «ключ — значение».

Составные типы данных

Несмотря на то что стандартные типы Go весьма удобны, быстры и гибки, они, как правило, не охватывают все типы данных, которые вам могут потребоваться в программе Go. Go решает эту проблему путем реализации структур, которые представляют собой пользовательские типы данных, создаваемые разработчиком. Кроме того, в Go есть собственный способ поддержки кортежей, главная задача которого — позволить функциям возвращать несколько значений без необходимости группировать их в структуры, как это реализовано в C.

Структуры

Массивы, срезы и хеш-таблицы, безусловно, очень полезны, однако они не позволяют группировать и хранить в одном месте несколько значений. Если нужно создать новый удобный тип, сгруппировав в нем несколько переменных разных типов, можно использовать структуру. Элементы структуры называются *полями структуры* или просто полями.

Рассмотрим для начала простую структуру, которая уже встречалась в файле `sortSlice.go` в предыдущей главе:

```
type aStructure struct {
    person string
    height int
    weight int
}
```

По причинам, которые станут понятны в главе 6, имена полей структуры обычно начинаются с заглавной буквы — это в основном зависит от того, что вы хотите делать с этими полями. Данная структура имеет три поля с именами `person`, `height` и `weight` соответственно. Теперь можно создать новую переменную типа `aStructure`:

```
var s1 aStructure
```

Кроме того, можно получить доступ к определенному полю структуры по его имени. Так, чтобы получить значение поля `person` переменной `s1`, нужно ввести `s1.person`.

Можно также определить *литерал структуры*:

```
p1 := aStructure{"fmt", 12, -2}
```

Но, поскольку запомнить последовательность полей структуры иногда очень сложно, Go позволяет использовать для определения литерала структуры другую форму:

```
p1 := aStructure{weight: 12, height: -2}
```

В этом случае не нужно определять начальное значение для каждого поля структуры.

Теперь, когда вы усвоили основы конструкций, пора рассмотреть более практический пример. Он хранится в файле `structures.go`, и мы рассмотрим его, разбив на четыре части.

Первая часть `structures.go` содержит следующий код:

```
package main
```

```
import (
    "fmt"
)
```



Типы Go и структуры в частности, как правило, определяются вне функции `main()`, чтобы они относились к глобальной области видимости и были доступны для всего пакета Go, если только вы не хотите специально подчеркнуть, что данный тип полезен лишь в пределах текущей области видимости и его не предполагается использовать в других местах.

Второй фрагмент `structures.go` содержит следующий код Go:

```
func main() {

    type XYZ struct {
        X int
        Y int
        Z int
    }

    var s1 XYZ
    fmt.Println(s1.Y, s1.Z)
```

Как видим, ничто не мешает нам определить новую структуру внутри функции, но должна быть причина для этого.

Третья часть `structures.go` выглядит так:

```
p1 := XYZ{23, 12, -2}
p2 := XYZ{Z: 12, Y: 13}
fmt.Println(p1)
fmt.Println(p2)
```

Здесь определяются два литерала структур с именами `p1` и `p2`, которые затем выводятся на экран.

Последняя часть `structures.go` содержит следующий код Go:

```
pSlice := [4]XYZ{}
pSlice[2] = p1
pSlice[0] = p2
fmt.Println(pSlice)
p2 = XYZ{1, 2, 3}
fmt.Println(pSlice)
}
```

В этой части мы создали массив структур с именем `pSlice`. Как станет ясно из результатов работы `structures.go`, если присвоить структуру элементу массива, эта структура копируется в массив, поэтому изменение значения исходной структуры не повлияет на объекты массива.

Выполнение `structures.go` приводит к следующим результатам:

```
$ go run structures.go
0 0
{23 12 -2}
{0 13 12}
[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
```



Обратите внимание, что последовательность, в которой вы добавляли поля в определение типа структуры, имеет значение для идентификации типа этой структуры. Проще говоря, две структуры с одинаковыми полями в Go не будут считаться одинаковыми, если эти поля определены у них в разном порядке.

Как видно из результатов работы `structures.go`, нулевое значение переменной типа `struct` создается путем обнуления всех полей этой переменной в соответствии с их типами.

Указатели на структуры

В главе 3 уже говорилось об указателях. В этом разделе рассмотрим пример, связанный с указателями на структуры. Программа называется `pointerStruct.go`, и мы ее изучим, разделив на четыре части.

Первая часть программы содержит следующий код Go:

```
package main

import (
    "fmt"
)

type myStructure struct {
    Name      string
    Surname   string
    Height    int32
}
```

Второй фрагмент кода `pointerStruct.go` выглядит так:

```
func createStruct(n, s string, h int32) *myStructure {
    if h > 300 {
        h = 0
    }
```

```

    }
    return &myStructure{n, s, h}
}

```

Подход, использованный в `createStruct()` для создания новой структурной переменной, имеет много преимуществ по сравнению с самостоятельной инициализацией структурных переменных. Одно из этих преимуществ — возможность проверить, является ли предоставленная информация правильной и достоверной. Кроме того, данный подход более понятен — есть центральная точка, в которой инициализируются переменные типа `struct`. Если со структурными переменными что-то пойдет не так, вы знаете, где искать ошибку и кого винить! Обратите внимание, что некоторые предпочли бы назвать такую функцию не `createStruct()`, а `NewStruct()`.



Для тех, у кого есть опыт работы на C или C++, вполне естественно, что функция Go возвращает адрес памяти локальной переменной. Ничто не теряется, все счастливы.

Третья часть `pointerStruct.go` выглядит так:

```

func retStructure(n, s string, h int32) myStructure {
    if h > 300 {
        h = 0
    }
    return myStructure{n, s, h}
}

```

В этой части представлена функция `retStructure()` — то же самое, что `createStruct()`, но без указателя. Обе функции работают нормально, поэтому выбор между `createStruct()` и `retStructure()` — вопрос личных предпочтений. Более подходящие имена для этих двух функций могли бы быть `NewStructurePointer()` и `NewStructure()` соответственно.

Последняя часть `pointerStruct.go` содержит следующий код Go:

```

func main() {
    s1 := createStruct("Mihalis", "Tsoukalos", 123)
    s2 := retStructure("Mihalis", "Tsoukalos", 123)

    fmt.Println((*s1).Name)
    fmt.Println(s2.Name)
    fmt.Println(s1)
    fmt.Println(s2)
}

```

Если выполнить `pointerStruct.go`, то получим следующий результат:

```

$ go run pointerStruct.go
Mihalis
Mihalis

```

```
&{Mihalis Tsoukalos 123}
{Mihalis Tsoukalos 123}
```

Как вы видите, главное различие между `createStruct()` и `retStructure()` состоит в том, что первая функция возвращает указатель на структуру, следовательно, вам нужно разыменовывать этот указатель, чтобы использовать объект, на который он указывает, из-за чего код может быть несколько некрасивым, тогда как вторая функция возвращает весь объект структуры.



Структуры очень важны в Go и широко используются в реальных программах, так как позволяют группировать произвольное количество значений и обрабатывать эти значения как единое целое.

Ключевое слово `new`

Go поддерживает ключевое слово `new`, которое позволяет размещать в памяти новые объекты. Однако, используя `new`, нужно помнить очень важную деталь: `new` возвращает адрес памяти выделенного объекта. Проще говоря, `new` возвращает указатель.

Итак, можно создать переменную `aStructure` следующим образом:

```
pS := new(aStructure)
```

После того как выполнится оператор `new`, вы готовы работать с новой переменной, для которой выделена память, и эта память уже обнулена (значения переменных структуры установлены в нулевые значения), но не инициализирована.



Основное различие между `new` и `make` состоит в том, что переменные, созданные с помощью `make`, правильно инициализируются, а не только обнуляется выделенная для них память. Кроме того, `make` можно применять только к хеш-таблицам, каналам и срезам, и эта функция не возвращает адрес памяти, то есть `make` не возвращает указатель.

В следующей строке кода с помощью `new` создается срез, указывающий на `nil`¹:

```
sP := new([]aStructure)
```

¹ В этом примере команда `new([]aStructure)` выделяет в памяти место под поля описания среза (срез в Go — это структура), но так как поля описания среза не инициализированы, возвращается указатель на `nil`. Чтобы в дальнейшем использовать переменную `sP`, нужно инициализировать срез, выполнив команду `make`, например, `*sP = make([]aStructure, 10)`.

Кортежи

Строго говоря, *кортеж* — это конечный упорядоченный список, состоящий из нескольких частей. Главное в кортежах то, что Go не поддерживает тип кортежа и, следовательно, официально не обслуживает кортежи, несмотря на то что поддерживает определенные способы использования кортежей.

Интересный момент: в этой книге мы уже использовали кортежи Go, начиная с главы 1, в выражениях наподобие следующего, где функция возвращает два значения в виде одного выражения:

```
min, _ := strconv.ParseFloat(arguments[1], 64)
```

Мы рассмотрим кортежи Go на примере программы, которая называется `tuples.go`, разделив ее для удобства на три фрагмента кода. Обратите внимание, что в представленном коде используется функция, которая возвращает три значения в виде кортежа. Подробнее о функциях вы узнаете из главы 6.

Первая часть `tuples.go` выглядит так:

```
package main

import (
    "fmt"
)

func retThree(x int) (int, int, int) {
    return 2 * x, x * x, -x
}
```

Здесь вы видите реализацию функции `retThree()`, которая возвращает кортеж, содержащий три целочисленных значения. Такая возможность позволяет функции возвращать несколько значений, не требуя группировать различные возвращаемые значения в структуру и возвращать структурную переменную.

В главе 6 вы узнаете, как помещать имена в возвращаемые значения функции Go — очень удобное свойство, способное избавить вас от возможных ошибок.

Вторая часть `tuples.go` выглядит так:

```
func main() {
    fmt.Println(retThree(10))
    n1, n2, n3 := retThree(20)
    fmt.Println(n1, n2, n3)
```

Здесь функция `retThree()` используется дважды: в первый раз без сохранения возвращаемых значений, а во второй — с сохранением трех возвращаемых значений в трех разных переменных, используя один оператор, который в терминологии Go

называется *присваиванием кортежей*. Последнее иногда вводит в заблуждение, заставляя думать, что Go поддерживает кортежи.

Если вас интересуют лишь некоторые из значений, возвращаемых функцией, вы можете поставить вместо не интересующих вас значений *символ подчеркивания* (`_`). Обратите внимание, что если объявить переменную в Go и затем не использовать ее, то это приведет к ошибке компиляции.

Третья часть программы содержит следующий код Go:

```
n1, n2 = n2, n1
fmt.Println(n1, n2, n3)

x1, x2, x3 := n1*2, n1*n1, -n1
fmt.Println(x1, x2, x3)
}
```

Как видим, кортежи позволяют делать много умных вещей, таких как обмен значениями без необходимости создавать временную переменную, а также вычисление выражений.

Выполнение `tuples.go` приведет к следующим результатам:

```
$ go run tuples.go
20 100 -10
40 400 -20
400 40 -20
800 160000 -400
```

Регулярные выражения и сопоставление с образцом

Сопоставление с образцом, которое играет ключевую роль в Go, — это метод поиска в строке некоторого набора символов на основе определенного шаблона поиска, основанного на регулярных выражениях и *грамматиках*. Если сопоставление с образцом прошло успешно, то вы сможете извлечь из строки нужные данные, заменить или удалить их.

Пакет Go, отвечающий за определение регулярных выражений и сопоставление с образцом, называется `regexp`. Примеры его использования вы увидите далее в этой главе.



Используя в коде регулярные выражения, следует рассматривать определение регулярного выражения как самую важную часть соответствующего кода, поскольку именно от регулярного выражения зависит функциональность данного кода.

Немного теории

Регулярные выражения компилируются в распознаватель путем построения обобщенной диаграммы переходов, называемой *конечным автоматом*. Конечные автоматы бывают одного из двух видов: детерминированные или недетерминированные. В недетерминированных конечных автоматах у одного и того же набора входных данных может существовать несколько вариантов перехода из этого состояния. *Распознаватель* — это программа, которая принимает на входе строку x и определяет, является ли x предложением из заданного языка.

Грамматика — это набор порождающих правил, представленных на формальном языке. Порождающие правила описывают, как создавать строки из алфавита языка в соответствии с синтаксисом этого языка. Грамматика не описывает значение строки или того, что можно с ней сделать в каком-либо контексте, — она описывает лишь форму строки. Здесь важно понимать, что грамматики лежат в основе регулярных выражений, потому что без грамматики невозможно определить или использовать регулярное выражение.



Регулярные выражения позволяют решать проблемы, которые иначе было бы слишком трудно решить. Однако не пытайтесь решать с помощью регулярных выражений любые проблемы, с которыми вы сталкиваетесь. Всегда выбирайте для работы правильный инструмент.

В оставшейся части этого раздела будут представлены три примера регулярных выражений и сопоставления с образцом.

Простой пример

В этом подразделе мы узнаем, как выбрать конкретный столбец из строки текста. Чтобы все стало еще интереснее, мы также научимся построчно читать текстовые файлы. Однако о вводе-выводе из файлов вы узнаете из главы 8, поэтому, чтобы получить больше информации о соответствующем коде Go, вам следует прочитать ее.

Наш исходный файл Go называется `selectColumn.go`, и мы его рассмотрим, разделив на пять фрагментов. Для работы утилиты нужны как минимум два аргумента командной строки: первый — требуемый номер столбца, а второй — путь к текстовому файлу, который нужно обработать. Однако мы можем использовать любое количество текстовых файлов — `selectColumn.go` обработает их все по одному.

Первая часть `selectColumn.go` выглядит так:

```
package main
```

```
import (
```



```

    "bufio"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)

```

Вторая часть `selectColumn.go` содержит следующий код:

```

func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: selectColumn column <file1> [<file2> [... <fileN>]]\n")
        os.Exit(1)
    }

    temp, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Column value is not an integer:", temp)
        return
    }

    column := temp
    if column < 0 {
        fmt.Println("Invalid Column number!")
        os.Exit(1)
    }
}

```

В первую очередь программа выполняет проверку, чтобы убедиться, что она получила достаточное количество аргументов командной строки (`len(arguments) < 2`). Кроме того, нам нужны еще две проверки, чтобы убедиться, что предоставленное значение столбца является числом и что оно больше 0.

Третья часть `selectColumn.go` выглядит так:

```

for _, filename := range arguments[2:] {
    fmt.Println("\t\t", filename)
    f, err := os.Open(filename)
    if err != nil {
        fmt.Printf("error opening file %s\n", err)
        continue
    }
    defer f.Close()
}

```

Программа выполняет различные проверки, чтобы убедиться, что текстовый файл существует и его можно прочитать — для открытия текстового файла используется функция `os.Open()`. Помните, что права доступа к текстовому файлу в UNIX не всегда допускают чтение файла.

Четвертый фрагмент кода `selectColumn.go` выглядит так:

```

r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
}

```

```

if err == io.EOF {
    break
} else if err != nil {
    fmt.Printf("error reading file %s", err)
}

```

Как вы узнаете из главы 8, функция `bufio.ReadString()` считывает файл до первого появления переданного этой функции параметра. Так, `bufio.ReadString('\n')` считывает одну строку, потому что `\n` в UNIX является символом новой строки. Функция `bufio.ReadString()` возвращает *байтовый срез*.

Последний фрагмент `selectColumn.go` содержит следующий код:

```

data := strings.Fields(line)
if len(data) >= column {
    fmt.Println((data[column-1]))
}
}
}
}

```

Логика программы довольно проста: вы разбиваете текст на строки и выбираете нужный столбец. Однако, поскольку нет уверенности в том, что в текущей строке есть необходимое количество полей, нужно проверять это перед любым выводом на экран. Это самая простая форма сопоставления с образцом, потому что каждая строка разбивается на отдельные слова с использованием пробелов в качестве разделителей.

Если вас заинтересовала тема разбиения строк, то вам будет полезно узнать, что функция `strings.Fields()` разбивает строку на основе пробельных символов, определенных в функции `unicode.IsSpace()`, и возвращает срез строки.

Выполнение `selectColumn.go` приведет к результатам следующего вида:

```

$ go run selectColumn.go 15 /tmp/swtag.log /tmp/adobegc.log | head
/tmp/swtag.log
/tmp/adobegc.log

AdobeGCData
Successfully
Initializing
Stream
*****AdobeGC
Perform
Perform
Trying

```

Утилита `selectColumn.go` выводит имя каждого обработанного файла, даже если из этого файла не были получены данные для вывода.



Важно помнить, что никогда не следует доверять входным данным, особенно если они получены от нетехнических пользователей. Проще говоря, всегда проверяйте, являются ли входные данные тем, что вы ожидали получить.

Более сложный пример

В этом разделе вы узнаете, как сопоставить строку даты и времени, полученную из файла журнала веб-сервера Apache. Чтобы еще больше заинтересовать вас, я покажу, как изменить формат даты и времени файла журнала на другой формат. Напомню: для этого потребуется построчно прочитать файл журнала Apache — файл в простом текстовом формате.

Для этого создадим утилиту командной строки под названием `changeDT.go`. Для ее изучения мы разделим код утилиты на пять частей. Обратите внимание, что `changeDT.go` представляет собой улучшенную версию утилиты `timeDate.go`, о которой речь шла в главе 3, не только потому, что получает данные из внешнего файла, но также по той причине, что `changeDT.go` использует два регулярных выражения и таким образом может сопоставлять строки времени и даты, представленные в двух форматах.



Очень важный момент: не пытайтесь реализовать все возможные функции в первой же версии своих утилит. Лучше создать рабочую версию с меньшим количеством функций и затем постепенно улучшать ее.

Первый фрагмент кода `changeDT.go` выглядит так:

```
package main
```

```
import (
    "bufio"
    "fmt"
    "io"
    "os"
    "regexp"
    "strings"
    "time"
)
```

Утилита `changeDT.go` делает много интересного, и поэтому нам нужно достаточное количество пакетов.

Второй фрагмент кода `changeDT.go` выглядит так:

```
func main() {

    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one text file to process!")
        os.Exit(1)
    }

    filename := arguments[1]
    f, err := os.Open(filename)
```

```

if err != nil {
    fmt.Printf("error opening file %s", err)
    os.Exit(1)
}
defer f.Close()

notAMatch := 0
r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
    }
}

```

В этой части кода мы просто пытаемся открыть входной файл для чтения, чтобы затем читать его построчно. Переменная `notAMatch` содержит количество строк входного файла, которые не соответствуют ни одному из двух регулярных выражений программы.

Третий фрагмент кода `changedT.go` содержит следующий код Go:

```

r1 := regexp.MustCompile(`.*\[(\d\d\/\w+\/\d\d\d\d:\d\d:\d\d\d\d.*)\]`.*`)
if r1.MatchString(line) {
    match := r1.FindStringSubmatch(line)
    d1, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
    if err == nil {
        newFormat := d1.Format(time.Stamp)
        fmt.Print(strings.Replace(line, match[1], newFormat, 1))
    } else {
        notAMatch++
    }
    continue
}

```

Как видим, если первый формат даты и времени не совпадает, программа продолжает выполнение. Однако когда выполнение доходит до внутренней части блока `if`, будет выполнен оператор `continue`, то есть оставшийся код внешнего цикла `for` будет пропущен. Таким образом, строки, соответствующие первому из поддерживаемых форматов, имеют вид `21/Nov/2017:19:28:09 +0200`.

Функция `regexp.MustCompile()` похожа на функцию `regexp.Compile()`, но вызывает панику, если не может проанализировать выражение. Скобки, в которые заключено регулярное выражение, позволяют использовать найденные впоследствии совпадения. В данном случае у нас может быть только одно совпадение, которое мы получим с помощью функции `regexp.FindStringSubmatch()`.

Четвертая часть `changedT.go` выглядит так:

```

r2 := regexp.MustCompile(`.*\[(\w+\/-\d\d-\d\d:\d\d:\d\d\d\d.*)\]`.*`)
if r2.MatchString(line) {

```

```

match := r2.FindStringSubmatch(line)
d1, err := time.Parse("Jan-02-06:15:04:05 -0700", match[1])
if err == nil {
    newFormat := d1.Format(time.Stamp)
    fmt.Print(strings.Replace(line, match[1], newFormat, 1))
} else {
    notAMatch++
}
continue
}

```

Второй поддерживаемый формат времени и даты имеет вид `Jun-21-17:19:28:09 +0200`. Как вы понимаете, между этими двумя форматами не так уж много различий. Обратите внимание, что, хотя в программе используется всего два формата даты и времени, вы можете использовать столько форматов, сколько пожелаете.

Последняя часть кода `changeDT.go` содержит следующий код Go:

```

}
fmt.Println(notAMatch, "lines did not match!")
}

```

Здесь выводится на экран количество строк, которые не соответствуют ни одному из двух форматов.

Текстовый файл, который используется для тестирования `changeDT.go`, содержит следующие строки:

```

$ cat logEntries.txt
- - [21/Nov/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
- - [21/Jun/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [25/Lun/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [Jun-21-17:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [20/Nov/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [35/Nov/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362

```

Выполнение `changeDT.go` приводит к следующим результатам:

```

$ go run changeDT.go logEntries.txt
- - [Nov 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
- - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [Nov 20 20:05:34] "GET /MongoDjango.zip HTTP/1.1" 200 118362
2 lines did not match!

```

Проверка IPv4-адресов

Адрес IPv4, или просто *IP-адрес*, состоит из четырех частей. Поскольку адрес IPv4 хранится с использованием восьмиразрядных двоичных чисел, каждая часть может иметь значения от 0 (00000000 в двоичном формате) до 255 (11111111 в двоичном формате).



Формат адреса IPv6 намного сложнее, чем формат IPv4, поэтому представленная здесь программа не будет работать с адресами IPv6.

Программа, которую мы рассмотрим, называется `findIPv4.go`. Разделим ее на пять частей. Первая часть `findIPv4.go`:

```
package main
```

```
import (  
    "bufio"  
    "fmt"  
    "io"  
    "net"  
    "os"  
    "path/filepath"  
    "regexp"  
)
```

Поскольку `findIPv4.go` — довольно сложная утилита, ей нужно много стандартных пакетов Go.

Вторая часть содержит следующий код Go:

```
func findIP(input string) string {  
    partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])"  
    grammar := partIP + "\\." + partIP + "\\." + partIP + "\\." + partIP  
    matchMe := regexp.MustCompile(grammar)  
    return matchMe.FindString(input)  
}
```

В этом коде внутри функции определено регулярное выражение, которое может обнаружить IPv4-адрес. Это самая важная часть программы, потому что если вы неправильно определите регулярное выражение, то никогда не сможете распознавать адреса IPv4.

Прежде чем разъяснить регулярное выражение, что будет немного позже, важно понять, что перед определением одного или нескольких регулярных выражений необходимо знать особенности задачи, которую предстоит решить. Другими словами, если не знать, что десятичные значения адреса IPv4 не могут превышать 255, то никакие регулярные выражения не спасут!

Теперь, когда мы настроились на одну волну, рассмотрим следующие две строки кода:

```
partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])"  
grammar := partIP + "\\." + partIP + "\\." + partIP + "\\." + partIP
```

Регулярное выражение, определенное в `partIP`, соответствует каждой из четырех частей IP-адреса. Допустимый адрес IPv4 может начинаться с 25 и заканчиваться 0, 1, 2, 3, 4 или 5, потому что самое большое восьмибитное двоичное число

(25[0-5]), или же оно может начинаться с цифры 2, после которой следует 0, 1, 2, 3 или 4, и заканчиваться 0, 1, 2, 3, 4, 5, 6, 7, 8 или 9 (2[0-4][0-9]).

Либо адрес может начинаться с цифры 1, после которой следуют еще две цифры от 0 до 9 (1[0-9][0-9]). И последним вариантом будет натуральное число, состоящее из одной или двух цифр. Первой, необязательной цифрой может быть любая цифра от 1 до 9; вторая является обязательной, и это может быть цифра от 0 до 9 ([1-9]?[0-9]).

Переменная `grammar` говорит о том, что то, что мы ищем, состоит из четырех отдельных частей, каждая из которых должна соответствовать `partIP`. Переменная `grammar` соответствует полному IPv4-адресу, который мы ищем.



Поскольку `findIPv4.go` работает с регулярными выражениями для поиска IPv4-адреса в файле, программа может обрабатывать любой текстовый файл, содержащий правильные IPv4-адреса.

Наконец, если есть какие-либо особые требования, например, исключить определенные IPv4-адреса или отслеживать конкретные адреса или сети, то можно легко изменить код Go в `findIPv4.go` и добавить нужные функциональные возможности — удобная гибкость, когда разрабатываешь собственные инструменты.

Третья часть утилиты `findIPv4.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: %s logFile\n", filepath.Base(arguments[0]))
        os.Exit(1)
    }

    for _, filename := range arguments[1:] {
        f, err := os.Open(filename)
        if err != nil {
            fmt.Printf("error opening file %s\n", err)
            os.Exit(-1)
        }
        defer f.Close()
    }
}
```

Прежде всего убедитесь, что вы получили достаточное количество аргументов командной строки, проверив длину `os.Args`. Затем используйте цикл `for` для перебора всех аргументов командной строки.

Четвертая часть кода выглядит так:

```
r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
```

```
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
}
```

Как и в `selectColumn.go`, для построчного чтения входных данных используется функция `bufio.ReadString()`.

Последняя часть `findIPv4.go` содержит следующий код Go:

```
        ip := findIP(line)
        trial := net.ParseIP(ip)
        if trial.To4() == nil {
            continue
        }
        fmt.Println(ip)
    }
}
```

Для каждой строки входного текстового файла нужно вызывать функцию `findIP()`. Функция `net.ParseIP()` проверяет, имеем ли мы дело с корректным адресом IPv4, — перепроверить еще раз никогда не бывает лишним! Если вызов `net.ParseIP()` выполнен успешно, выводим найденный IPv4-адрес на экран, после чего программа переходит к следующей строке входных данных.

Выполнение `findIPv4.go` приводит к следующим результатам:

```
$ go run findIPv4.go /tmp/auth.log
116.168.10.9
192.31.20.9
10.10.16.9
10.31.160.9
192.168.68.194
```

Таким образом, среди результатов `findIPv4.go` могут оказаться одинаковые строки. Помимо этой детали, результат работы утилиты вполне понятен.

Обработка показанных выше результатов с помощью некоторых традиционных утилит командной строки UNIX помогает получить больше информации о данных:

```
$ go run findIPv4.go /tmp/auth.log.1 /tmp/auth.log | sort -rn | uniq -c | sort -rn
38 xxx.z.z.116.9
33 x.z.z.2.190
25 xx.z.z.z.1.41
20 178.132.1.18
18 x.z.z.z.63.53
17 178.z.z.z.y.9
15 103.y.y.y.xxx.179
10 213.z.y.y.194
```



```
10 ууу.zzz.110.4
9 уу.хх.65.113
```

Здесь с помощью утилит командной строки UNIX `sort(1)` и `uniq(1)` мы выяснили десять наиболее часто встречающихся адресов IPv4 из обработанных текстовых файлов. В основе этой довольно длинной команды из оболочки `bash(1)` лежит простая логика: результаты утилиты `findIPv4.go` становятся входными данными для первой команды `sort -rn`, которая сортирует их по числам в обратном порядке. Затем команда `uniq -c` удаляет появляющиеся строки, заменяя их одной строкой, перед которой указано, сколько раз данная строка встречается во входных данных. Затем результат сортируется еще раз, чтобы сначала отображались адреса IPv4 с большим числом вхождений.



Заметьте: важно понимать, что в основе функциональности `findIPv4.go` лежат регулярные выражения. Если регулярное выражение определено неправильно — либо не учитывает всех вариантов (ложноотрицательное), либо пропускает вещи, которые не должны попадать в результат (ложноположительное), то программа не будет работать правильно.

Строки

Строки в Go не являются составным типом, однако функций, которые поддерживают строки, в Go так много, что я решил более подробно описать их в этой главе.

Как отмечено в главе 3, строки в Go являются переменными, хранящими значения, а не указателями, как **строки в C**. Кроме того, Go по умолчанию поддерживает строки формата UTF-8, так что не приходится загружать специальные пакеты или предпринимать другие хитрые маневры для вывода символов Unicode. Однако существуют тонкие различия между символами, рунами и байтами, а также различия между строками и строковыми литералами, которые разъяснены далее.

Строка Go — это байтовый срез, предназначенный только для чтения, который может содержать байты любого типа и иметь произвольную длину.

Строковый литерал можно определить следующим образом:

```
const sliteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
```

Возможно, внешний вид строкового литерала вас удивит. Строковую переменную можно определить следующим образом:

```
s2 := "€£³"
```

Чтобы определить длину строковой переменной или строкового литерала, можно воспользоваться функцией `len()`.

Проиллюстрируем всевозможные стандартные операции, связанные со строками, на примере программы `strings.go`, которую рассмотрим, разделив на пять частей. Первая из них содержит следующий код Go:

```
package main
```

```
import (
    "fmt"
)
```

Вторая часть кода Go выглядит следующим образом:

```
func main() {
    const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
    fmt.Println(sLiteral)
    fmt.Printf("x: %x\n", sLiteral)

    fmt.Printf("sLiteral length: %d\n", len(sLiteral))
```

Каждая последовательность `\xAB` представляет отдельный символ `sLiteral`. В результате вызов функции `len(sLiteral)` вернет количество символов `sLiteral`. Вместо `%x` в `fmt.Printf()` будет подставлена часть `AB` последовательности `\xAB`.

Третий фрагмент кода `strings.go` содержит следующий код Go:

```
for i := 0; i < len(sLiteral); i++ {
    fmt.Printf("%x ", sLiteral[i])
}
fmt.Println()

fmt.Printf("q: %q\n", sLiteral)
fmt.Printf("+q: %+q\n", sLiteral)
fmt.Printf(" x: % x\n", sLiteral)

fmt.Printf("s: As a string: %s\n", sLiteral)
```

Как видите, здесь получен доступ к строковому литералу как к срезу. Использование `%q` в `fmt.Printf()` со строковым аргументом приведет к выводу строки в двойных кавычках, которая безопасно экранируется синтаксисом Go. Использование `%+q` в `fmt.Printf()` со строковым аргументом гарантирует вывод только символов ASCII.

Наконец, использование `%x` (обратите внимание на пробел между символами `%` и `x`) в `fmt.Printf()` приведет к появлению пробелов между выводимыми байтами. Чтобы вывести строковый литерал в виде строки, нужно вызвать `fmt.Printf()` с `%s`.

Четвертая часть кода `strings.go` выглядит так:

```
s2 := "€£³"
for x, y := range s2 {
    fmt.Printf("%#U starts at byte position %d\n", y, x)
}

fmt.Printf("s2 length: %d\n", len(s2))
```

Здесь определяем строку с именем `s2`, которая содержит три символа Unicode. Вызов `fmt.Printf()` с `%#U` приведет к выводу на экран символов в формате `U+0058`. Использование ключевого слова `range` в строке, содержащей символы Unicode, позволяет обрабатывать содержащиеся в ней символы Unicode по одному.

Результат `len(s2)` может вас немного удивить. Поскольку переменная `s2` содержит символы Unicode, то ее размер в байтах превышает количество хранящихся в ней символов.

Последняя часть `strings.go` выглядит так:

```
const s3 = "ab12AB"
fmt.Println("s3:", s3)
fmt.Printf("x: % x\n", s3)

fmt.Printf("s3 length: %d\n", len(s3))

for i := 0; i < len(s3); i++ {
    fmt.Printf("%x ", s3[i])
}
fmt.Println()
}
```

Запуск `strings.go` приведет к следующим результатам:

```
$ go run strings.go
❖B2UP5#P)❖
x: 9942325550352350299c
sLiteral length: 10
99 42 32 55 50 35 23 50 29 9c
q: "\x99B2UP5#P)\x9c"
+q: "\x99B2UP5#P)\x9c"
x: 99 42 32 55 50 35 23 50 29 9c
s: As a string: ❖B2UP5#P)❖
U+20AC '€' starts at byte position 0
U+00A3 '£' starts at byte position 3
U+00B3 '³' starts at byte position 5
s2 length: 7
s3: ab12AB
x: 61 62 31 32 41 42
s3 length: 6
61 62 31 32 41 42
```

Неудивительно, если информация, представленная в этом разделе, покажется вам сложной, особенно если вы не знакомы с представлением букв и символов в формате Unicode и UTF-8. К счастью, большинство из них в ежедневной работе как разработчика Go вам не понадобится. Едва ли вам придется использовать для вывода данных в своих программах столь примитивные команды, как `fmt.Println()` и `fmt.Printf()`. Но если вы живете за пределами Европы и США, то некоторая информация из этого раздела может оказаться весьма кстати.

Что такое руны

Руны — это значения типа `int32`, следовательно, тип Go, который используется для представления *кодových пунктов* Unicode. Кодовый пункт, или кодовая позиция Unicode, — это числовое значение, которое обычно используется для представления отдельных символов Unicode; однако он также может иметь и другие значения, например, содержать информацию о форматировании.



Внимание! Строку можно рассматривать как множество рун.

Рунный литерал — это символ, заключенный в одинарные кавычки. Рунный литерал также можно рассматривать как *рунную константу*. Внутреннее представление рунного литерала основано на кодовой позиции Unicode.

Использование рун показано на примере программы `runes.go`, которую мы разделим на три части. Первая часть `runes.go` выглядит так:

```
package main
```

```
import (
    "fmt"
)
```

Вторая часть `runes.go` содержит следующий код:

```
func main() {
    const r1 = '€'
    fmt.Println("(int32) r1:", r1)
    fmt.Printf("(HEX) r1: %x\n", r1)
    fmt.Printf("(as a String) r1: %s\n", r1)
    fmt.Printf("(as a character) r1: %c\n", r1)
}
```

Сначала мы определяем рунный литерал с именем `r1`. (Обратите внимание, что знак евро не относится к таблице символов ASCII.) Затем выводим `r1`, используя различные операторы, а потом выводим его значение в формате `int32` и шестнадцатеричное значение. После этого пытаемся вывести `r1` в виде строки. Наконец, выводим `r1` как символ, что дает тот же результат, что и в определении `r1`.

Третий, последний фрагмент кода `runes.go` содержит следующий код Go:

```
fmt.Println("A string is a collection of runes:", []byte("Mihalis"))
aString := []byte("Mihalis")
for x, y := range aString {
    fmt.Println(x, y)
    fmt.Printf("Char: %c\n", aString[x])
}
fmt.Printf("%s\n", aString)
}
```

Здесь видно, что байтовый срез представляет собой набор рун и что вывод байтового среза с помощью `fmt.Println()` может дать не те результаты, которых мы ожидали. Чтобы преобразовать руну в символ, нужно использовать в `fmt.Printf()` символ подстановки `%с`. Чтобы вывести байтовый срез в виде строки, нужно использовать в `fmt.Printf()` символ подстановки `%s`.

Выполнение `runes.go` приведет к следующим результатам:

```
$ go run runes.go
(int32) r1: 8364
(HEX) r1: 20ac
(as a String) r1: %!s(int32=8364)
(as a character) r1: €
A string is a collection of runes: [77 105 104 97 108 105 115]
0 77
Char: M
1 105
Char: i
2 104
Char: h
3 97
Char: a
4 108
Char: l
5 105
Char: i
6 115
Char: s
Mihalis
```

Наконец, самый простой способ получить сообщение об ошибке `illegal rune literal` — использовать при импорте пакета одинарные, а не двойные кавычки:

```
$ cat a.go
package main
import (
    'fmt'
)
func main() {
}
$ go run a.go
package main:
a.go:4:2: illegal rune literal
```

Пакет unicode

Стандартный пакет Go `unicode` содержит множество удобных функций. Одна из них, которая называется `unicode.IsPrint()`, помогает определить, какие части строки можно вывести с использованием рун. Этот прием проиллюстрирован в коде

Go, содержащемуся в файле `unicode.go`, который мы рассмотрим, разделив на две части. Первая часть `unicode.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "unicode"
)

func main() {
    const sL = "\x99\x00ab\x50\x00\x23\x50\x29\x9c"
```

Второй фрагмент кода `unicode.go` содержит следующий код Go:

```
    for i := 0; i < len(sL); i++ {
        if unicode.IsPrint(rune(sL[i])) {
            fmt.Printf("%c\n", sL[i])
        } else {
            fmt.Println("Not printable!")
        }
    }
}
```

Как уже отмечалось, всю грязную работу выполняет функция `unicode.IsPrint()`, которая возвращает `true`, если руну можно напечатать, и `false`, если нельзя. При желании узнать больше о символах Unicode обязательно загляните на страницу документации пакета `unicode`. Выполнение `unicode.go` приведет к следующим результатам:

```
$ go run unicode.go
Not printable!
Not printable!
a
b
P
Not printable!
#
P
)
Not printable!
```

Пакет `strings`

Стандартный пакет Go `strings` позволяет манипулировать строками UTF-8 в Go и включает в себя множество мощных функций. Большинство из них будут проиллюстрированы в файле `useStrings.go`, который рассмотрим, разделив на пять частей. Обратите внимание, что функции пакета `strings`, связанные с вводом и выводом файлов, будут продемонстрированы в главе 8.

Первая часть `useStrings.go` выглядит так:

```
package main

import (
    "fmt"
    s "strings"
    "unicode"
)

var f = fmt.Printf
```



Пакет `strings` импортируется особым образом. При использовании такого оператора импорта Go создает псевдоним для пакета. Таким образом, вместо записи `strings.FunctionName()` теперь можно написать `s.FunctionName()`, что немного короче. Обратите внимание, что вы больше не сможете вызывать функцию пакета `strings` как `strings.FunctionName()`.

Еще один полезный прием заключается в том, что если вы постоянно обращаетесь к одной и той же функции и хотите использовать более короткую запись, то можно назначить этой функции имя переменной и использовать это имя вместо функции. В данном примере этот прием применен к функции `fmt.Printf()`. Однако не следует злоупотреблять такой возможностью, потому что иначе будет трудно читать код!

Вторая часть `useStrings.go` содержит следующий код Go:

```
func main() {
    upper := s.ToUpper("Hello there!")
    f("To Upper: %s\n", upper)
    f("To Lower: %s\n", s.ToLower("Hello THERE"))
    f("%s\n", s.Title("tHis wILL be A title!"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALIS"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALi"))
}
```

В этом фрагменте кода нам встретилось множество функций, которые позволяют играть с регистром букв в строке. Кроме того, как вы можете видеть, функция `strings.EqualFold()` позволяет определить, являются ли две строки одинаковыми, несмотря на различия в регистре букв.

Третья часть кода `useStrings.go` выглядит так:

```
f("Prefix: %v\n", s.HasPrefix("Mihalis", "Mi"))
f("Prefix: %v\n", s.HasPrefix("Mihalis", "mi"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "is"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "IS"))

f("Index: %v\n", s.Index("Mihalis", "ha"))
f("Index: %v\n", s.Index("Mihalis", "Ha"))
f("Count: %v\n", s.Count("Mihalis", "i"))
```

```
f("Count: %v\n", s.Count("Mihalis", "I"))
f("Repeat: %s\n", s.Repeat("ab", 5))

f("TrimSpace: %s\n", s.TrimSpace(" \tThis is a line. \n"))
f("TrimLeft: %s", s.TrimLeft(" \tThis is a\t line. \n", "\n\t "))
f("TrimRight: %s\n", s.TrimRight(" \tThis is a\t line. \n", "\n\t "))
```

Функция `strings.Count()` подсчитывает количество случаев, когда второй параметр появляется в строке, заданной в качестве первого параметра. Функция `strings.HasPrefix()` возвращает `true`, если строка, переданная в качестве первого параметра, начинается со строки, переданной в качестве второго параметра, и `false` — в противном случае. Аналогично функция `strings.HasSuffix()` возвращает `true`, когда строка, переданная в качестве первого параметра, заканчивается строкой, переданной в качестве второго параметра, и `false` — в противном случае.

Четвертый фрагмент кода `useStrings.go` содержит следующий код Go:

```
f("Compare: %v\n", s.Compare("Mihalis", "MIHALIS"))
f("Compare: %v\n", s.Compare("Mihalis", "Mihalis"))
f("Compare: %v\n", s.Compare("MIHALIS", "MIHALIS"))

f("Fields: %v\n", s.Fields("This is a string!"))
f("Fields: %v\n", s.Fields("Thisis\na\tstring!"))

f("%s\n", s.Split("abcd efg", ""))
```

Эта часть кода содержит несколько более глубоких и оригинальных функций. Первая из них, `strings.Split()`, позволяет разбить заданную строку на части, ограниченные строкой-разделителем, — функция `strings.Split()` возвращает *строковый срез*. Использование "" в качестве второго параметра `strings.Split()` позволяет разделить строку на отдельные символы и затем обрабатывать их по одному.

Функция `strings.Compare()` сравнивает две строки лексикографически; она возвращает 0, если строки идентичны, и -1 или +1 — если нет.

Наконец, функция `strings.Fields()` разбивает строку, переданную в качестве параметра, на части, используя для разделения пробельные символы. Пробельные символы определяются в функции `unicode.IsSpace()`.



`strings.Split()` — мощная функция, которую просто необходимо изучить, потому что рано или поздно (скорее, рано) вам придется использовать ее в своих программах.

Последняя часть `useStrings.go` содержит следующий код Go:

```
f("%s\n", s.Replace("abcd efg", "", "_", -1))
f("%s\n", s.Replace("abcd efg", "", "_", 4))
f("%s\n", s.Replace("abcd efg", "", "_", 2))
```



```

lines := []string{"Line 1", "Line 2", "Line 3"}
f("Join: %s\n", s.Join(lines, "+++"))

f("SplitAfter: %s\n", s.SplitAfter("123++432++", "++"))

trimFunction := func(c rune) bool {
    return !unicode.IsLetter(c)
}
f("TrimFunc: %s\n", s.TrimFunc("123 abc ABC \t .", trimFunction))
}

```

Как и в предыдущей части `useStrings.go`, последний фрагмент кода содержит функции, которые реализуют некоторые очень интеллектуальные функциональные возможности в простой для понимания и использования форме.

Функция `strings.Replace()` принимает четыре параметра. Первый параметр — это строка, которую следует обработать. Второй параметр содержит строку, которая, если будет найдена, заменится на третий параметр функции. Последний параметр — это максимальное количество замен, которые можно выполнить. Если этот параметр меньше нуля, то количество возможных замен не ограничено.

Последние два оператора программы определяют *функцию обрезки*, которая позволяет сохранять интересные вас руны строки и использовать эту функцию в качестве второго аргумента функции `strings.TrimFunc()`.

Наконец, функция `strings.SplitAfter()` разбивает строку, переданную в качестве первого параметра, на подстроки на основе строки-разделителя, заданной в качестве второго параметра.



Полный список функций, аналогичных `unicode.IsLetter()`, вы найдете на странице документации стандартного пакета Go `unicode`.

Выполнение `useStrings.go` приведет к следующим результатам:

```

$ go run useStrings.go
To Upper: HELLO THERE!
To Lower: hello there
THis Will Be A Title!
EqualFold: true
EqualFold: false
Prefix: true
Prefix: false
Suffix: true
Suffix: false
Index: 2
Index: -1
Count: 2
Count: 0
Repeat: ababababab

```

```

TrimSpace: This is a line.
TrimLeft: This is a   line.
TrimRight:  This is a   line.
Compare: 1
Compare: 0
Compare: -1
Fields: [This is a string!]
Fields: [This is a string!]
[a b c d
e f g]
_a_b_c_d _e_f_g_
_a_b_c_d efg
_a_bcd efg
Join: Line 1+++Line 2+++Line 3
SplitAfter: [123++ 432++ ]
TrimFunc: abc ABC

```

Обратите внимание, что здесь представлены далеко не все функции пакета `strings`. Полный перечень доступных функций вы найдете на странице документации пакета `strings` по адресу <https://golang.org/pkg/strings/>.

Если вы работаете с текстом, то вам определенно необходимо изучить все подробности и функции пакета `strings`. Поэтому обязательно поэкспериментируйте со всеми этими функциями и напишите множество примеров, которые помогут вам прояснить ситуацию.

Оператор switch

Главная причина, по которой оператор `switch` представлен в этой главе, заключается в том, что в `switch` можно использовать регулярные выражения. Однако для начала рассмотрим простой блок `switch`:

```

switch asString {
case "1":
    fmt.Println("One!")
case "0":
    fmt.Println("Zero!")
default:
    fmt.Println("Do not care!")
}

```

Этот блок `switch` выполняет разные действия для строки, равной `"1"`, `"0"`, и для всех остальных вариантов (`default`).



Наличие в блоке `switch` варианта для всех оставшихся случаев считается очень хорошей практикой. Однако, поскольку последовательность вариантов в блоке `switch` имеет значение, вариант для всех остальных случаев следует ставить последним. В Go такой вариант носит имя `default`.

Однако оператор `switch` может быть гораздо более гибким и адаптируемым:

```
switch {
case number < 0:
    fmt.Println("Less than zero!")
case number > 0:
    fmt.Println("Bigger than zero!")
default:
    fmt.Println("Zero!")
}
```

В этом блоке `switch` определяется, имеем ли мы дело с положительным целым числом, отрицательным целым числом или нулем. Как видим, в ветвях оператора `switch` допускаются условия. Вскоре вы узнаете, что ветви оператора `switch` также могут содержать регулярные выражения.

Все эти и некоторые другие примеры вы найдете в файле `switch.go`, который мы рассмотрим, разделив на пять частей.

Первая часть `switch.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "regexp"
    "strconv"
)

func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Println("usage: switch number")
        os.Exit(1)
    }
```

Пакет `regexp` необходим для поддержки регулярных выражений в `switch`.

Второй фрагмент кода `switch.go` содержит следующий код Go:

```
number, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println("This value is not an integer:", number)
} else {
    switch {
    case number < 0:
        fmt.Println("Less than zero!")
    case number > 0:
        fmt.Println("Bigger than zero!")
    default:
        fmt.Println("Zero!")
    }
}
```

Третья часть `switch.go` выглядит так:

```
asString := arguments[1]
switch asString {
case "5":
    fmt.Println("Five!")
case "0":
    fmt.Println("Zero!")
default:
    fmt.Println("Do not care!")
}
```

В этом фрагменте кода мы видим, что в вариантах `switch` могут присутствовать жестко закодированные значения. Как правило, это используется тогда, когда после ключевого слова `switch` стоит имя переменной.

Четвертая часть кода `switch.go` содержит следующий код Go:

```
var negative = regexp.MustCompile(`-`)
var floatingPoint = regexp.MustCompile(`\d?\.\d`)
var email = regexp.MustCompile(`^[^@]+@[^@.]+\.[^@.]+`)

switch {
case negative.MatchString(asString):
    fmt.Println("Negative number")
case floatingPoint.MatchString(asString):
    fmt.Println("Floating point!")
case email.MatchString(asString):
    fmt.Println("It is an email!")
    fallthrough
default:
    fmt.Println("Something else!")
}
```

Здесь происходит много интересного. Во-первых, определяются три регулярных выражения с именами `negative`, `floatingPoint` и `email` соответственно. Во-вторых, все три выражения используются в блоке `switch` с помощью функции `regexp.MatchString()`, которая и выполняет сопоставление.

Наконец, встретив ключевое слово `fallthrough`, Go выполняет ветвь, которая следует за текущей, — в данном случае это ветвь `default`. Это означает, что, после того как выполнится код ветви `email.MatchString(asString)`, будет также выполнен код для ветви `default`.

Последняя часть `switch.go` выглядит так:

```
var aType error = nil
switch aType.(type) {
case nil:
    fmt.Println("It is nil interface!")
default:
    fmt.Println("Not nil interface!")
}
}
```

Как видим, этот `switch` различает типы. Подробнее о взаимодействии `switch` с *интерфейсами* Go вы читаете в главе 7.

Выполнение `switch.go` с различными входными аргументами приведет к следующим результатам:

```
$ go run switch.go
usage: switch number.
exit status 1
$ go run switch.go mike@g.com
This value is not an integer: 0
Do not care!
It is an email!
Something else!
It is nil interface!
$ go run switch.go 5
Bigger than zero!
Five!
Something else!
It is nil interface!
$ go run switch.go 0
Zero!
Zero!
Something else!
It is nil interface!
$ go run switch.go 1.2
This value is not an integer: 0
Do not care!
Floating point!
It is nil interface!
$ go run switch.go -1.5
This value is not an integer: 0
Do not care!
Negative number
It is nil interface!
```

Вычисление числа π с высокой точностью

В этом разделе вы узнаете, как с высокой точностью вычислить число π , используя стандартный пакет Go, именуемый `math/big`, и определенные в нем специализированные типы.



В этом разделе содержится самый уродливый код Go, который мне когда-либо встречался; даже код на Java, и тот выглядит лучше!

Программа вычисления числа π по формуле Белларда называется `calculatePi.go`, и мы ее рассмотрим, разделив на четыре части.

Первая часть `convertPi.go` выглядит следующим образом:

```
package main

import (
    "fmt"
    "math"
    "math/big"
    "os"
    "strconv"
)

var precision uint = 0
```

Переменная `precision` определяет желаемую точность вычислений; она сделана глобальной, чтобы быть доступной из любой точки программы.

Второй фрагмент кода `calculatePi.go` содержит следующий код Go:

```
func Pi(accuracy uint) *big.Float {
    k := 0
    pi := new(big.Float).SetPrec(precision).SetFloat64(0)
    k1k2k3 := new(big.Float).SetPrec(precision).SetFloat64(0)
    k4k5k6 := new(big.Float).SetPrec(precision).SetFloat64(0)
    temp := new(big.Float).SetPrec(precision).SetFloat64(0)
    minusOne := new(big.Float).SetPrec(precision).SetFloat64(-1)
    total := new(big.Float).SetPrec(precision).SetFloat64(0)

    two2Six := math.Pow(2, 6)
    two2SixBig := new(big.Float).SetPrec(precision).SetFloat64(two2Six)
```

Вызов функции `new(big.Float)` создает переменную `big.Float` с заданной точностью, которая определяется с помощью функции `SetPrec()`.

Третья часть `convertPi.go` содержит оставшийся код Go функции `Pi()`:

```
for {
    if k > int(accuracy) {
        break
    }
    t1 := float64(float64(1) / float64(10*k+9))
    k1 := new(big.Float).SetPrec(precision).SetFloat64(t1)
    t2 := float64(float64(64) / float64(10*k+3))
    k2 := new(big.Float).SetPrec(precision).SetFloat64(t2)
    t3 := float64(float64(32) / float64(4*k+1))
    k3 := new(big.Float).SetPrec(precision).SetFloat64(t3)
    k1k2k3.Sub(k1, k2)
    k1k2k3.Sub(k1k2k3, k3)

    t4 := float64(float64(4) / float64(10*k+5))
    k4 := new(big.Float).SetPrec(precision).SetFloat64(t4)
    t5 := float64(float64(4) / float64(10*k+7))
    k5 := new(big.Float).SetPrec(precision).SetFloat64(t5)
```

```

t6 := float64(float64(1) / float64(4*k+3))
k6 := new(big.Float).SetPrec(precision).SetFloat64(t6)
k4k5k6.Add(k4, k5)
k4k5k6.Add(k4k5k6, k6)
k4k5k6 = k4k5k6.Mul(k4k5k6, minusOne)
temp.Add(k1k2k3, k4k5k6)

k7temp := new(big.Int).Exp(big.NewInt(-1), big.NewInt(int64(k)), nil)
k8temp := new(big.Int).Exp(big.NewInt(1024), big.NewInt(int64(k)), nil)

k7 := new(big.Float).SetPrec(precision).SetFloat64(0)
k7.SetInt(k7temp)
k8 := new(big.Float).SetPrec(precision).SetFloat64(0)
k8.SetInt(k8temp)
t9 := float64(256) / float64(10*k+1)
k9 := new(big.Float).SetPrec(precision).SetFloat64(t9)
k9.Add(k9, temp)
total.Mul(k9, k7)
total.Quo(total, k8)
pi.Add(pi, total)

k = k + 1
}
pi.Quo(pi, twoSixBig)
return pi
}

```

Эта часть программы представляет собой реализацию формулы Белларда на языке Go. Недостаток пакета `math/big` состоит в том, что практически для любого вида вычислений нужна специальная функция этого пакета. По большей части так сделано для того, чтобы эти функции поддерживали точность на заданном уровне. Таким образом, без использования переменных `big.Float` и `big.Int`, а также функций `math/big` невозможно вычислить число π с требуемой точностью.

В последней части `calculatePi.go` содержится реализация функции `main()`:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one numeric argument!")
        os.Exit(1)
    }

    temp, _ := strconv.ParseUint(arguments[1], 10, 32)
    precision = uint(temp) * 3

    PI := Pi(precision)
    fmt.Println(PI)
}

```

Выполнение `convertPi.go` приведет к следующим результатам:

```
$ go run calculatePi.go
Please provide one numeric argument!
exit status 1
$ go run calculatePi.go 20
3.141592653589793258
$ go run calculatePi.go 200
3.141592653589793256960399361738762404019183156248573243493179283571046450248913467
11851178431761535428201792941629280905081393787528343561058631336354860243676804770
6489838924381929
```

Разработка на Go хранилища типа «ключ — значение»

В этом разделе вы узнаете, как разработать на Go несложную версию хранилища типа «ключ — значение», как реализовать основную функциональность такого хранилища без каких-либо дополнительных наворотов. Идея хранилища типа «ключ — значение» скромна: быстро отвечать на запросы и вообще работать максимально быстро. Это означает использование простых алгоритмов и простых структур данных.

Представленная здесь программа реализует четыре основные задачи хранилища «ключ — значение».

1. Добавление нового элемента.
2. Удаление существующего элемента из хранилища «ключ — значение» по ключу этого элемента.
3. Поиск по хранилищу значения конкретного ключа.
4. Изменение значения существующего ключа.

Эти четыре функции позволяют полностью управлять хранилищем «ключ — значение». Команды для них будут называться `ADD`, `DELETE`, `LOOKUP` и `CHANGE` соответственно. Это означает, что программа будет работать, только если получит одну из четырех команд. Кроме того, программа будет останавливаться, если ввести слово `STOP`, и выводить все содержимое хранилища «ключ — значение» при вводе команды `PRINT`.

Программа называется `keyValue.go`; рассмотрим ее, разделив на пять фрагментов кода.

Первый фрагмент кода `keyValue.go` выглядит так:

```
package main

import (
```



```

    "bufio"
    "fmt"
    "os"
    "strings"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

var DATA = make(map[string]myElement)

```

Хранилище «ключ — значение» представляет собой отдельную хеш-таблицу Go, поскольку встроенные структуры данных Go, как правило, работают быстрее. Переменная хеш-таблицы определяется как глобальная; ее ключами являются переменные типа `string`, а значениями — переменные типа `myElement`. Как видим, здесь также определена структура `myElement`.

Второй фрагмент кода `keyValue.go` выглядит так:

```

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

```

В этом фрагменте кода реализованы две функции, которые обеспечивают функциональность команд `ADD` и `DELETE`. Обратите внимание: если пользователь попытается добавить в хранилище новый элемент, не предоставив достаточного количества значений для заполнения структуры `myElement`, функция `ADD` завершится с ошибкой. Для этой конкретной программы недостающие поля структуры `myElement` будут заполнены пустыми строками. Однако при попытке добавить в хранилище уже существующий ключ значение этого ключа не изменится, а вы получите сообщение об ошибке.

Третья часть `keyValue.go` содержит следующий код:

```
func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}
```

В этом фрагменте кода Go реализованы функции, которые поддерживают функциональность команд `LOOKUP` и `CHANGE`. При попытке изменить несуществующий ключ программа добавит этот ключ в хранилище, не создавая сообщений об ошибках. В данной части программы также реализована функция `PRINT()`, которая выводит на экран все содержимое хранилища «ключ — значение».

Имена функций написаны ПРОПИСНЫМИ БУКВАМИ, поскольку эти функции действительно важны для программы.

Четвертая часть `keyValue.go` выглядит так:

```
func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        text := scanner.Text()
        text = strings.TrimSpace(text)
        tokens := strings.Fields(text)

        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        }
    }
}
```

```

case 3:
    tokens = append(tokens, "")
    tokens = append(tokens, "")
case 4:
    tokens = append(tokens, "")
}

```

В этой части `keyValue.go` происходит чтение данных, вводимых пользователем. Во-первых, цикл `for` гарантирует, что программа будет продолжать работать до тех пор, пока пользователь что-то вводит. Во-вторых, программа проверяет, чтобы срез `tokens` содержал как минимум пять элементов, хотя такое количество элементов требуется только для команды `ADD`. Таким образом, чтобы операция `ADD` была завершена и не имела пропущенных значений, потребуется ввести команду вида `ADD aKey Field1 Field2 Field3`.

Последняя часть `keyValue.go` содержит следующий код Go:

```

switch tokens[0] {
case "PRINT":
    PRINT()
case "STOP":
    return
case "DELETE":
    if !DELETE(tokens[1]) {
        fmt.Println("Delete operation failed!")
    }
case "ADD":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !ADD(tokens[1], n) {
        fmt.Println("Add operation failed!")
    }
case "LOOKUP":
    n := LOOKUP(tokens[1])
    if n != nil {
        fmt.Printf("%v\n", *n)
    }
case "CHANGE":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !CHANGE(tokens[1], n) {
        fmt.Println("Update operation failed!")
    }
default:
    fmt.Println("Unknown command - please try again!")
}
}
}

```

В этой части программы обрабатываются данные, поступающие от пользователя. Благодаря оператору `switch` программа имеет очень ясную структуру, так что вы избавлены от необходимости использовать несколько операторов `if...else`.

Выполнение `keyValue.go` приводит к следующим результатам:

```
$ go run keyValue.go
UNKNOWN
Unknown command - please try again!
ADD 123 1 2 3
ADD 234 2 3 4
ADD 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
ADD 345 3 4 5
Add operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
CHANGE 345 3 4 5
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
DELETE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
DELETE 345
Delete operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
ADD 345 3 4 5
ADD 567 -5 -6 -7
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
key: 567 value: {-5 -6 -7}
CHANGE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
key: 567 value: {-5 -6 -7}
STOP
```

Чтобы узнать, как сделать хранилище данных «ключ — значение» постоянным, вам придется подождать до главы 8.

Вы также можете улучшить программу `keyValue.go`, добавив в нее горутин и каналы. Однако добавление горутин и каналов в однопользовательское приложение не имеет практического смысла. Но если сделать так, чтобы программа `keyValue.go` могла работать в сетях с протоколом *Transmission Control Protocol/Internet Protocol (TCP/IP)*, то использование горутин и каналов позволит поддерживать несколько соединений и обслуживать нескольких пользователей.

Подробнее о горутинах и каналах вы узнаете из глав 9 и 10. Затем, в главах 12, 13, вы узнаете о том, как создавать в Go сетевые приложения.

Go и формат JSON

JSON — это очень популярный текстовый формат, обеспечивающий простой и удобный способ передачи информации между приложениями на JavaScript. Однако JSON также используется для создания файлов конфигурации приложений и для хранения данных в структурированном формате.

В пакете `encoding/json` реализованы функции `Encode()` и `Decode()`, которые позволяют преобразовать объект Go в документ JSON и наоборот. Кроме того, в пакет `encoding/json` входят функции `Marshal()` и `Unmarshal()`, работающие аналогично `Encode()` и `Decode()` и основанные на методах `Encode()` и `Decode()`. Основное различие между парой `Marshal()` и `Unmarshal()` и парой `Encode()` и `Decode()` заключается в том, что первая работает с отдельными объектами, а вторая может работать как с несколькими объектами, так и с потоками байтов.

Чтение данных из формата JSON

В этом разделе вы узнаете, как прочитать запись JSON с диска. Воспользуемся программой `readJSON.go`, код которой здесь будет представлен в трех частях.

Первая часть `readJSON.go` содержит следующий код Go:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}
```

```
type Telephone struct {
    Mobile    bool
    Number    string
}
```

В этом коде Go определены структурные переменные, в которых будут храниться данные JSON.

Вторая часть `readJSON.go` выглядит так:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Здесь определяется функция с именем `loadFromJSON()`, которая используется для декодирования данных файла JSON в соответствии со структурой данных, передаваемой функции в качестве второго аргумента. Сначала вызываем функцию `json.NewDecoder()`, чтобы создать новый декодер JSON, связанный с файлом, а затем — функцию `Decode()`, чтобы действительно декодировать содержимое файла и поместить его в нужную структуру данных.

Последняя часть `readJSON.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println(myRecord)
    } else {
        fmt.Println(err)
    }
}
```

В файле `readMe.json` содержатся следующие данные:

```
$ cat readMe.json
{
  "Name": "Mihalis",
  "Surname": "Tsoukalos",
  "Tel": [
    {"Mobile": true, "Number": "1234-567"},
    {"Mobile": true, "Number": "1234-abcd"},
    {"Mobile": false, "Number": "abcc-567"}
  ]
}
```

Выполнение `readJSON.go` приведет к следующим результатам:

```
$ go run readJSON.go readMe.json
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Сохранение данных в формате JSON

В этом подразделе вы узнаете, как записывать данные в формат JSON. Воспользуемся утилитой `writeJSON.go`, которая представлена здесь в трех частях. Она записывает данные в стандартный поток вывода (`os.Stdout`), то есть выводит данные в окно терминала.

Первая часть `writeJSON.go` выглядит следующим образом:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile    bool
    Number    string
}
```

Вторая часть `writeJSON.go` выглядит так:

```
func saveToJSON(filename *os.File, key interface{}) {
    encodeJSON := json.NewEncoder(filename)
    err := encodeJSON.Encode(key)
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
}

```

Функция `saveToJSON()` выполняет за вас всю работу: она создает переменную кодировщика JSON с именем `encodeJSON`; эта переменная связана с именем файла, в который будут помещаться данные. Вызов функции `Encode()` кодирует данные и сохраняет их в нужный файл.

Последняя часть `writeJSON.go` выглядит так:

```

func main() {
    myRecord := Record{
        Name:      "Mihalis",
        Surname:    "Tsoukalos",
        Tel: []Telephone{Telephone{Mobile: true, Number: "1234-567"},
                     Telephone{Mobile: true, Number: "1234-abcd"},
                     Telephone{Mobile: false, Number: "abcc-567"}},
    },
}
saveToJSON(os.Stdout, myRecord)
}

```

В этом коде определена структурная переменная, в которой хранятся данные, которые нужно сохранить в формате JSON с помощью функции `saveToJSON()`. Поскольку используется `os.Stdout`, то данные выводятся на экран, а не сохраняются в файл.

Выполнение `writeJSON.go` приведет к следующим результатам:

```

$ go run writeJSON.go
{"Name":"Mihalis","Surname":"Tsoukalos","Tel":[{"Mobile":true,"Number":"1234-567"}, {"Mobile":true,"Number":"1234-abcd"}, {"Mobile":false,"Number":"abcc-567"}]}

```

Использование функций `Marshal()` и `Unmarshal()`

В этом подразделе вы узнаете, как использовать методы `Marshal()` и `Unmarshal()` для реализации тех же функциональных возможностей, что были представлены в программах `readJSON.go` и `writeJSON.go`. Код Go, который иллюстрирует функции `Marshal()` и `Unmarshal()`, вы найдете в файле `mUJSON.go`. Рассмотрим его, разделив на три части.

Первая часть `mUJSON.go` выглядит так:

```

package main

import (
    "encoding/json"
    "fmt"
)

type Record struct {
    Name    string

```



```

    Surname    string
    Tel        []Telephone
}

```

```

type Telephone struct {
    Mobile    bool
    Number    string
}

```

В этой части программы определяются две структуры: `Record` и `Telephone`, которые будут использоваться для хранения данных, помещаемых в запись JSON.

Вторая часть `mUJSON.go` выглядит так:

```

func main() {
    myRecord := Record{
        Name:      "Mihalis",
        Surname:   "Tsoukalos",
        Tel: []Telephone{Telephone{Mobile: true, Number: "1234-567"},
                     Telephone{Mobile: true, Number: "1234-abcd"},
                     Telephone{Mobile: false, Number: "abcc-567"}},
    }

    rec, err := json.Marshal(&myRecord)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(rec))
}

```

В этой части мы определяем переменную `myRecord`, которая содержит нужные данные. Здесь также продемонстрировано использование функции `json.Marshal()`, которая принимает ссылку на переменную `myRecord`. Обратите внимание: `json.Marshal()` требует переменную-указатель, которая преобразуется в формат JSON.

Последняя часть `mUJSON.go` содержит следующий код:

```

var unRec Record
err1 := json.Unmarshal(rec, &unRec)
if err1 != nil {
    fmt.Println(err1)
    return
}
fmt.Println(unRec)
}

```

Функция `json.Unmarshal()` получает входные данные JSON и преобразует их в структуру Go. Как и в случае с `json.Marshal()`, `json.Unmarshal()` также требует указатель в качестве аргумента.

Выполнение `mUJSON.go` приводит к следующим результатам:

```

$ go run mUJSON.go
{"Name":"Mihalis","Surname":"Tsoukalos","Tel":[{"Mobile":true,"Number":"1234-567"},

```

```
{ "Mobile": true, "Number": "1234-abcd" }, { "Mobile": false, "Number": "abcc-567" } ] }
```

```
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}] }
```



Пакет Go `encoding/json` включает в себя два интерфейса: `Marshaler` и `Unmarshaler`. Каждый из этих интерфейсов требует реализации своего метода, называемого `MarshalJSON()` и `UnmarshalJSON()` соответственно. Если вы хотите выполнить какую-либо нестандартную JSON-маршализацию и демаршализацию, то названные интерфейсы позволят вам это сделать¹.

Синтаксический анализ данных в формате JSON

До сих пор мы изучали обработку структурированных данных JSON в заранее известном формате. Данные такого типа можно сохранять в виде структур Go с использованием методов, описанных в предыдущих подразделах.

В этом подразделе мы покажем, как читать и хранить *неструктурированные данные JSON*. Важно помнить, что неструктурированные данные JSON помещаются не в структуры, а в хеш-таблицы Go — этот процесс будет продемонстрирован на примере программы `parsingJSON.go`, которая представлена здесь в четырех частях.

Первая часть `parsingJSON.go` выглядит так:

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
)
```

В этой части просто импортируются необходимые пакеты Go.

Вторая часть `parsingJSON.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]
```

¹ Функция `json.Marshal()` возвращает срез байт `[]byte`, который после записи обычно становится не нужен и впоследствии удаляется сборщиком мусора. Если ваша программа ориентирована на массовую обработку и запись JSON, то постоянное выделение и освобождение `[]byte` создает огромную нагрузку на сборщик мусора. Для снижения этой нагрузки лучше применять `json.NewEncoder().Encode()`, который задействует стандартный пакет `sync.Pool` с целью повторного использования срезов байт `[]byte` для маршализации JSON.

Этот код считывает переданные программе аргументы командной строки и получает первый из них — имя файла JSON, который следует прочитать.

Третья часть `parsingJSON.go` выглядит так:

```
fileData, err := ioutil.ReadFile(filename)
if err != nil {
    fmt.Println(err)
    return
}

var parsedData map[string]interface{}
json.Unmarshal([]byte(fileData), &parsedData)
```

Функция `ioutil.ReadFile()` позволяет прочитать сразу весь файл, что и нужно.

В этой части программы также определена хеш-таблица с именем `parsedData`, в которой будет сохранено разобранное содержимое прочитанного файла JSON. Каждый ключ хеш-таблицы является строкой и соответствует отдельному свойству JSON. Значение каждого ключа имеет тип `interface{}`, который может быть любым, то есть значением ключа хеш-таблицы может быть в том числе другая хеш-таблица.

Функция `json.Unmarshal()` используется для того, чтобы поместить содержимое файла в хеш-таблицу `parsedData`.



В главе 7 вы узнаете больше об интерфейсах вообще и `interface{}` в частности, а также о хеш-таблицах, которые позволяют динамически исследовать тип произвольного объекта и получать информацию о его структуре.

Последняя часть `parsingJSON.go` содержит следующий код:

```
for key, value := range parsedData {
    fmt.Println("key:", key, "value:", value)
}
}
```

Как видно из этого кода, можно перебирать ключи хеш-таблицы, чтобы получить ее содержимое. Однако интерпретация этого содержимого — совсем другое дело, так как это зависит от структуры данных, которая нам пока неизвестна.

Файл JSON с примерами данных, которые используются в этом подразделе, называется `noStr.json` и имеет такое содержимое:

```
$ cat noStr.json
{
    "Name": "John",
```

```

    "Surname": "Doe",
    "Age": 25,
    "Parents": [
        "Jim",
        "Mary"
    ],
    "Tel": [
        {"Mobile": true, "Number": "1234-567"},
        {"Mobile": true, "Number": "1234-abcd"},
        {"Mobile": false, "Number": "abcc-567"}
    ]
}

```

Выполнение `parsingJSON.go` приведет к следующим результатам:

```

$ go run parsingJSON.go noStr.json
key: Tel value: [map[Mobile:true Number:1234-567] map[Mobile:true
Number:1234-abcd] map[Mobile:false Number:abcc-567]]
key: Name value: John
key: Surname value: Doe
key: Age value: 25
key: Parents value: [Jim Mary]

```

Обратите внимание: как видно из результатов работы программы, ключи хеш-таблицы выводятся в случайном порядке.

Go и XML

Go поддерживает *XML* — язык разметки, похожий на HTML, но с гораздо более широкими возможностями.

Разработанная нами утилита `rwXML.go` считывает с диска запись JSON, вносит в нее изменения, преобразует ее в формат XML и выводит на экран. Затем данные XML преобразуются в формат JSON. Рассмотрим код Go этой утилиты, разделив его на четыре части.

Первая часть `rwXML.go` выглядит так:

```

package main

import (
    "encoding/json"
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name    string

```

```

    Surname    string
    Tel        []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}

```

Вторая часть `rwXML.go` содержит такой код Go:

```

func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}

```

Третья часть `rwXML.go` содержит следующий код Go:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println("JSON:", myRecord)
    } else {
        fmt.Println(err)
    }

    myRecord.Name = "Dimitris"

    xmlData, _ := xml.MarshalIndent(myRecord, "", "  ")
    xmlData = []byte(xml.Header + string(xmlData))
    fmt.Println("\nxmlData:", string(xmlData))
}

```

После прочтения входного файла и преобразования его содержимого в формат JSON эти данные помещены в структуру Go. Затем мы внесли изменения

в содержимое этой структуры (`myRecord`). После этого с помощью функции `MarshalIndent()` эти данные преобразованы в формат XML, добавлен заголовок с помощью `xml.Header`.

Функция `MarshalIndent()`, которую, кроме прочего, можно использовать для данных в формате JSON, работает аналогично `Marshal()`, но каждый элемент XML начинается с новой строки и имеет отступ с учетом глубины вложенности. Это связано не со значениями XML, а с представлением данных в формате XML.

Последняя часть `rwXML.go` выглядит так:

```
data := &Record{}
err = xml.Unmarshal(xmlData, data)
if nil != err {
    fmt.Println("Unmarshalling from XML", err)
    return
}

result, err := json.Marshal(data)
if nil != err {
    fmt.Println("Error marshalling to JSON", err)
    return
}

_ = json.Unmarshal([]byte(result), &myRecord)
fmt.Println("\nJSON:", myRecord)
}
```

В этой части программы данные XML преобразуются в формат JSON с помощью функций `Marshal()` и `Unmarshal()`, а затем выводятся на экран.

Выполнение `rwXML.go` приведет к следующим результатам:

```
$ go run rwXML.go readMe.json
JSON: {Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false
abcc-567}]}
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
  <Name>Dimitris</Name>
  <Surname>Tsoukalos</Surname>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-567</Number>
  </Tel>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-abcd</Number>
  </Tel>
  <Tel>
    <Mobile>false</Mobile>
    <Number>abcc-567</Number>
  </Tel>
</Record>
JSON: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Чтение XML-файла

В этом подразделе вы узнаете, как прочитать XML-файл с диска и сохранить его содержимое в структуре Go. Программа, которая здесь рассмотрена, называется `readXML.go`. Разделим ее на три части.

Первая часть `readXML.go` выглядит так:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

Вторая часть `readXML.go` содержит следующий код:

```
func loadFromXML(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeXML := xml.NewDecoder(in)
    err = decodeXML.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Представленный процесс очень похож на чтение с диска файла JSON.

Последняя часть `readXML.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
```

```

        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromXML(filename, &myRecord)

    if err == nil {
        fmt.Println("XML:", myRecord)
    } else {
        fmt.Println(err)
    }
}

```

Выполнение `readXML.go` приведет к следующим результатам:

```

$ go run readXML.go data.xml
XML: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}

```

Файл `data.xml` содержит следующие данные:

```

$ cat data.xml
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
  <Name>Dimitris</Name>
  <Surname>Tsoukalos</Surname>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-567</Number>
  </Tel>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-abcd</Number>
  </Tel>
  <Tel>
    <Mobile>false</Mobile>
    <Number>abcc-567</Number>
  </Tel>
</Record>

```

Настройка вывода данных в формате XML

В этом подразделе вы узнаете, как изменить и настроить вид данных, выводимых в формате XML. Рассматриваемая утилита называется `modXML.go`, она представлена в трех частях. Обратите внимание, что данные, которые будут преобразованы в XML, жестко закодированы в программе для простоты.

Первая часть `modXML.go` выглядит следующим образом:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

func main() {
    type Address struct {
        City, Country string
    }
    type Employee struct {
        XMLName    xml.Name `xml:"employee"`
        Id          int      `xml:"id,attr"`
        FirstName   string   `xml:"name>first"`
        LastName    string   `xml:"name>last"`
        Initials    string   `xml:"name>initials"`
        Height      float32  `xml:"height,omitempty"`
        Address     Address
        Comment     string   `xml:":comment"`
    }
}
```

Здесь определена структура XML-данных, а также дополнительная информация, касающаяся имени и типа элементов XML. В поле `XMLName` хранится имя записи XML, в данном случае это `employee`.

Поле с тегом `":comment"` является комментарием и форматируется при выводе данных соответствующим образом. Поле с тегом `":attr"` появляется на выходе как атрибут для предоставленного имени поля (в данном случае это `id`). Нотация `"name>first"` указывает Go на то, что нужно встроить тег `first` в тег `name`.

Наконец, поле с параметром `"omitempty"` при выводе пропускается, если оно пустое. Пустое значение — это `0`, `false`, нулевой указатель или интерфейс, а также любой массив, срез, хеш-таблица или строка нулевой длины.

Вторая часть `modXML.go` выглядит так:

```
r := &Employee{Id: 7, FirstName: "Mihalis", LastName: "Tsoukalos",
    Initials: "MIT"}
r.Comment = "Technical Writer + DevOps"
r.Address = Address{"Somewhere 12", "12312, Greece"}
```

Здесь определяется и инициализируется структура `employee`.

Последняя часть `modXML.go` содержит следующий код Go:

```
output, err := xml.MarshalIndent(r, " ", " ")
if err != nil {
    fmt.Println("Error:", err)
}
output = []byte(xml.Header + string(output))
```

```

os.Stdout.Write(output)
os.Stdout.Write([]byte("\n"))
}

```

Выполнение `modXML.go` приведет к следующим результатам:

```

$ go run modXML.go
<?xml version="1.0" encoding="UTF-8"?>
  <employee id="7">
    <name>
      <first>Mihalis</first>
      <last>Tsoukalos</last>
      <initials>MIT</initials>
    </name>
    <City>Somewhere 12</City>
    <Country>12312, Greece</Country>
    <!--Technical Writer + DevOps-->
  </employee>

```

Go и формат YAML

YAML (Ain't Markup Language) — это еще один очень популярный текстовый формат. Стандартная библиотека Go не поддерживает формат *YAML*, но есть пакет Go (<https://github.com/go-yaml/yaml>), который реализует поддержку *YAML* в Go.



Формат *YAML* поддерживается пакетом *Viper*, о котором речь пойдет в главе 8. Если вы хотите больше узнать о том, как *Viper* выполняет синтаксический анализ *YAML*-файлов, обратитесь к Go-коду *Viper*, размещенному по адресу <https://github.com/spf13/viper>.

Дополнительные ресурсы

Советую вам ознакомиться со следующими ресурсами:

- ❑ прочитайте документацию стандартного пакета Go `regexp`, которую вы найдете по адресу <https://golang.org/pkg/regexp/>;
- ❑ посетите главную страницу утилиты `grep(1)` и узнайте, как она поддерживает регулярные выражения;
- ❑ более подробную информацию о Go-пакете `math/big` вы найдете по адресу <https://golang.org/pkg/math/big/>;
- ❑ больше информации о *YAML* вы найдете на сайте <https://yaml.org/>;
- ❑ возможно, вам будет интересно изучить тип `sync.Map`, описанный по адресу <https://golang.org/pkg/sync/>;

- ❑ советую ознакомиться с документацией стандартного Go-пакета `unicode` по адресу <https://golang.org/pkg/unicode/>;
- ❑ возможно, поначалу это покажется трудным, но начните читать спецификацию языка программирования Go, размещенную по адресу <https://golang.org/ref/spec>.

Упражнения

- ❑ Попробуйте написать программу Go, которая выводит недопустимые части адреса IPv4.
- ❑ Можете ли вы сказать, чем отличается `make` от `new`, не заглядывая в текст главы?
- ❑ Используя код `findIPv4.go`, напишите программу Go, которая бы выводила самые популярные адреса IPv4, найденные в файле журнала, без обработки вывода какими-либо утилитами UNIX.
- ❑ Разработайте программу Go, которая бы находила в файле журнала адреса IPv4, вызвавшие сообщение об ошибке 404 HTML.
- ❑ Прочитайте файл JSON, в котором содержится десять целочисленных значений, сохраните его в переменной `struct`, увеличьте каждое целое значение на единицу и запишите измененный JSON на диск. Теперь сделайте то же самое для файла XML.
- ❑ Разработайте программу Go, которая бы находила в файле журнала все адреса IPv4, с которых были загружены ZIP-файлы.
- ❑ Используя стандартный Go-пакет `math/big`, напишите программу Go, которая бы вычисляла квадратные корни с высокой точностью — алгоритм выберите на свой вкус.
- ❑ Напишите утилиту Go, которая бы находила во входных данных дату и время, представленные в заданном формате, и возвращала бы только время.
- ❑ Помните ли вы, в чем различие между символом, байтом и руной?
- ❑ Разработайте утилиту Go для сопоставления целых чисел в диапазоне от 200 до 400 с использованием регулярного выражения.
- ❑ Попробуйте улучшить программу `keyValue.go`, добавив в нее журналирование.

Резюме

В этой главе раскрыты многие удобные функции Go, в том числе создание и использование структур, кортежей, строк и рун, а также функциональности стандартного Go-пакета `unicode`. Кроме того, вы узнали о сопоставлении с образцом, регулярных выражениях, обработке файлов JSON и XML, операторе `switch` и стандартном Go-пакете `strings`.

Наконец, вы разработали на Go хранилище типа «ключ — значение» и узнали, как использовать типы из пакета `math/big` для вычисления числа π с желаемой точностью.

В следующей главе будет показано, как группировать данные и манипулировать ими, используя более сложные механизмы, такие как двоичные деревья, связные списки, двунаправленные связные списки, очереди, стеки и пользовательские хеш-таблицы. Мы также рассмотрим структуры, которые реализованы в стандартном Go-пакете `container`; вы узнаете, как выполнять в Go матричные операции и как проверять на корректность головоломки судоку. Последней темой следующей главы будут случайные числа и генерирование трудноугадываемых строк, которые затем можно использовать в качестве безопасных паролей.

5

Как улучшить код Go с помощью структур данных

В предыдущей главе мы обсуждали составные типы данных, которые создаются с помощью ключевого слова `struct`, обработку в Go данных формата JSON и XML, а также такие темы, как регулярные выражения, сопоставление с образцом, кортежи, руны, строки, стандартные Go-пакеты `unicode` и `strings`. Наконец, мы написали на Go простое хранилище типа «ключ — значение».

Однако бывает так, что структуры, предлагаемые языком программирования, не соответствуют конкретной проблеме. В таких случаях приходится явным, специализированным способом создавать собственные структуры данных для хранения, поиска и получения данных.

Поэтому данная глава посвящена разработке и использованию в Go многих известных структур данных, в том числе *двоичных деревьев*, *связных списков*, *пользовательских хеш-таблиц*, *стеков* и *очередей*, а также исследованию их преимуществ. Поскольку никакое описание структуры данных не сравнится с наглядным изображением, в этой главе вам встретится много поясняющих рисунков.

В последних частях главы мы поговорим о проверке головоломок судoku и выполнении матричных вычислений с использованием срезов.

В этой главе рассмотрены следующие темы:

- ❑ *графы* и *узлы*;
- ❑ измерение сложности алгоритмов;
- ❑ двоичные деревья;
- ❑ пользовательские хеш-таблицы;
- ❑ связные списки;
- ❑ двунаправленные связные списки;
- ❑ работа с очередями в Go;
- ❑ стеки;
- ❑ структуры данных, предлагаемые стандартным Go-пакетом `container`;
- ❑ выполнение матричных вычислений;
- ❑ работа с головоломками судoku;

- ❑ генерация случайных чисел в Go;
- ❑ создание случайных строк, которые можно использовать в качестве паролей, трудно поддающихся взлому.

О графах и узлах

Граф ($G(V, E)$) — это конечное непустое множество вершин (или узлов) V и множество ребер E . Существует два основных типа графов: *циклические* и *ациклические*. Циклический граф — это граф, в котором все или некоторые вершины образуют замкнутый контур. В ациклических графах замкнутых контуров нет.

Направленный граф — это граф, с ребрами которого связано некое направление, а *направленный ациклический граф* — это направленный граф без замкнутых контуров.



Поскольку узел может содержать любую информацию, узлы обычно реализуются на основе структур Go благодаря их универсальности.

Сложность алгоритма

Эффективность алгоритма оценивается по его вычислительной сложности, которая главным образом определяется тем, сколько раз алгоритму для выполнения его работы требуется доступ к входным данным. Для описания сложности алгоритма в информатике используется *нотация «большое O»*. Так, алгоритм со сложностью $O(n)$, которому требуется лишь однократный доступ к входным данным, считается лучше, чем алгоритм со сложностью $O(n^2)$. Последний, в свою очередь, лучше, чем алгоритм $O(n^3)$, и т. д. Однако наихудшими являются алгоритмы с временем работы $O(n!)$. Это делает их практически непригодными для случаев, когда объем входных данных насчитывает более чем 300 элементов.

Наконец, большинство операций поиска в Go для встроенных типов, таких как поиск значения по ключу в хеш-таблице или доступ к элементу массива, имеют постоянное время, которое обозначается как $O(1)$. Это означает, что встроенные типы данных работают быстрее, чем созданные пользователем. Таким образом, как правило, лучше использовать именно встроенные типы, если только вы не хотите полностью контролировать то, что происходит внутри программы.

Кроме того, не все структуры данных по своей природе одинаковы. Как правило, операции с массивами выполняются быстрее, чем операции с хеш-таблицами, — такова цена, которую приходится платить за универсальность хеш-таблиц.



Несмотря на то что у каждого алгоритма есть свои недостатки, если только речь не идет о большом количестве данных, сложность алгоритма не особенно важна — лишь бы он точно выполнял свою работу.

Двоичные деревья в Go

Двоичное дерево — это структура данных, в которой под каждым узлом располагается **не более** двух других узлов. Это означает, что узел может быть связан с одним или максимум двумя другими узлами. Первый узел дерева называется *корнем дерева*. *Глубина дерева*, которую также называют *высотой дерева*, определяется как самый длинный путь от корня до узла, тогда как *глубина узла* — это количество ребер, соединяющих узел с корнем дерева. Узел дерева без дочерних узлов называется *листом*.

Дерево является *сбалансированным*, если наибольшая длина от корневого узла до листа не более чем на единицу превышает самую короткую длину. Иначе дерево является *несбалансированным*. Балансировка дерева может быть сложной и медленной операцией, поэтому лучше сохранять сбалансированность дерева с самого начала, а не пытаться сбалансировать уже созданное дерево, особенно если оно состоит из большого числа узлов.

На рис. 5.1 показано несбалансированное двоичное дерево. Его корневой узел — J, а узлы A, G, W и D являются листьями.

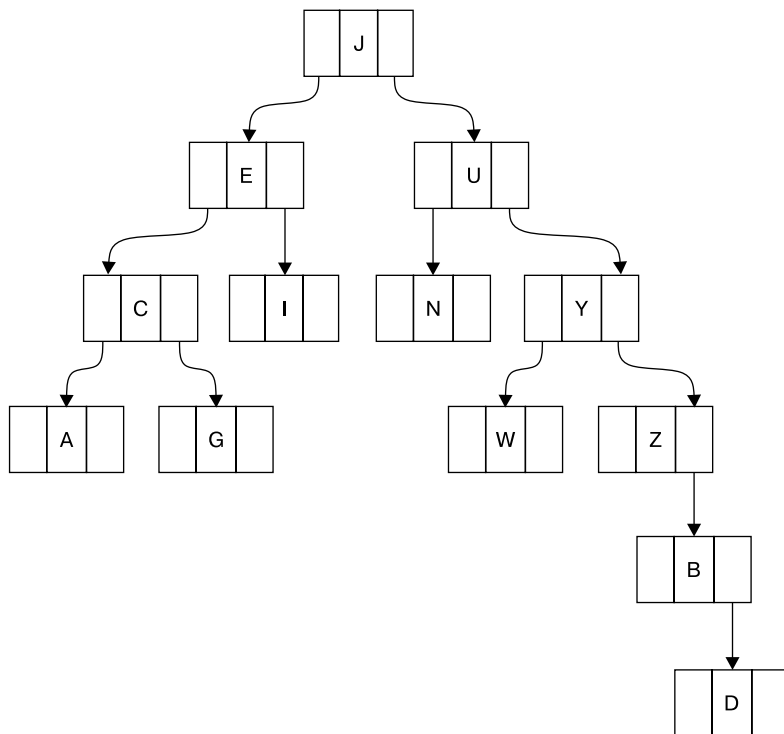


Рис. 5.1. Несбалансированное двоичное дерево

Реализация двоичного дерева в Go

В этом разделе показано, как реализовать двоичное дерево в Go. В качестве примера мы рассмотрим исходный код, который находится в файле `binTree.go`. Разделим `binTree.go` на пять частей. Первая часть программы выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type Tree struct {
    Left   *Tree
    Value  int
    Right  *Tree
}
```

Здесь мы видим определение узла дерева с использованием структуры Go. Поскольку у нас нет реальных данных, мы заполняем дерево случайными числами, для чего используем пакет `math/rand`.

Вторая часть `binTree.go` содержит следующий код Go:

```
func traverse(t *Tree) {
    if t == nil {
        return
    }
    traverse(t.Left)
    fmt.Print(t.Value, " ")
    traverse(t.Right)
}
```

На примере функции `traverse()` вы видите, как можно пройти по всем узлам двоичного дерева, используя рекурсию.

Третий фрагмент кода выглядит следующим образом:

```
func create(n int) *Tree {
    var t *Tree
    rand.Seed(time.Now().Unix())
    for i := 0; i < 2*n; i++ {
        temp := rand.Intn(n * 2)
        t = insert(t, temp)
    }
    return t
}
```

Функция `create()` используется только для заполнения двоичного дерева случайными целыми числами.

Четвертая часть программы выглядит так:

```
func insert(t *Tree, v int) *Tree {
    if t == nil {
        return &Tree{nil, v, nil}
    }
    if v == t.Value {
        return t
    }
    if v < t.Value {
        t.Left = insert(t.Left, v)
        return t
    }
    t.Right = insert(t.Right, v)
    return t
}
```

Функция `insert()` выполняет много важного¹, используя операторы `if`. Первый оператор `if` проверяет, является ли дерево пустым. Если дерево действительно пустое, то новый узел будет корневым и он будет создан как `&Tree{nil, v, nil}`.

Второй оператор `if` определяет, существует ли значение, которое мы пытаемся вставить, в обрабатываемом узле двоичного дерева. Если такое значение в узле дерева уже существует, то функция возвращает указатель на обрабатываемый узел и завершается, ничего не делая.

Третий оператор `if` определяет, будет ли значение, которое мы пытаемся вставить, размещаться слева или справа от проверяемого в данный момент узла, и действует соответственно. Обратите внимание, что в представленной реализации создается несбалансированное двоичное дерево.

Последняя часть `binTree.go` содержит следующий код Go:

```
func main() {
    tree := create(10)
    fmt.Println("The value of the root of the tree is", tree.Value)
    traverse(tree)
    fmt.Println()
    tree = insert(tree, -10)
    tree = insert(tree, -2)
    traverse(tree)
    fmt.Println()
    fmt.Println("The value of the root of the tree is",
        tree.Value)
}
```

¹ Нужно отметить, что функция `insert()` является рекурсивной, поэтому `&Tree{nil, v, nil}` создает не только корневой узел дерева, но и все листья, которые потом встраиваются в родительский узел операторами присваивания `t.Left = insert(t.Left, v)` и `t.Right = insert(t.Right, v)`.

Выполнение `binTree.go` приведет к следующему результату:

```
$ go run binTree.go
The value of the root of the tree is 18
0 3 4 5 7 8 9 10 11 14 16 17 18 19
-10 -2 0 3 4 5 7 8 9 10 11 14 16 17 18 19
The value of the root of the tree is 18
```

Преимущества двоичных деревьев

Нет ничего лучше дерева, если нужно представить иерархические данные. По этой причине деревья широко используются компиляторами языка программирования для анализа компьютерных программ.

Кроме того, деревья имеют **упорядоченную** структуру, так что не приходится прилагать усилий для их упорядочения; достаточно лишь размещать элементы в нужных местах, и дерево останется упорядоченным. Однако вследствие способа построения деревьев удаление элементов из дерева не всегда является простой операцией.

Если двоичное дерево сбалансировано, то операции поиска, вставки и удаления в нем выполняются за $\log(n)$ шагов, где n — количество элементов, которые содержит дерево. Кроме того, высота сбалансированного двоичного дерева приблизительно равна $\log_2(n)$. Другими словами, высота сбалансированного дерева из 10 000 элементов примерно равна 14, что на удивление мало.

Аналогично высота сбалансированного дерева из 100 000 элементов составляет примерно 17, а сбалансированного дерева из 1 000 000 элементов — около 20. Таким образом, значительное увеличение количества элементов в сбалансированном двоичном дереве не вызывает сильный рост высоты дерева. Иначе говоря, мы можем достичь любого узла дерева из 1 000 000 узлов менее чем за 20 шагов!

Основным недостатком двоичных деревьев является то, что вид дерева зависит от последовательности вставки в него элементов. Если у дерева длинные и сложные ключи, то вставка или поиск элемента могут оказаться медленными из-за большого количества требуемых операций сравнения. Наконец, если дерево не сбалансировано, то скорость выполняемых в нем операций будет непредсказуемой.



Связный список или массив создается быстрее, чем двоичное дерево, однако гибкость, которую обеспечивает двоичное дерево в операциях поиска, возможно, стоит дополнительных затрат и обслуживания.

При поиске элемента в двоичном дереве мы каждый раз проверяем, является ли значение искомого элемента больше или меньше значения текущего узла, и используем это решение, чтобы выбрать, по какой части дерева перейти вниз. Это экономит много времени.

Пользовательские хеш-таблицы в Go

Хеш-таблица — это структура данных, в которой хранится одна или несколько пар «ключ — значение» и используется *хеш-функция* для вычисления индекса в массиве блоков, или слотов памяти, в котором хранится нужное значение. В идеале хеш-функция должна назначать каждому ключу уникальный блок — при наличии необходимого количества блоков (их обычно достаточно).

Хорошая хеш-функция должна обеспечивать равномерное распределение значений хеш-функции, потому что неэффективно иметь неиспользуемые блоки или большие различия в размере блоков. Кроме того, хеш-функция должна работать предсказуемо и выдавать одинаковое хеш-значение для идентичных ключей (рис. 5.2). Иначе было бы невозможно найти нужную информацию.

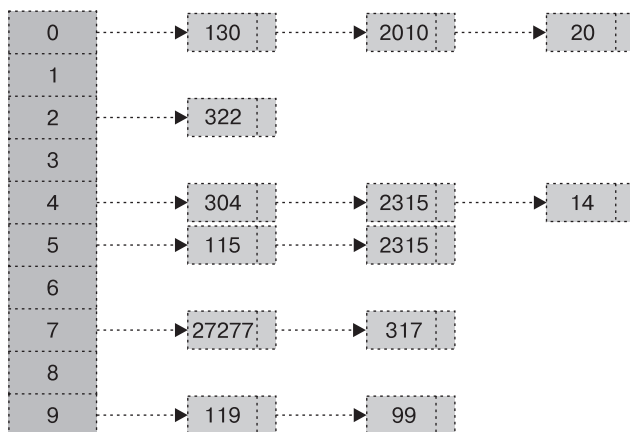


Рис. 5.2. Хеш-таблица из десяти блоков

Реализация пользовательской хеш-таблицы в Go

Go-код `hashTable.go`, который мы разделим на пять частей, поможет понять многое о пользовательских хеш-таблицах.

Первая часть `hashTable.go` выглядит следующим образом:

```
package main

import (
    "fmt"
)

const SIZE = 15
```

```
type Node struct {
    Value int
    Next *Node
}
```

Здесь мы видим определение узла пользовательской хеш-таблицы, который, как и ожидалось, устанавливается с помощью структуры Go. Именованная константа `SIZE` выявляет количество блоков пользовательской хеш-таблицы.

Второй фрагмент `hashTable.go` содержит следующий код Go:

```
type HashTable struct {
    Table map[int]*Node
    Size int
}

func hashFunction(i, size int) int {
    return (i % size)
}
```

В этом фрагменте кода мы видим реализацию хеш-функции, используемой для конкретного хеша. В функции `hashFunction()` используется *оператор деления по модулю*. Основная причина выбора оператора деления по модулю (взятие остатка) заключается в том, что данная пользовательская хеш-таблица должна обрабатывать целочисленные значения. Если бы мы имели дело со строками или числами с плавающей точкой, то мы бы использовали в хеш-функции другую логику.

Физически хеш хранится в структуре `HashTable`, которая состоит из двух полей. Первое поле — это хеш-таблица Go (`map`), которая связывает целое число со связным списком (`*Node`), а второе — размер пользовательской хеш-таблицы. В результате пользовательская хеш-таблица будет иметь столько связных списков, сколько в ней блоков. Это также означает, что узлы каждого блока такой хеш-таблицы будут храниться в связных списках. Со связными списками вы вскоре познакомитесь ближе.

Третья часть кода `hashTable.go` выглядит так:

```
func insert(hash *HashTable, value int) int {
    index := hashFunction(value, hash.Size)
    element := Node{Value: value, Next: hash.Table[index]}
    hash.Table[index] = &element
    return index
}
```

Функция `insert()` вызывается для вставки элементов в пользовательскую хеш-таблицу. Обратите внимание, что в данной реализации функции `insert()` не проверяется наличие дублирующихся значений.

Четвертая часть `hashTable.go` выглядит так:

```
func traverse(hash *HashTable) {
    for k := range hash.Table {
```

```

        if hash.Table[k] != nil {
            t := hash.Table[k]
            for t != nil {
                fmt.Printf("%d -> ", t.Value)
                t = t.Next
            }
            fmt.Println()
        }
    }
}

```

Функция `traverse()` используется для вывода всех значений пользовательской хеш-таблицы. Эта функция перебирает все связанные списки данной хеш-таблицы по одному и выводит хранящиеся в них значения.

Последняя часть кода `hashTable.go` выглядит так:

```

func main() {
    table := make(map[int]*Node, SIZE)
    hash := &HashTable{Table: table, Size: SIZE}
    fmt.Println("Number of spaces:", hash.Size)
    for i := 0; i < 120; i++ {
        insert(hash, i)
    }
    traverse(hash)
}

```

В этой части кода создается новая пользовательская хеш-таблица с именем `hash`, для чего используется переменная `table`. Эта переменная является хеш-таблицей Go, в которой хранятся блоки пользовательской хеш-таблицы. Как вы уже знаете, блоки пользовательской хеш-таблицы реализованы с использованием связанных списков.

Основная причина, по которой для хранения связанных списков хеш-таблицы использована хеш-таблица Go, а не срез или массив, заключается в том, что ключи среза или массива могут быть только положительными целыми числами, а ключи хеш-таблицы Go могут быть практически любыми.

Выполнение `hashTable.go` приведет к следующим результатам:

```

$ go run hashTable.go
Number of spaces: 15
105 -> 90 -> 75 -> 60 -> 45 -> 30 -> 15 -> 0 ->
110 -> 95 -> 80 -> 65 -> 50 -> 35 -> 20 -> 5 ->
114 -> 99 -> 84 -> 69 -> 54 -> 39 -> 24 -> 9 ->
118 -> 103 -> 88 -> 73 -> 58 -> 43 -> 28 -> 13 ->
119 -> 104 -> 89 -> 74 -> 59 -> 44 -> 29 -> 14 ->
108 -> 93 -> 78 -> 63 -> 48 -> 33 -> 18 -> 3 ->
112 -> 97 -> 82 -> 67 -> 52 -> 37 -> 22 -> 7 ->
113 -> 98 -> 83 -> 68 -> 53 -> 38 -> 23 -> 8 ->
116 -> 101 -> 86 -> 71 -> 56 -> 41 -> 26 -> 11 ->
106 -> 91 -> 76 -> 61 -> 46 -> 31 -> 16 -> 1 ->

```

```

107 -> 92 -> 77 -> 62 -> 47 -> 32 -> 17 -> 2 ->
109 -> 94 -> 79 -> 64 -> 49 -> 34 -> 19 -> 4 ->
117 -> 102 -> 87 -> 72 -> 57 -> 42 -> 27 -> 12 ->
111 -> 96 -> 81 -> 66 -> 51 -> 36 -> 21 -> 6 ->
115 -> 100 -> 85 -> 70 -> 55 -> 40 -> 25 -> 10 ->

```

Данная хеш-таблица идеально сбалансирована, поскольку она имеет дело с непрерывными числами, которые помещаются в блок в соответствии с результатами оператора деления по модулю. В реальных задачах таких удобных результатов может и не быть!



Остаток от евклидова деления натурального числа a на натуральное число b вычисляется по формуле $a = bq + r$, где q — частное, а r — остаток. Допустимые значения остатка находятся в пределах от 0 до $b - 1$. Это и является возможным результатом оператора деления по модулю (взятия остатка).

Обратите внимание, что если выполнить `hashTable.go` несколько раз, то, скорее всего, получим результаты, в которых строки будут располагаться в разной последовательности, поскольку способ вывода пар «ключ — значение» для хеш-таблицы Go является преднамеренно абсолютно случайным, поэтому на него нельзя полагаться.

Реализация функции поиска

В этом разделе мы рассмотрим реализацию функции `lookup()`, которая позволяет определить, существует ли данный элемент в пользовательской хеш-таблице. Код функции `lookup()` основан на `traverse()`:

```

func lookup(hash *HashTable, value int) bool {
    index := hashFunction(value, hash.Size)
    if hash.Table[index] != nil {
        t := hash.Table[index]
        for t != nil {
            if t.Value == value {
                return true
            }
            t = t.Next
        }
    }
    return false
}

```

Этот код находится в исходном файле `hashTableLookup.go`. Выполнение `hashTableLookup.go` приведет к следующим результатам:

```

$ go run hashTableLookup.go
120 is not in the hash table!

```

```

121 is not in the hash table!
122 is not in the hash table!
123 is not in the hash table!
124 is not in the hash table!

```

Как видите, функция `lookup()` хорошо справляется со своей работой.

Преимущества пользовательских хеш-таблиц

Если вам показалось, что пользовательские хеш-таблицы не особенно полезны, удобны или интеллектуальны, то подумайте о следующем: если пользовательская хеш-таблица имеет n ключей и k блоков, то, в отличие от линейного поиска со скоростью $O(n)$, скорость поиска для n ключей в такой пользовательской хеш-таблице равна $O(n/k)$. Это улучшение может показаться незначительным, однако для массива хешей, состоящего всего из 20 блоков, время поиска сократится в 20 раз! Это делает пользовательские хеш-таблицы идеальными для таких приложений, как словари, и аналогичных, где приходится искать большие объемы данных.

Связные списки в Go

Связный список — это структура данных, состоящая из конечного множества элементов. Здесь каждый элемент состоит как минимум из двух областей памяти: одна используется для хранения фактических данных, а вторая — для хранения указателя, который связывает текущий элемент со следующим, создавая таким образом последовательность элементов, образующих связный список.

Первый элемент связного списка называется *головным* (head), а последний элемент — *хвостовым* (tail). Первое, что следует сделать при определении связного списка, — это сохранить его головной элемент в отдельной переменной, потому что головной элемент — это единственное, что нужно для доступа ко всему связному списку. Обратите внимание, что если потерять указатель на первый узел односвязного списка, то найти его снова будет невозможно. На рис. 5.3 представлен связный список из пяти узлов.

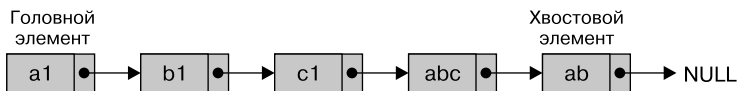


Рис. 5.3. Связный список из пяти узлов

На рис. 5.4 показано, как удалить узел из связного списка, что позволит вам лучше понять этапы данного процесса. Главное, что нужно сделать, — это расположить

указатель на узел, находящийся слева от удаляемого узла, чтобы он ссылался на узел справа от удаляемого узла.

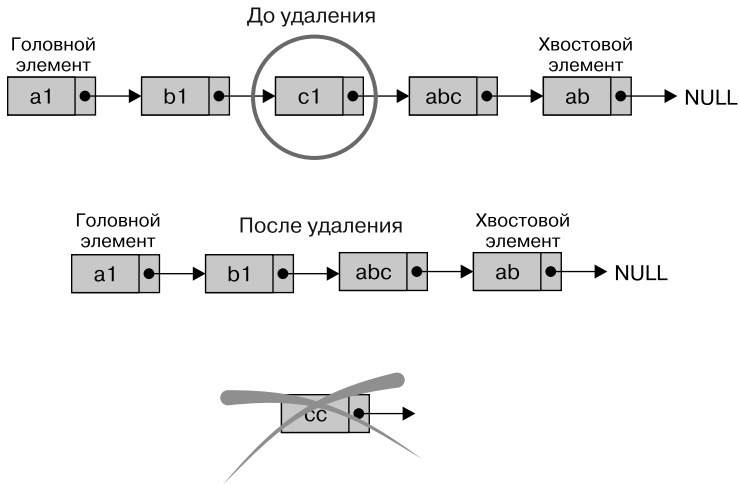


Рис. 5.4. Удаление узла из связанного списка

Приведенная далее реализация связанного списка является относительно простой и не включает в себя функцию удаления узла — реализация этого функционала оставлена вам в качестве упражнения.

Реализация связанного списка в Go

Исходный файл Go, в котором реализован связный список, называется `connectedList.go`. Рассмотрим его, разделив на пять частей.

Первый фрагмент кода файла `LinkedList.go` выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next *Node
}

var root = new(Node)
```


В этой части программы определяется структурный тип `Node`, который будет использоваться для узлов связанного списка, а также глобальная переменная `root` — она содержит первый элемент связанного списка, который будет доступен везде в коде. Учтите, что использование глобальных переменных, вообще говоря, приемлемо разве что для небольших программ и примеров кода — в больших программах это может привести к ошибкам.

Вторая часть файла `linkedList.go` содержит следующий код Go:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        t.Next = &Node{v, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

Вследствие способа работы связанных списков в таких списках обычно не содержатся повторяющиеся записи. Более того, новые узлы обычно добавляются в конец связанного списка, если связанный список не отсортирован. Именно таким образом функция `addNode()` используется для добавления новых узлов в связанный список.

В этой реализации предусмотрены три отдельных варианта, которые рассматриваются с использованием операторов `if`. В первом случае проверяем, имеем ли мы дело с пустым связанным списком. Во втором — существует ли уже в списке значение, которое мы хотим добавить. В третьем случае проверяем, достигли ли мы конца связанного списка, и тогда добавляем новый узел в конец списка с желаемым значением, используя код `t.Next = &Node{v, nil}`. Если ни одно из этих условий не выполняется, то мы повторяем этот процесс рекурсивным вызовом функции `addNode()` для следующего узла связанного списка, используя код `return addNode(t.Next, v)`.

Третий фрагмент кода программы `connectedList.go` содержит реализацию функции `traverse()`:

```
func traverse(t *Node) {
    if t == nil {
```

```

        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

```

Четвертая часть `linkedList.go` содержит следующий код:

```

func lookupNode(t *Node, v int) bool {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}

func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    i := 0
    for t != nil {
        i++
        t = t.Next
    }
    return i
}

```

В этой части вы видите реализацию двух очень удобных функций: `lookupNode()` и `size()`. Первая из них проверяет, существует ли данный элемент в связанном списке, а вторая возвращает размер связанного списка — количество узлов в этом списке.

Логика реализации функции `lookupNode()` проста и понятна: мы перебираем по очереди все элементы односвязного списка, пока не найдем нужное значение. Если мы дойдем до конца связного списка, не найдя этого значения, то поймем, что в данном списке это значение не содержится.

Последняя часть файла `connectedList.go` содержит реализацию функции `main()`:

```
func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, -1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 45)
    addNode(root, 5)
    addNode(root, 5)
    traverse(root)
    addNode(root, 100)
    traverse(root)

    if lookupNode(root, 100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }

    if lookupNode(root, -100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }
}
```

Выполнение `linkedList.go` приведет к следующим результатам:

```
$ go run linkedList.go
&{0 <nil>}
-> Empty list!
1 -> -1 ->
Node already exists: 5
Node already exists: 5
1 -> -1 -> 10 -> 5 -> 45 ->
1 -> -1 -> 10 -> 5 -> 45 -> 100 ->
Node exists!
Node does not exist!
```

Преимущества связанных списков

Самым большим преимуществом связанных списков является то, что они просты для понимания и реализации и достаточно универсальны, чтобы их можно было использовать в разных ситуациях. Связные списки применяются для моделирования различных типов данных, начиная с отдельных значений и заканчивая сложными структурами со многими полями. Кроме того, в связанных списках очень быстро выполняется последовательный поиск с помощью указателей.

Связные списки помогают не только сортировать данные, но и сохранять порядок данных даже после вставки или удаления элементов. Удаление узла из упорядоченного связанного списка выполняется так же, как и в неупорядоченном связанном списке; однако вставка нового узла в упорядоченный связанный список отличается тем, что новый узел следует поместить в нужное место, чтобы список оставался упорядоченным. На практике это означает, что если у вас есть много данных и вы знаете, что нужно будет все время удалять данные, то лучше использовать связанный список, а не пользовательскую хеш-таблицу или двоичное дерево.

Наконец, в *упорядоченных связанных списках* можно применять различные методы оптимизации при поиске или вставке узла. Самым распространенным из этих методов является сохранение указателя на центральный узел упорядоченного связанного списка, чтобы начинать поиск именно с данного указателя. Эта простая оптимизация позволяет сократить время операции поиска в два раза!

Двусвязные списки в Go

Двусвязный (двунаправленный связный) список — это список, в котором каждый узел хранит указатель на предыдущий и на следующий элементы списка. На рис. 5.5 представлен двунаправленный связный список.

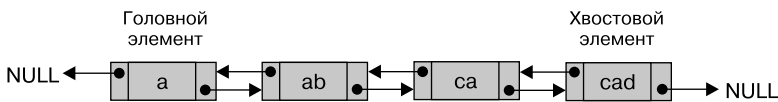


Рис. 5.5. Двунаправленный связный список

Таким образом, в двусвязном списке ссылка первого узла на следующий узел указывает на второй узел, а его ссылка на предыдущий элемент указывает на `nil` (константа, или `NULL`). Аналогично ссылка последнего узла на следующий элемент указывает на `nil`, а его ссылка на предыдущий элемент — на предпоследний узел двусвязного списка.

На рис. 5.6 показано добавление узла в двусвязный список. Как вы понимаете, в данном случае основная задача — это работа с указателями трех узлов: нового узла, узла, расположенного слева от нового узла, и узла, находящегося справа от нового узла.

Таким образом, в действительности основное различие между односвязным и двусвязным списком состоит в том, что последний требует больше операций по обслуживанию. Такова цена, которую приходится платить за возможность доступа к двусвязному списку в обоих направлениях.

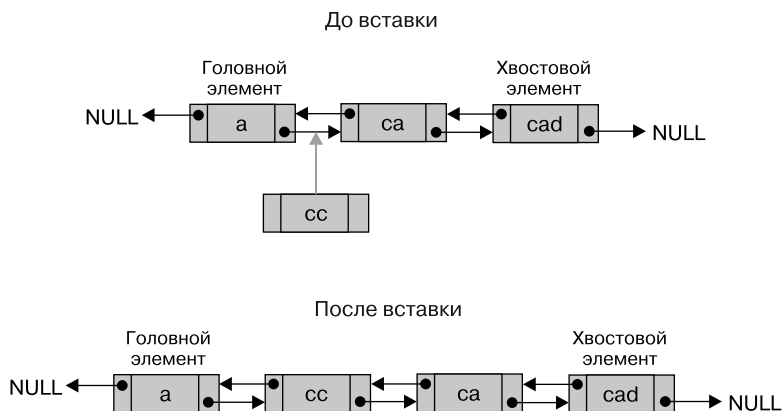


Рис. 5.6. Вставка нового узла в середину двусвязного списка

Реализация двусвязного списка в Go

Программа, реализующая двусвязный список на Go, называется `doublyllyList.go`. Рассмотрим ее, разделив на пять частей. Общая идея двусвязного списка такая же, как и у односвязного, но здесь из-за наличия двух указателей в каждом узле списка придется выполнять больше операций по обслуживанию списка.

Первая часть `dublyllyList.go` выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value      int
    Previous   *Node
    Next       *Node
}
```

В данной части мы видим определение узла двусвязного списка как структуры Go. Однако на этот раз по понятным причинам структура имеет два поля указателей.

Вторая часть файла `dplyList.go` содержит следующий код Go:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        temp := t
        t.Next = &Node{v, temp, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

Как и в случае односвязного списка, каждый новый узел помещается в конец текущего двусвязного списка. Однако так поступать не обязательно, если вы хотите построить упорядоченный двусвязный список.

Третья часть `doublyLList.go` выглядит так:

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

func reverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    temp := t
    for t != nil {
```

```

    temp = t
    t = t.Next
}

for temp.Previous != nil {
    fmt.Printf("%d -> ", temp.Value)
    temp = temp.Previous
}
fmt.Printf("%d -> ", temp.Value)
fmt.Println()
}

```

Здесь вы видите код Go для функций `traverse()` и `reverse()`. Реализация функции `traverse()` такая же, как и в программе `connectedList.go`. Однако логика функции `reverse()` очень интересна. Поскольку мы не храним указатель на конец двусвязного списка, то нам нужно перейти к концу такого списка, чтобы получить доступ к его узлам в обратном порядке.

Обратите внимание, что Go позволяет писать такой код, как `a, b = b, a`, чтобы поменять местами значения двух переменных, не создавая временную переменную.

Четвертая часть `dublyLList.go` содержит следующий код Go:

```

func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    n := 0
    for t != nil {
        n++
        t = t.Next
    }
    return n
}

func lookupNode(t *Node, v int) bool {
    if root == nil {
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }
    return lookupNode(t.Next, v)
}

```

Последний фрагмент файла `dublyLList.go` содержит следующий код Go:

```
var root = new(Node)

func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, 1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 0)
    addNode(root, 0)
    traverse(root)
    addNode(root, 100)
    fmt.Println("Size:", size(root))
    traverse(root)
    reverse(root)
}
```

Если выполнить `dublyLList.go`, то получим следующий результат:

```
$ go run doublyLList.go
&{0 <nil> <nil>}
-> Empty list!
Node already exists: 1
1 ->
Node already exists: 0
1 -> 10 -> 5 -> 0 ->
Size: 5
1 -> 10 -> 5 -> 0 -> 100 ->
100 -> 0 -> 5 -> 10 -> 1 ->
```

Как видите, функция `reverse()` работает просто отлично!

Преимущества двусвязных списков

Двунаправленные связные списки более универсальны, чем односвязные, поскольку по ним можно перемещаться в любом направлении, в них легче вставлять и удалять элементы. Кроме того, даже если вы потеряете указатель на заголовок двусвязного списка, то все равно сможете найти головной узел такого списка. Однако за эту универсальность приходится платить: поддерживать два указателя для каждого узла. Разработчик должен сам решить, насколько оправданна такая дополнительная сложность. Например, в вашем музыкальном проигрывателе двусвязный список может использоваться для возможности перехода к предыдущей и к следующей композиции.

Очереди в Go

Очередь — это особый вид связного списка, в котором элементы вставляются в начало, а удаляются из хвоста. Нам не понадобится рисунок для описания очереди: представьте себе, что вы идете в банк и ждете, пока те, кто пришел раньше вас, решат свои вопросы, прежде чем вы сможете пообщаться с кассиром.

Основным преимуществом очередей является простота: для доступа к очереди нужны всего две функции. А чем меньше функций, тем меньше вам придется беспокоиться о том, что что-нибудь пойдет не так. Вы можете реализовать очередь любым удобным способом, лишь бы обеспечить поддержку этих двух функций.

Реализация очереди в Go

Программа, на примере которой мы проиллюстрируем реализацию очереди в Go, называется `queue.go`. Разделим ее на пять частей. Обратите внимание, что для реализации очереди используется связный список. Функции `Push()` и `Pop()` предназначены для вставки и удаления узлов из очереди соответственно.

Первая часть кода `queue.go` выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next *Node
}

var size = 0
var queue = new(Node)
```

Наличие переменной `size` для хранения количества узлов в очереди удобно, но не обязательно. В представленной здесь реализации это сделано для того, чтобы упростить работу. Вероятно, вы тоже захотите создать эти поля в своей структуре.

Вторая часть `queue.go` содержит следующий код Go:

```
func Push(t *Node, v int) bool {
    if queue == nil {
        queue = &Node{v, nil}
        size++
        return true
    }

    t = &Node{v, nil}
```

```

    t.Next = queue
    queue = t
    size++

    return true
}

```

Здесь показана реализация функции `Push()`, которая сама по себе очень проста. Если очередь пуста, то новый узел становится очередью. Если очередь не пуста, то создается новый узел, который помещается перед текущей очередью. После этого только что созданный узел становится заголовком очереди.

Третья часть `queue.go` содержит следующий код Go:

```

func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        queue = nil
        size--
        return t.Value, true
    }

    temp := t
    for (t.Next) != nil {
        temp = t
        t = t.Next
    }

    v := (temp.Next).Value
    temp.Next = nil

    size--
    return v, true
}

```

В этом коде показана реализация функции `Pop()`, которая удаляет из очереди самый старый элемент. Если очередь пуста (`size == 0`), то из нее ничего не извлекается.

Если в очереди есть только один узел, то извлекается значение этого узла, после чего очередь становится пустой. В противном случае извлекается последний элемент очереди, удаляется последний узел и изменяются необходимые указатели, после чего возвращается нужное значение.

Четвертая часть `queue.go` содержит следующий код Go:

```

func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Queue!")
    }
}

```

```

        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

```

Функция `traverse()` не обязательна для работы с очередью, но она удобна для просмотра всех узлов очереди.

Последний фрагмент кода `queue.go` содержит следующий код Go:

```

func main() {
    queue = nil
    Push(queue, 10)
    fmt.Println("Size:", size)
    traverse(queue)

    v, b := Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    for i := 0; i < 5; i++ {
        Push(queue, i)
    }
    traverse(queue)
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)
    traverse(queue)
}

```

Почти весь код Go в функции `main()` предназначен для проверки работы очереди. Самая важная часть этого кода — два оператора `if`, которые позволяют узнать, вернула ли функция `Pop()` значение, или очередь была пуста и функция не вернула ничего.

Выполнение `queue.go` приведет к следующим результатам:

```
$ go run queue.go
Size: 1
10 ->
Pop: 10
Size: 0
4 -> 3 -> 2 -> 1 -> 0 ->
Size: 5
Pop: 0
Size: 4
Pop: 1
Size: 3
4 -> 3 -> 2 ->
```

Стеки в Go

Стек — это структура данных, которая похожа на стопку тарелок: когда вам понадобится новая тарелка, вы возьмете верхнюю.

Как и в случае с очередью, основным преимуществом стека является его простота, поскольку, чтобы работать со стеком, нужно реализовать всего две функции: добавление в стек нового узла и удаление узла из стека.

Реализация стека в Go

Пришло время рассмотреть реализацию стека в Go. Мы покажем это на примере программы `stack.go`. Как и в случае с очередью, для реализации стека будет использоваться связный список. Как вы знаете, нам понадобятся две функции: одна, с именем `Push()`, для вставки в стек, и вторая, с именем `Pop()`, — для удаления из стека.

Хотя в этом нет необходимости, однако полезно хранить количество элементов в стеке в отдельной переменной, чтобы можно было определить, является ли стек пустым, без необходимости обращаться к самому связному списку.

Мы рассмотрим исходный код `stack.go`, разделив его на четыре части. Первая часть выглядит так:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
```

```

    Next *Node
}

var size = 0
var stack = new(Node)

```

Во второй части `stack.go` содержится реализация функции `Push()`:

```

func Push(v int) bool {
    if stack == nil {
        stack = &Node{v, nil}
        size = 1
        return true
    }

    temp := &Node{v, nil}
    temp.Next = stack
    stack = temp
    size++
    return true
}

```

Если стек не пустой, то функция создает новый узел (`temp`), который помещается перед текущим стеком. Затем новый узел становится головным узлом стека. Данная версия функции `Push()` всегда возвращает значение `true`, но если занимаемое стеком пространство ограничено, то можно изменить код так, чтобы функция возвращала `false` при попытке превысить емкость стека.

В третьей части содержится реализация функции `Pop()`:

```

func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        size = 0
        stack = nil
        return t.Value, true
    }

    stack = stack.Next
    size--
    return t.Value, true
}

```

Четвертый фрагмент кода `stack.go` выглядит так:

```

func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Stack!")
        return
    }
}

```

```

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

```

Поскольку стек реализован с использованием связанного списка, для него можно реализовать функцию `traverse()`.

Последняя часть `stack.go` содержит следующий код Go:

```

func main() {
    stack = nil
    v, b := Pop(stack)
    if b {
        fmt.Print(v, " ")
    } else {
        fmt.Println("Pop() failed!")
    }

    Push(100)
    traverse(stack)
    Push(200)
    traverse(stack)

    for i := 0; i < 10; i++ {
        Push(i)
    }

    for i := 0; i < 15; i++ {
        v, b := Pop(stack)
        if b {
            fmt.Print(v, " ")
        } else {
            break
        }
    }
    fmt.Println()
    traverse(stack)
}

```

Как видно, исходный код `stack.go` немного короче, чем код Go для `queue.go`, — прежде всего потому, что принцип стека проще, чем у очереди.

Выполнение `stack.go` приведет к следующим результатам:

```

$ go run stack.go
Pop() failed!
100 ->
200 -> 100 ->
9 8 7 6 5 4 3 2 1 0 200 100
Empty Stack!

```



Внимание! Теперь вы знаете, как можно использовать связный список для реализации пользовательской хеш-таблицы, очереди и стека. Эти примеры должны помочь вам осознать полезность и важность связных списков в программировании в частности и в информатике в целом.

Пакет container

В этом разделе я расскажу об использовании стандартного пакета Go `container`. Пакет `container` поддерживает три структуры данных: *кучу*, *список* и *кольцо*. Эти структуры данных реализованы в пакетах `container/heap`, `container/list` и `container/ring` соответственно.

На случай, если вы не знаете: кольцо представляет собой *циклический список*, так что последний элемент кольца указывает на его первый элемент. По сути, это означает, что все узлы кольца эквивалентны, кольцо не имеет ни начала, ни конца. В результате можно пройти через все кольцо, начиная с любого элемента.

В следующих трех подразделах описаны все пакеты, входящие в состав `container`. Совет: если функциональность стандартного Go-пакета `container` соответствует вашим потребностям, используйте его; в противном случае лучше реализовать и использовать собственные структуры данных.

Использование пакета `container/heap`

В этом подразделе мы рассмотрим функциональность, которую обеспечивает пакет `container/heap`. Запомните, что пакет `container/heap` реализует кучу — дерево, где значение каждого узла является наименьшим элементом в его поддереве. Обратите внимание, что я использую выражение «*наименьший элемент*» вместо «*минимальное значение*», чтобы подчеркнуть, что куча поддерживает не только числовые значения.

Однако, как легко догадаться, для того чтобы реализовать дерево кучи в Go, необходимо самостоятельно разработать способ, позволяющий определить, какой из двух элементов меньше. В таких случаях в Go используются *интерфейсы*, которые позволяют определять такое поведение.

Это означает, что пакет `container/heap` сложнее, чем два остальных пакета, входящих в состав `container`, и что вам, прежде чем использовать функциональность пакета `container/heap`, придется кое-что определить. Строго говоря, пакет `container/heap` требует реализации `container/heap.Interface`, который определяется следующим образом:

```
type Interface interface {
    sort.Interface
    Push(x interface{}) // вставляет x как элемент Len()
    Pop() interface{}    // удаляет и возвращает элемент Len() - 1
}
```

Подробнее об интерфейсах вы узнаете в главе 7. Пока просто запомните: для соответствия интерфейсу Go требуется реализация одной или нескольких функций или других интерфейсов, в данном случае это `sort.Interface` и функции `Push()` и `Pop()`.

Для `sort.Interface` необходимо реализовать функции `Len()`, `Less()` и `Swap()`. Это имеет смысл, поскольку вы не сможете выполнять какую-либо сортировку, не имея возможности поменять местами два элемента и вычислить значение того, что хотите сортировать, и не имея возможности определять, какой из двух элементов больше, на основе вычисленных ранее значений. Вам может показаться, что это значительный объем работы. Однако в большинстве случаев реализация этих функций является очень простой.

Поскольку цель данного раздела — продемонстрировать использование пакета `container/heap`, а не усложнить вам жизнь, в примере используются элементы с типом данных `float32`.

Мы рассмотрим код Go, представленный в файле `conHeap.go`. Разделим его на пять частей. Первая часть выглядит следующим образом:

```
package main

import (
    "container/heap"
    "fmt"
)

type heapFloat32 []float32
```

Вторая часть `conHeap.go` содержит следующий код Go:

```
func (n *heapFloat32) Pop() interface{} {
    old := *n
    x := old[len(old)-1]
    new := old[0 : len(old)-1]
    *n = new
    return x
}

func (n *heapFloat32) Push(x interface{}) {
    *n = append(*n, x.(float32))
}
```

Здесь мы определили две функции с именами `Pop()` и `Push()`, которые используются для того, чтобы соответствовать интерфейсу¹. Чтобы добавлять и удалять элементы в куче, необходимо вызывать функции `heap.Push()` и `heap.Pop()` соответственно.

¹ Функция `Pop()` возвращает тип `interface{}`, которому удовлетворяет любой тип Go. В приведенном выше примере возвращается тип данных `float32`, но если нужно хранить в куче указатели, то при извлечении его из кучи нужно не забывать обнулять место его хранения, например, так: `old[len(old) - 1] = nil`. В противном случае сборщик мусора не сможет очистить место под объект, на который указывает извлеченный из кучи указатель.

Третий фрагмент кода `conHeap.go` содержит следующий код Go:

```
func (n heapFloat32) Len() int {
    return len(n)
}

func (n heapFloat32) Less(a, b int) bool {
    return n[a] < n[b]
}

func (n heapFloat32) Swap(a, b int) {
    n[a], n[b] = n[b], n[a]
}
```

В этой части реализованы три функции, необходимые для совместимости с интерфейсом `sort.Interface`.

Четвертая часть `conHeap.go` выглядит так:

```
func main() {
    myHeap := &heapFloat32{1.2, 2.1, 3.1, -100.1}
    heap.Init(myHeap)
    size := len(*myHeap)
    fmt.Printf("Heap size: %d\n", size)
    fmt.Printf("%v\n", myHeap)
```

И последняя часть кода `conHeap.go` выглядит следующим образом:

```
    myHeap.Push(float32(-100.2))
    myHeap.Push(float32(0.2))
    fmt.Printf("Heap size: %d\n", len(*myHeap))
    fmt.Printf("%v\n", myHeap)
    heap.Init(myHeap)
    fmt.Printf("%v\n", myHeap)
}
```

В этой части `conHeap.go` мы добавляем в кучу `myHeap` два новых элемента, используя функцию `heap.Push()`. Однако, для того чтобы восстановить правильную упорядоченность кучи, нужно снова вызвать `heap.Init()`¹.

Выполнение `conHeap.go` приведет к следующим результатам:

```
$ go run conHeap.go
Heap size: 4
&[-100.1 1.2 3.1 2.1]
Heap size: 6
&[-100.1 1.2 3.1 2.1 -100.2 0.2]
&[-100.2 -100.1 0.2 2.1 1.2 3.1]
```

¹ В приведенном примере для добавления в кучу использовалась функция `myHeap.Push()`, которая не поддерживает упорядоченность кучи. Это и привело к необходимости повторной инициации кучи функцией `heap.Init()`. Для автоматического поддержания упорядоченности кучи лучше использовать функцию `heap.Push()`.

Если вам показалось странным, что триплет 2.1 1.2 3.1 в последней строке результатов не отсортирован в соответствии с линейной логикой, то напомним, что куча — это не линейная структура, подобная массиву или срезу, а дерево¹.

Использование пакета container/list

В этом подразделе продемонстрирована работа пакета `container/list` на примере Go-кода из файла `conList.go`, который мы разделим на три части.



Внимание! В пакете `container/list` реализован двусвязный список.

Первая часть `conList.go` содержит следующий код Go:

```
package main

import (
    "container/list"
    "fmt"
    "strconv"
)

func printList(l *list.List) {
    for t := l.Back(); t != nil; t = t.Prev() {
        fmt.Print(t.Value, " ")
    }
    fmt.Println()

    for t := l.Front(); t != nil; t = t.Next() {
        fmt.Print(t.Value, " ")
    }

    fmt.Println()
}
```

Здесь вы видите функцию `printList()`, которая позволяет выводить на экран содержимое переменной `list.List`, переданной в виде указателя. В коде Go показано, как вывести элементы `list.List`, начиная с первого и заканчивая последним, и в обратном порядке. Обычно в программах нужно использовать только один из

¹ Куча гарантирует, что нулевой элемент `(*myHeap)[0]` всегда будет минимальным. Для извлечения из кучи элементов в соответствии с приоритетом нужно использовать функцию `heap.Pop()`.

двух методов. Функции `Prev()` и `Next()` позволяют перебирать элементы списка в прямом и обратном порядке.

Второй фрагмент кода `conList.go` выглядит так:

```
func main() {
    values := list.New()

    e1 := values.PushBack("One")
    e2 := values.PushBack("Two")
    values.PushFront("Three")
    values.InsertBefore("Four", e1)
    values.InsertAfter("Five", e2)
    values.Remove(e2)
    values.Remove(e2)
    values.InsertAfter("FiveFive", e2)
    values.PushBackList(values)

    printList(values)

    values.Init()
}
```

Функция `list.PushBack()` позволяет вставлять объект в конец связанного списка, а функция `list.PushFront()` — в начало списка. Обе функции возвращают вставленный в список элемент.

Если вы хотите вставить новый элемент после определенного элемента, то следует использовать функцию `list.InsertAfter()`. Аналогично, для того чтобы вставить элемент перед конкретным элементом, необходимо применить функцию `list.InsertBefore()`. Если такой элемент не существует, то список не изменится. Функция `list.PushBackList()` вставляет копию существующего списка в конец другого списка, а `list.PushFrontList()` помещает копию существующего списка в начало другого списка. Функция `list.Remove()` удаляет из списка заданный элемент.

Обратите внимание на использование функции `values.Init()`, которая либо очищает существующий список, либо инициализирует новый список.

Последняя часть `conList.go` содержит следующий код Go:

```
fmt.Printf("After Init(): %v\n", values)

for i := 0; i < 20; i++ {
    values.PushFront(strconv.Itoa(i))
}

printList(values)
}
```

Здесь мы создаем новый список с помощью цикла `for`. Функция `strconv.Itoa()` преобразует целочисленное значение в строку.

Таким образом, функции пакета `container/list` достаточно просты и их использование не должно вызывать затруднений.

Выполнение `conList.go` приведет к следующим результатам:

```
$ go run conList.go
Five One Four Three Five One Four Three
Three Four One Five Three Four One Five
After Init(): &{{0xc420074180 0xc420074180 <nil> <nil>}} 0}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Использование пакета `container/ring`

В этом разделе продемонстрировано применение пакета `container/ring` на примере Go-кода из файла `conRing.go`. Разделим его на четыре части. Обратите внимание, что пакет `container/ring` намного проще, чем `container/list` и `container/heap`, следовательно, содержит меньше функций, чем эти два пакета.

Первый фрагмент кода `conRing.go` выглядит так:

```
package main

import (
    "container/ring"
    "fmt"
)

var size int = 10
```

Переменная `size` содержит размер создаваемого кольца.

Вторая часть `conRing.go` содержит следующий код Go:

```
func main() {
    myRing := ring.New(size + 1)
    fmt.Println("Empty ring:", *myRing)

    for i := 0; i < myRing.Len()-1; i++ {
        myRing.Value = i
        myRing = myRing.Next()
    }

    myRing.Value = 2
```

Таким образом, новое кольцо создается с помощью функции `ring.New()`, для которой требуется один параметр — размер кольца. Оператор `myRing.Value = 2` в конце кода добавляет в кольцо значение 2. Однако такое значение в кольце уже есть, поскольку оно уже было добавлено в цикле `for`. Наконец, нулевое значение для кольца — это кольцо с одним элементом, значение которого равно `nil`.

Третья часть `conRing.go` содержит следующий код Go:

```
sum := 0
myRing.Do(func(x interface{}) {
    t := x.(int)
```

```

    sum = sum + t
  })
  fmt.Println("Sum:", sum)

```

Функция `ring.Do()` позволяет вызывать функцию для каждого элемента кольца в хронологическом порядке. Но если эта функция вносит какие-либо изменения в кольцо, то поведение `ring.Do()` становится неопределенным.

Оператор `x.(int)` называется *утверждением типа*. Подробнее об утверждениях типа вы прочитаете в главе 7. Пока просто запомните: этот оператор показывает, что `x` имеет тип `int`.

Последняя часть программы `conRing.go` выглядит так:

```

for i := 0; i < myRing.Len()+2; i++ {
    myRing = myRing.Next()
    fmt.Print(myRing.Value, " ")
}
fmt.Println()
}

```

Единственная проблема с кольцами — это то, что мы можем вызывать `ring.Next()` до бесконечности, поэтому нужно найти способ, как это прекратить. В данном случае это сделано с помощью функции `ring.Len()`. Я предпочитаю использовать для перебора всех элементов кольца функцию `ring.Do()`, поскольку она генерирует более чистый код, но использовать цикл `for` тоже неплохо.

Выполнение `conRing.go` приведет к следующим результатам:

```

$ go run conRing.go
Empty ring: {0xc42000a080 0xc42000a1a0 <nil>}
Sum: 47
0 1 2 3 4 5 6 7 8 9 2 0 1

```

Как видно из результатов, кольцо может содержать повторяющиеся значения, следовательно, если не использовать функцию `ring.Len()`, то у нас не будет безопасного способа узнать размер кольца.

Генерация случайных чисел

Генерация случайных чисел — это искусство, а также предмет исследований в области компьютерных наук. Дело в том, что компьютеры являются чисто логическими машинами и, как оказалось, применять их для генерации случайных чисел очень сложно!

В Go для генерации псевдослучайных чисел применяется пакет `math/rand`. Чтобы начать генерировать числа, нужно *начальное число*. Начальное число используется для инициализации всего процесса и имеет важное значение, поскольку если всегда начинать с одного и того же числа, то мы будем получать одну и ту же последова-

тельность псевдослучайных чисел. Это означает, что данную последовательность сможет восстановить любой желающий и она в итоге не будет случайной.

Утилита, которая поможет нам генерировать псевдослучайные числа, называется `randomNumbers.go`. Рассмотрим ее, разделив на четыре части. Эта утилита принимает различные параметры: нижний и верхний пределы числовой последовательности, которая будет сформирована, а также количество генерируемых чисел. Если указать четвертый параметр команды, то программа будет использовать его в качестве начального числа генератора псевдослучайных чисел, что поможет получить ту же последовательность чисел, — главная причина, для чего это нужно, заключается в возможности тестирования кода.

Первая часть утилиты выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

Всю работу выполняет функция `random()` — она генерирует псевдослучайные числа в заданном диапазоне, вызывая `rand.Intn()`.

Вторая часть утилиты командной строки выглядит так:

```
func main() {
    MIN := 0
    MAX := 100
    TOTAL := 100
    SEED := time.Now().Unix()

    arguments := os.Args
```

В этой части мы инициализируем переменные, которые будут использоваться в программе.

В третьей части `randomNumbers.go` содержится следующий код Go:

```
switch len(arguments) {
case 2:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX = MIN + 100
case 3:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
```

```

    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
case 4:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
    TOTAL, _ = strconv.Atoi(arguments[3])
case 5:
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
    TOTAL, _ = strconv.Atoi(arguments[3])
    SEED, _ = strconv.ParseInt(arguments[4], 10, 64)
default:
    fmt.Println("Using default values!")
}

```

Логика этого блока `switch` относительно проста: в зависимости от количества полученных аргументов командной строки мы используем либо начальные значения отсутствующих аргументов, либо значения, заданные пользователем. Для простоты переменные типа `error` для функций `strconv.Atoi()` и `strconv.ParseInt()` игнорируются (для чего используются символы подчеркивания). Если бы это была коммерческая программа, то нельзя было бы проигнорировать переменные `error` функций `strconv.Atoi()` и `strconv.ParseInt()`.

Наконец, причина использования функции `strconv.ParseInt()` для установки нового значения переменной `SEED` заключается в том, что для функции `rand.Seed()` требуется параметр типа `int64`. Первый параметр `strconv.ParseInt()` — это строка для синтаксического анализа, второй — основание системы счисления для генерируемого числа, а третий — количество битов для этого числа.

Поскольку мы хотим создать десятичное целое число, которое будет занимать 64 бита, то мы указали `10` в качестве основания и `64` в качестве числа битов. Обратите внимание, что для синтаксического анализа целого числа без знака следовало бы использовать функцию `strconv.ParseUint()`.

Последняя часть `randomNumbers.go` содержит следующий код Go:

```

rand.Seed(SEED)
for i := 0; i < TOTAL; i++ {
    myrand := random(MIN, MAX)
    fmt.Print(myrand)
    fmt.Print(" ")
}
fmt.Println()
}

```



Вместо использования времени эпохи UNIX в качестве начального числа для генератора псевдослучайных чисел можно применять системное устройство `/dev/random`. Подробнее о чтении из устройства `/dev/random` вы прочитаете в главе 8.

Выполнение `randomNumbers.go` приведет к следующим результатам:

```
$ go run randomNumbers.go
Using default values!
75 69 15 75 62 67 64 8 73 1 83 92 7 34 8 70 22 58 38 8 54 34 91 65 1 50 76
5 82 61 90 10 38 40 63 6 28 51 54 49 27 52 92 76 35 44 9 66 76 90 10 29 22
20 83 33 92 80 50 62 26 19 45 56 75 40 30 97 23 87 10 43 11 42 65 80 82 25
53 27 51 99 88 53 36 37 73 52 61 4 81 71 57 30 72 51 55 62 63 79
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
1 1
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
2 2
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
```

Если вас действительно заинтересовала генерация случайных чисел, прочитайте второй том книги «Искусство программирования»¹ Дональда Кнута.

Если эти псевдослучайные числа нужны вам для обеспечения безопасности, важно использовать пакет `crypto/rand`. В этом пакете реализован криптографически безопасный генератор псевдослучайных чисел. Рассмотрим его в этой главе.

Генерация случайных строк

Зная, как в компьютере представляются отдельные символы, нетрудно перейти от псевдослучайных чисел к случайным строкам. В этом разделе описан способ создания трудно угадываемых паролей на основе Go-кода `randomNumbers.go`, представленного в предыдущем разделе. Программа Go для решения этой задачи называется `generatePassword.go`. Рассмотрим ее, разделив на четыре части. Утилита принимает только один параметр командной строки — длину пароля, который требуется сгенерировать.

Первая часть `generatePassword.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

¹ Кнут Д. Э. Искусство программирования. Т. 2. Получисленные алгоритмы (The Art of Computer Programming. Volume 2. Seminumerical Algorithms.) — М.: Вильямс, 2001.


```
func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

Вторая часть кода `generatePassword.go` содержит следующий код Go:

```
func main() {
    MIN := 0
    MAX := 94
    var LENGTH int64 = 8

    arguments := os.Args
```

Поскольку мы хотим получать только печатаемые символы ASCII, мы ограничили диапазон генерируемых псевдослучайных чисел: в таблице ASCII содержится 94 печатаемых символа. Это значит, что диапазон псевдослучайных чисел, которые может сгенерировать программа, составляет от 0 до 94, не включая 94.

Третий фрагмент `generatePassword.go` содержит следующий код Go:

```
switch len(arguments) {
case 2:
    LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
default:
    fmt.Println("Using default values!")
}

SEED := time.Now().Unix()
rand.Seed(SEED)
```

Последняя часть `generatePassword.go` выглядит так:

```
startChar := "!"
var i int64 = 1
for {
    myRand := random(MIN, MAX)
    newChar := string(startChar[0] + byte(myRand))
    fmt.Print(newChar)
    if i == LENGTH {
        break
    }
    i++
}
fmt.Println()
}
```

В переменной `startChar` хранится первый символ ASCII, который может быть сгенерирован утилитой. В данном случае это восклицательный знак, ASCII-код которого в десятичном представлении равен 33. Учитывая, что программа генерирует псевдослучайные числа от 0 до 94, максимальное значение ASCII, которое можно получить, равно $93 + 33$, то есть 126, что соответствует ASCII-коду символа `~`.

Далее показана таблица ASCII-кодов с соответствующими десятичными значениями для каждого символа:

0	nul	1	soh	2	st	3	etx	4	eot	5	enq	6	ack	7	bel
8	bs	9	ht	10	nl	11	vt	12	np	13	cr	14	so	15	si
16	dle	17	dc1	18	dc	19	dc3	20	dc4	21	nak	22	syn	23	etb
24	can	25	em	26	su	27	esc	28	fs	29	gs	30	rs	31	us
32	sp	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	-	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	;	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	\	93	]	94	^	95	_
96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	del



Чтобы получить таблицу ASCII-кодов в удобочитаемой форме, введите в любой оболочке UNIX команду `man ascii`.

Выполнение программы `generatePassword.go` с соответствующими параметрами командной строки приведет к следующим результатам:

```
$ go run generatePassword.go
Using default values!
ugs$5mv1
$ go run generatePassword.go
Using default values!
PA/8hA@?
$ go run generatePassword.go 20
HBR+=3\UA'B@ExT4QG|o
$ go run generatePassword.go 20
XLcr|R{*pX/::'t2u^T'
```

Генерация безопасной последовательности случайных чисел

Если вы хотите генерировать на Go более безопасные псевдослучайные числа, следует использовать пакет `crypto/rand`, в котором реализован криптографически безопасный генератор псевдослучайных чисел. Именно об этом пакете речь пойдет в данном разделе.

Мы продемонстрируем применение пакета `crypto/rand` на примере Go-кода `cryptoRand.go`, который разделим на три части.

Первая часть `cryptoRand.go` выглядит так:

```
package main

import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
    "os"
    "strconv"
)

func generateBytes(n int64) ([]byte, error) {
    b := make([]byte, n)
    _, err := rand.Read(b)
    if err != nil {
        return nil, err
    }

    return b, nil
}
```

Вторая часть `cryptoRand.go` содержит следующий код Go:

```
func generatePass(s int64) (string, error) {
    b, err := generateBytes(s)
    return base64.URLEncoding.EncodeToString(b), err
}
```

Последняя часть `cryptoRand.go` выглядит так:

```
func main() {
    var LENGTH int64 = 8
    arguments := os.Args
    switch len(arguments) {
    case 2:
        LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
        if LENGTH <= 0 {
            LENGTH = 8
        }
    default:
        fmt.Println("Using default values!")
    }

    myPass, err := generatePass(LENGTH)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(myPass[0:LENGTH])
}
```

Выполнение `cryptoRand.go` приведет к следующему результату:

```
$ go run cryptoRand.go
Using default values!
hIAFYuvW
$ go run cryptoRand.go 120
WTR15SIcjYQmaMKds0lDfFturG27ovH_HZ6iAi_k0nJC88EDLdvNPcv1Jj0d9DcF0r0S3q2itXZ
801TNaNFpHkT-aMrsjeue6kUyHnx_EaL_vJHy9wL5RTr8
```

Дополнительную информацию о пакете `crypto/rand` вы найдете на странице документации пакета, расположенной по адресу <https://golang.org/pkg/crypto/rand/>.

Выполнение матричных вычислений

Матрица — это двумерный массив. Самый простой способ представить матрицу в Go — с помощью среза. Но если размеры массива заранее известны, то массив тоже отлично справится с этой задачей. Если оба размера матрицы одинаковы, то такая матрица называется *квадратной*.

Есть несколько правил, которые позволяют определить, можно ли выполнять вычисления между двумя матрицами:

- ❑ при сложении или вычитании обе матрицы должны иметь одинаковые размеры;
- ❑ при умножении матрицы *A* на матрицу *B* количество столбцов матрицы *A* должно быть равно количеству строк матрицы *B*. В противном случае умножение матриц *A* и *B* невозможно;
- ❑ при делении матрицы *A* на матрицу *B* должны быть выполнены два условия. Во-первых, необходимо, чтобы у матрицы *B* существовала обратная матрица, а во-вторых, должна быть возможность умножить матрицу *A* на обратную матрицу *B* в соответствии с предыдущим правилом. Обратные матрицы существуют только у квадратных матриц.

Сложение и вычитание матриц

В этом разделе вы узнаете, как складывать и вычитать матрицы с помощью утилиты `addMat.go`, которую мы рассмотрим, разделив на три части. Для реализации матриц в программе используются срезы.

Первая часть `addMat.go` выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)
```

```

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func negativeMatrix(s [][]int) [][]int {
    for i, x := range s {
        for j, _ := range x {
            s[i][j] = -s[i][j]
        }
    }
    return s
}

```

Функция `negativeMatrix()` получает входные данные среза и возвращает новый срез, в котором каждый целочисленный элемент исходного среза заменяется на это же целое число с противоположным знаком. Как вы вскоре увидите, элементы двух исходных матриц генерируются с использованием псевдослучайных чисел, для чего используется функция `random()`.

Вторая часть `addMat.go` содержит следующий код Go:

```

func addMatrices(m1 [][]int, m2 [][]int) [][]int {
    result := make([][]int, len(m1))
    for i, x := range m1 {
        for j, _ := range x {
            result[i] = append(result[i], m1[i][j]+m2[i][j])
        }
    }
    return result
}

```

Функция `addMatrices()` получает элементы обеих матриц, складывает их и создает матрицу результатов.

Последняя часть `addMat.go` выглядит так:

```

func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row, col int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[1])
        return
    }

    col, err = strconv.Atoi(arguments[2])
    if err != nil {

```

```

    fmt.Println("Need an integer: ", arguments[2])
    return
}

fmt.Printf("Using %dx%d arrays\n", row, col)
if col <= 0 || row <= 0 {
    fmt.Println("Need positive matrix dimensions!")
    return
}

m1 := make([][]int, row)
m2 := make([][]int, row)

rand.Seed(time.Now().Unix())
// Инициализация m1 и m2 случайными числами
for i := 0; i < row; i++ {
    for j := 0; j < col; j++ {
        m1[i] = append(m1[i], random(-1, i*j+rand.Intn(10)))
        m2[i] = append(m2[i], random(-1, i*j+rand.Intn(10)))
    }
}
fmt.Println("m1:", m1)
fmt.Println("m2:", m2)

// Сложение
r1 := addMatrices(m1, m2)
// Вычитание
r2 := addMatrices(m1, negativeMatrix(m2))
fmt.Println("r1:", r1)
fmt.Println("r2:", r2)
}

```

Функция `main()` представляет собой контроллер программы. Помимо прочего, она проверяет, правильного ли типа данные введены, создает нужные матрицы и заполняет их сгенерированными псевдослучайными числами.

Выполнение `addMat.go` приведет к следующим результатам:

```

$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 3 2

```

Using 3x2 arrays

```
m1: [[0 -1] [0 0] [0 1]]
m2: [[2 1] [0 3] [1 9]]
r1: [[2 0] [0 3] [1 10]]
r2: [[-2 -2] [0 -3] [-1 -8]]
```

Умножение матриц

Как вы уже знаете, умножение матриц намного сложнее, чем сложение или вычитание. Это заметно по количеству аргументов командной строки, которое требуется утилите `mulMat.go`, описанной в этом разделе. Утилита `mulMat.go`, которую мы рассмотрим, разделив на четыре части, принимает четыре аргумента командной строки, соответствующие размерам первой и второй матриц.

Первая часть `mulMat.go` выглядит так:

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

Вторая часть `mulMat.go` содержит следующий код Go:

```
func multiplyMatrices(m1 [][]int, m2 [][]int) ([][]int, error) {
    if len(m1[0]) != len(m2) {
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]int, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]int, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}
```

Умножение матриц совершенно не похоже на сложение, и это видно по реализации функции `multiplyMatrices()`. Функция `multiplyMatrices()` также возвращает собственное сообщение об ошибке в случае, если размеры входных матриц не позволяют их перемножить.

Третья часть `mulMat.go` выглядит так:

```
func createMatrix(row, col int) [][]int {
    r := make([][]int, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 5 {
        fmt.Println("Wrong number of arguments!")
        return
    }
```

Функция `createMatrix()` создает срез нужных размеров и заполняет его целыми числами, которые генерируются случайным образом.

Последняя часть `mulMat.go` выглядит так:

```
var row, col int
row, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println("Need an integer: ", arguments[1])
    return
}

col, err = strconv.Atoi(arguments[2])
if err != nil {
    fmt.Println("Need an integer: ", arguments[2])
    return
}

if col <= 0 || row <= 0 {
    fmt.Println("Need positive matrix dimensions!")
    return
}

fmt.Printf("m1 is a %dx%d matrix\n", row, col)
// Инициализация m1 случайными числами
m1 := createMatrix(row, col)
```



```

row, err = strconv.Atoi(arguments[3])
if err != nil {
    fmt.Println("Need an integer: ", arguments[3])
    return
}

col, err = strconv.Atoi(arguments[4])
if err != nil {
    fmt.Println("Need an integer: ", arguments[4])
    return
}

if col <= 0 || row <= 0 {
    fmt.Println("Need positive matrix dimensions!")
    return
}

fmt.Printf("m2 is a %dx%d matrix\n", row, col)
// Инициализация m2 случайными числами
m2 := createMatrix(row, col)
fmt.Println("m1:", m1)
fmt.Println("m2:", m2)

// Умножение
r1, err := multiplyMatrices(m1, m2)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Println("r1:", r1)
}

```

Функция `main()` является контроллером программы, который определяет способ ее работы и проверяет правильность ввода аргументов командной строки.

Выполнение `mulMat.go` приведет к следующим результатам:

```

$ go run mulMat.go 1 2 2 1
m1 is a 1x2 matrix
m2 is a 2x1 matrix
m1: [[-3 -1]]
m2: [[-2] [-1]]
r1: [[7]]
$ go run mulMat.go 5 2 2 1
m1 is a 5x2 matrix
m2 is a 2x1 matrix
m1: [[-1 -2] [-4 -4] [-4 -1] [-2 2] [-5 -5]]
m2: [[-5] [-3]]
r1: [[11] [32] [23] [4] [40]]

```

```
$ go run mulMat.go 1 2 3 4
m1 is a 1x2 matrix
m2 is a 3x4 matrix
m1: [[-3 -4]]
m2: [[-5 -2 -2 -3] [-1 -4 -3 -5] [-5 -2 3 3]]
Cannot multiply the given matrices!
```

Деление матриц

В этом подразделе вы узнаете, как разделить одну матрицу на другую, используя код Go из файла `divMat.go`, который мы рассмотрим, разделив на пять частей. Основная функция `divMat.go` называется `inverseMatrix()`. Назначение `inverseMatrix()` — вычисление обратной матрицы для исходной матрицы, что является довольно сложной задачей. Существуют готовые пакеты Go, которые позволяют инвертировать матрицу, но я решил реализовать ее с нуля.



Не все матрицы обратимы. Неквадратные матрицы не являются обратимыми. Квадратная матрица, которая необратима, называется сингулярной или вырожденной — так случается, если определитель квадратной матрицы равен нулю. Сингулярные матрицы встречаются очень редко.

Утилита `divMat.go` принимает один аргумент командной строки, который определяет размеры используемых квадратных матриц.

Первая часть `divMat.go` выглядит так:

```
package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) float64 {
    return float64(rand.Intn(max-min) + min)
}
```

На этот раз функция `random()` генерирует числа формата `float64`. Вся утилита `divMat.go` оперирует числами с плавающей точкой, главным образом потому, что элементы матрицы, обратной для матрицы с целочисленными элементами, скорее всего, не будут целочисленными.

Вторая часть `divMat.go` содержит следующий код Go:

```
func getCofactor(A [][]float64, temp [][]float64, p int, q int, n int) {
    i := 0
    j := 0

    for row := 0; row < n; row++ {
        for col := 0; col < n; col++ {
            if row != p && col != q {
                temp[i][j] = A[row][col]
                j++
                if j == n-1 {
                    j = 0
                    i++
                }
            }
        }
    }
}

func determinant(A [][]float64, n int) float64 {
    D := float64(0)
    if n == 1 {
        return A[0][0]
    }

    temp := createMatrix(n, n)
    sign := 1

    for f := 0; f < n; f++ {
        getCofactor(A, temp, 0, f, n)
        D += float64(sign) * A[0][f] * determinant(temp, n-1)
        sign = -sign
    }
    return D
}
```

Функции `getCofactor()` и `determinant()` вычисляют элементы, необходимые для обращения матрицы. Если определитель матрицы равен 0, то матрица является сингулярной.

Третья часть `divMat.go` выглядит так:

```
func adjoint(A [][]float64) ([][]float64, error) {
    N := len(A)
    adj := createMatrix(N, N)
    if N == 1 {
        adj[0][0] = 1
        return adj, nil
    }
    sign := 1
    var temp = createMatrix(N, N)
```

```

    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            getCofactor(A, temp, i, j, N)
            if (i+j)%2 == 0 {
                sign = 1
            } else {
                sign = -1
            }
            adj[j][i] = float64(sign) * (determinant(temp, N-1))
        }
    }
    return adj, nil
}

func inverseMatrix(A [][]float64) ([][]float64, error) {
    N := len(A)
    var inverse = createMatrix(N, N)
    det := determinant(A, N)
    if det == 0 {
        fmt.Println("Singular matrix, cannot find its inverse!")
        return nil, nil
    }

    adj, err := adjoint(A)
    if err != nil {
        fmt.Println(err)
        return nil, nil
    }

    fmt.Println("Determinant:", det)
    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            inverse[i][j] = float64(adj[i][j]) / float64(det)
        }
    }
    return inverse, nil
}

```

Функция `adjoint()` вычисляет сопряженную матрицу для данной матрицы. Функция `inverseMatrix()` вычисляет обратную матрицу для данной матрицы.



Программа `divMat.go` — отличный пример кода Go, который необходимо тщательно протестировать, прежде чем использовать на практике. Подробнее о тестировании вы прочтаете в главе 11.

Четвертая часть `divMat.go` содержит следующий код Go:

```

func multiplyMatrices(m1 [][]float64, m2 [][]float64) ([][]float64, error)
{
    if len(m1[0]) != len(m2) {

```

```

        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]float64, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]float64, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}

func createMatrix(row, col int) [][]float64 {
    r := make([][]float64, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

```

Функция `multiplyMatrices()` нужна нам потому, что разделить одну матрицу на другую — это все равно что умножить первую матрицу на матрицу, обратную второй.

Последняя часть `divMat.go` выглядит так:

```

func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer:", arguments[1])
        return
    }

    col := row
    if col <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }
}

```

```

m1 := createMatrix(row, col)
m2 := createMatrix(row, col)
fmt.Println("m1:", m1)
fmt.Println("m2:", m2)

inverse, err := inverseMatrix(m2)
if err != nil {
    fmt.Println(err)
    return
}

fmt.Println("\t\t\tPrinting inverse matrix!")
for i := 0; i < len(inverse); i++ {
    for j := 0; j < len(inverse[0]); j++ {
        fmt.Printf("%.2f\t", inverse[i][j])
    }
    fmt.Println()
}

fmt.Println("\t\t\tPrinting result!")
r1, err := multiplyMatrices(m1, inverse)
if err != nil {
    fmt.Println(err)
    return
}

for i := 0; i < len(r1); i++ {
    for j := 0; j < len(r1[0]); j++ {
        fmt.Printf("%.3f\t", r1[i][j])
    }
    fmt.Println()
}
}

```

Здесь, как и в предыдущих утилитах, функция `main()` управляет выполнением программы и осуществляет необходимые проверки данных, вводимых пользователем, прежде чем продолжать работу.

Выполнение `divMat.go` приведет к следующим результатам:

```

$ go run divMat.go 2
m1: [[-3 -3] [-4 -4]]
m2: [[-3 -5] [-4 -1]]
Determinant: -17
        Printing inverse matrix!
0.06      -0.29
-0.24      0.18
        Printing result!
0.529      0.353
0.706      0.471
$ go run divMat.go 3

```

```

m1: [[-3 -5 -2] [-1 -4 1] [-2 -5 -1]]
m2: [[-2 -4 -5] [-1 0 -2] [-2 -2 1]]
Determinant: -22
    Printing inverse matrix!
0.18    -0.64    -0.36
-0.23    0.55    -0.05
-0.09    -0.18    0.18
    Printing result!
0.773    -0.455    0.955
0.636    -1.727    0.727
0.864    -1.273    0.773
$ go run divMat.go 2
m1: [[-3 -5] [-5 -5]]
m2: [[-5 -3] [-5 -3]]
Singular matrix, cannot find its inverse!
    Printing inverse matrix!
    Printing result!
Cannot multiply the given matrices!

```

Как определить размеры массива

В этом разделе покажу вам способ найти размеры массива, используя Go-программу `dimensions.go`. Этот же метод можно использовать для определения размеров среза.

Go-код программы `dimensions.go` выглядит так:

```

package main

import (
    "fmt"
)

func main() {
    array := [12][4][7][10]float64{}
    x := len(array)
    y := len(array[0])
    z := len(array[0][0])
    w := len(array[0][0][0])
    fmt.Println("x:", x, "y:", y, "z:", z, "w:", w)
}

```

Существует массив `array`, в котором хранятся четыре измерения. Функция `len()`, которой предоставлены соответствующие аргументы, позволяет найти эти измерения. Для того чтобы получить первое измерение, нужно вызвать `len(array)`, чтобы получить второе — `len(array[0])` и т. д.

Выполнение `dimensions.go` приведет к следующим результатам:

```

$ go run dimensions.go
x: 12 y: 4 z: 7 w: 10

```

Разгадывание головоломок sudoku

Основная цель этого раздела — помочь вам понять, что всегда следует использовать простейшую структуру данных, способную выполнить необходимую работу. В данном случае эта структура данных будет срезом, который мы используем для представления и проверки головоломки sudoku. Мы могли бы использовать массив, потому что головоломки sudoku имеют предопределенный размер.

Судoku — это логическая комбинаторная головоломка с числами. Проверка головоломки sudoku заключается в том, чтобы убедиться, что она правильно решена, — такие задачи легко выполняются с помощью компьютерных программ.

Чтобы программа была как можно более универсальной, представленная здесь утилита `sudoku.go`, которую мы разделим на четыре части, будет загружать головоломки sudoku из внешних файлов.

Первая часть `sudoku.go` выглядит так:

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)

func importFile(file string) ([][]int, error) {
    var err error
    var mySlice = make([][]int, 0)

    f, err := os.Open(file)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        fields := strings.Fields(line)
        temp := make([]int, 0)
        for _, v := range fields {
            n, err := strconv.Atoi(v)
            if err != nil {
                return nil, err
            }
        }
    }
}
```



```

        temp = append(temp, n)
    }

    if len(temp) != 0 {
        mySlice = append(mySlice, temp)
    }

    if err == io.EOF {
        break
    } else if err != nil {
        return nil, err
    }

    if len(temp) != len(mySlice[0]) {
        return nil, errors.New("Wrong number of elements!")
    }
}
return mySlice, nil
}

```

Функция `importFile()` проверяет, являются ли прочитанные данные допустимыми целыми числами. Проще говоря, `importFile()` принимает отрицательные целые или целые числа больше 9, но не примет значение `a`, которое не является целым числом или числом с плавающей точкой. Функция `importFile()` выполнит еще одну проверку, чтобы убедиться, что все строки входного файла имеют одинаковое количество целых чисел. В первой строке входного текстового файла указывается количество столбцов, которые должны присутствовать во всех остальных входных строках.

Вторая часть `sudoku.go` содержит следующий код Go:

```

func validPuzzle(sl [][]int) bool {
    for i := 0; i <= 2; i++ {
        for j := 0; j <= 2; j++ {
            iEl := i * 3
            jEl := j * 3
            mySlice := []int{0, 0, 0, 0, 0, 0, 0, 0, 0}
            for k := 0; k <= 2; k++ {
                for m := 0; m <= 2; m++ {
                    bigI := iEl + k
                    bigJ := jEl + m
                    val := sl[bigI][bigJ]
                    if val > 0 && val < 10 {
                        if mySlice[val-1] == 1 {
                            fmt.Println("Appeared 2 times:", val)
                            return false
                        } else {
                            mySlice[val-1] = 1
                        }
                    }
                }
            }
        }
    }
}

```

```

        fmt.Println("Invalid value:", val)
        return false
    }
}
}
}
}
}

```

Чтобы перебрать все элементы головоломки sudoku, в `sudoku.go` используется четыре цикла `for`. Применение четырех циклов `for` для массива 9×9 едва ли вызовет проблемы с производительностью, однако такие проблемы наверняка возникли бы, если бы нам приходилось работать с массивами больших размеров.

Третья часть `sudoku.go` выглядит так:

```

// Тестирование столбцов
for i := 0; i <= 8; i++ {
    sum := 0
    for j := 0; j <= 8; j++ {
        sum = sum + sl[i][j]
    }
    if sum != 45 {
        return false
    }
    sum = 0
}

// Тестирование строк
for i := 0; i <= 8; i++ {
    sum := 0
    for j := 0; j <= 8; j++ {
        sum = sum + sl[j][i]
    }
    if sum != 45 {
        return false
    }
    sum = 0
}

return true
}

```

Представленный код гарантирует, что во всех столбцах и строках головоломки sudoku содержатся только числа.

Последняя часть `sudoku.go` выглядит следующим образом:

```

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Printf("usage: loadFile textFile size\n")
        return
    }
}

```

```

file := arguments[1]

mySlice, err := importFile(file)
if err != nil {
    fmt.Println(err)
    return
}

if validPuzzle(mySlice) {
    fmt.Println("Correct Sudoku puzzle!")
} else {
    fmt.Println("Incorrect Sudoku puzzle!")
}
}

```

Функция `main()` управляет всей программой.

Выполнение `sudoku.go` для различных входных файлов приведет к результатам следующего вида:

```

$ go run sudoku.go OK.txt
Correct Sudoku puzzle!
$ go run sudoku.go noOK1.txt
Incorrect Sudoku puzzle!

```

Дополнительные ресурсы

Вам будет полезно обратиться к следующим ресурсам:

- ❑ изучите сайт, посвященный утилите *Graphviz*. Она позволяет строить графики, используя собственный язык: <http://graphviz.org/>;
- ❑ прочитайте страницу документации подпакетов стандартного Go-пакета `container`: <https://golang.org/pkg/container/>;
- ❑ если вы хотите больше узнать о структурах данных, советую прочитать книгу *The Design and Analysis of Computer Algorithms* Альфреда В. Ахо (Alfred V. Aho), Джона Э. Хопкрофта (John E. Hopcroft) и Джеффри Д. Уллмана (Jeffrey D. Ullman) (Addison-Wesley, 1974). Это отличная книга;
- ❑ чтобы больше узнать о хеш-функциях, посетите страницу https://en.wikipedia.org/wiki/Hash_function;
- ❑ еще одна действительно интересная книга об алгоритмах и структурах данных — *Programming Pearls* Джона Бентли (Jon Bentley) (Addison-Wesley Professional, 1999), а также *More Programming Pearls: Confessions of a Coder*, написанная тем же Джоном Бентли (Addison-Wesley Professional, 1988). Прочитав обе эти книги, вы сможете улучшить свои навыки программирования.

Упражнения

- ❑ Попробуйте изменить логику `generatePassword.go`, выбрав пароль из списка паролей, хранящихся в срезе Go, в сочетании с текущим системным временем или датой.
- ❑ Измените код `queue.go` так, чтобы вместо целых чисел хранить числа с плавающей точкой.
- ❑ Измените код Go в файле `stack.go` так, чтобы его узлы имели три поля с целочисленными данными — `Value`, `Number` и `Seed`. Какие еще изменения, помимо очевидных изменений в определении `Nodestruct`, необходимо внести в остальную часть программы?
- ❑ Можете ли вы изменить код `relatedList.go` так, чтобы узлы связного списка были упорядоченными?
- ❑ Аналогичным образом, можете ли вы изменить код Go в файле `duplyLList.go` так, чтобы узлы списка были упорядоченными? Можете ли вы разработать функцию удаления существующих узлов?
- ❑ Измените код `hashTableLookup.go` так, чтобы в хеш-таблице не было повторяющихся значений. Используйте для этого функцию `lookup()`.
- ❑ Перепишите `sudoku.go` так, чтобы вместо среза использовать хеш-таблицу Go.
- ❑ Перепишите `sudoku.go` так, чтобы вместо среза использовать связный список. Почему это сложно?
- ❑ Создайте программу Go, которая бы вычисляла мощность матриц. Существуют ли какие-либо условия, которые должны быть выполнены, чтобы можно было вычислить мощность матрицы?
- ❑ Реализуйте сложение и вычитание трехмерных массивов.
- ❑ Попробуйте реализовать матрицы на основе структур Go. Каковы основные проблемы такой реализации?
- ❑ Можете ли вы изменить Go-код `generatePassword.go` так, чтобы генерировать пароли, содержащие только заглавные буквы?
- ❑ Попробуйте изменить код `conHeap.go` так, чтобы поддерживать нестандартную и более сложную структуру, а не просто элементы типа `float32`.
- ❑ Реализуйте функциональность удаления узла, которая отсутствует в `connectedList.go`.
- ❑ Считаете ли вы, что двусвязный список улучшил бы код программы `queue.go`? Попробуйте реализовать очередь, используя двусвязный список вместо односвязного.

Резюме

В этой главе рассмотрено много интересных и практических тем, в том числе реализация связных списков, двусвязных списков, пользовательских хеш-таблиц, очередей и стеков в Go, а также использование функций стандартного Go-пакета `container`, проверка головоломок судоку, а также генерация в Go последовательностей псевдослучайных чисел и трудноугадываемых паролей.

Из этой главы вам следует запомнить, что основой каждой структуры данных является определение и реализация ее узлов. Наконец, вы узнали о реализации матричных вычислений.

Я уверен, что следующая глава станет для вас одной из самых интересных и ценных в книге. Ее основная тема — пакеты Go, а также информация о том, как определять и использовать различные типы функций Go в ваших программах. Кроме того, в следующей главе рассмотрены модули, которые, в сущности, являются пакетами, имеющими версии.

6

Неочевидные знания о пакетах и функциях Go

В предыдущей главе вы познакомились с разработкой и использованием специализированных структур данных, таких как связные списки, двоичные деревья и пользовательские хеш-таблицы, а также генерацией случайных чисел и трудноугадываемых паролей, выполнением операций с матрицами в Go.

В этой главе основное внимание уделено *пакетам* Go, которые позволяют организовывать, доставлять и использовать код Go. Наиболее распространенными компонентами пакета Go являются *функции*, они весьма гибки. Кроме того, из данной главы вы узнаете о *модулях* Go, которые представляют собой пакеты с версиями. В последнем разделе рассмотрены несколько расширенных пакетов — они относятся к стандартной библиотеке Go. Это позволит вам лучше понять, что не все пакеты Go одинаково полезны.

В данной главе рассмотрены следующие темы:

- ❑ разработка функций в Go;
- ❑ анонимные функции;
- ❑ функции, которые возвращают несколько значений;
- ❑ присваивание имен возвращаемым значениям функций;
- ❑ функции, которые возвращают другие функции;
- ❑ функции, которые получают другие функции в качестве параметров;
- ❑ функции с переменным числом параметров;
- ❑ разработка пакетов в Go;
- ❑ разработка модулей Go и работа с ними;
- ❑ публичные и приватные объекты пакетов;
- ❑ использование функции `init()` в пакетах;
- ❑ сложный стандартный Go-пакет `html/template`;

- ❑ стандартный пакет `text/template` — еще один по-настоящему сложный Go-пакет, который имеет собственный язык;
- ❑ расширенные пакеты `go/scanner`, `go/parser` и `go/token`;
- ❑ стандартный Go-пакет `syscall` — пакет низкого уровня, который хоть и редко применяется напрямую, однако широко используется другими Go-пакетами.

Что такое Go-пакеты

В Go все поставляется в виде пакетов. Go-пакет — это файл с исходным кодом Go, который начинается с ключевого слова `package`, после него стоит имя пакета. Некоторые пакеты имеют свою структуру. Например, в пакете `net` есть подкаталоги с именами `http`, `mail`, `rpc`, `smtp`, `textproto` и `url`, которые следует импортировать как `net/http`, `net/mail`, `net/rpc`, `net/smtp`, `net/textproto` и `net/url` соответственно.

Кроме пакетов стандартной библиотеки Go, существуют внешние пакеты, которые можно импортировать по полному адресу. Их следует загружать перед первым использованием. Одним из таких примеров является `github.com/matryer/is`. Он хранится в репозитории GitHub.

Как правило, пакеты используются для объединения взаимосвязанных функций, переменных и констант, чтобы их можно было легко переносить и использовать в программах Go. Заметьте, что, кроме пакета `main`, остальные Go-пакеты не являются автономными программами и не могут быть скомпилированы в исполняемые файлы. Это означает, что данные пакеты необходимо прямо или косвенно вызывать из основного пакета, чтобы их можно было использовать. Поэтому если вы попытаетесь выполнить Go-пакет как автономную программу, то будете разочарованы:

```
$ go run aPackage.go
go run: cannot run non-main package
```

Что такое функции Go

Функции являются важным элементом любого языка программирования: именно они позволяют разбивать большие программы на более мелкие и более управляемые части. Функции, насколько это возможно, должны быть независимы друг от друга. Они должны выполнять только одну задачу, и делать это хорошо. Поэтому, если вы обнаружите, что написали функцию, которая выполняет несколько задач, имеет смысл заменить ее несколькими функциями.

Самая популярная функция Go — `main()`, которая используется в любой независимой программе Go. Как вы уже наверняка знаете, все определения функций начинаются с ключевого слова `func`.

Анонимные функции

Анонимные функции можно определить как встроенные элементы и функции без имени. Обычно такие функции используются для реализации того, что не требует длинного кода. В Go функция может возвращать анонимную функцию или принимать ее в качестве аргумента. Кроме того, анонимная функция может быть присвоена переменной Go в качестве значения. Обратите внимание, что анонимные функции также называются *замыканиями*, особенно это принято в терминологии функционального программирования.



Считается хорошей практикой, чтобы анонимная функция была небольшого размера и имела локальное применение. Если она не локальна, то, возможно, лучше сделать ее обычной.

Если анонимная функция подходит для решения задачи, это очень удобно и значительно облегчает жизнь; но не используйте в своих программах слишком много анонимных функций без веской причины. Немного позже мы рассмотрим применение анонимных функций на практике.

Функции, которые возвращают несколько значений

Как вы уже знаете по примеру `strconv.Atoi()`, функции Go способны возвращать несколько значений, что избавляет от необходимости создавать специальную структуру для получения нескольких значений, возвращаемых из функции. Допустим, можно объявить следующую функцию, которая возвращает четыре значения (из них два — типа `int`, одно — типа `float64` и одно — `string`):

```
func aFunction() (int, int, float64, string) {
}
```

Теперь пора более подробно продемонстрировать работу анонимных функций и функций, которые возвращают несколько значений, используя в качестве примера код Go из файла `functions.go`. Мы рассмотрим этот код, разделив его на пять частей.

Первая часть кода `functions.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)
```


Второй фрагмент `functions.go` содержит следующий код Go:

```
func doubleSquare(x int) (int, int) {
    return x * 2, x * x
}
```

Здесь мы видим определение и реализацию функции `doubleSquare()`, которая принимает один параметр типа `int` и возвращает два значения типа `int`.

Третья часть программы `functions.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("The program needs 1 argument!")
        return
    }

    y, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Этот код обрабатывает аргументы командной строки программы.

Четвертая часть программы `functions.go` содержит следующий код Go:

```
square := func(s int) int {
    return s * s
}
fmt.Println("The square of", y, "is", square(y))

double := func(s int) int {
    return s + s
}
fmt.Println("The double of", y, "is", double(y))
```

Обе переменные, `square` и `double`, содержат по анонимной функции. Нужно иметь в виду, что язык позволяет переопределять значение `square`, `double` или любой другой переменной, которой впоследствии будет присвоена анонимная функция, следовательно, функции, присвоенные этим переменным, могут измениться так, что они будут вычислять что-то другое.



В программировании не принято изменять код анонимных функций, которые присвоены переменным, так как это может стать главной причиной ошибок.

Последняя часть `functions.go` выглядит так:

```
fmt.Println(doubleSquare(y))
d, s := doubleSquare(y)
```

```
    fmt.Println(d, s)
}
```

Таким образом, мы можем либо вывести на экран значения, возвращаемые функцией, такой как `doubleSquare()`, либо присвоить их другим переменным.

Выполнение `functions.go` приведет к следующим результатам:

```
$ go run functions.go 1 21
The program needs 1 argument!
$ go run functions.go 10.2
strconv.Atoi: parsing "10.2": invalid syntax
$ go run functions.go 10
The square of 10 is 100
The double of 10 is 20
20 100
20 100
```

Функции, возвращающие именованные значения

В отличие от C, Go позволяет присваивать имена значениям, возвращаемым из функций. Кроме того, если в такой функции встречается оператор `return` без аргументов, то функция автоматически возвращает текущее состояние каждого именованного возвращаемого значения в той последовательности, в которой эти значения были объявлены в определении функции.

Файл с исходным кодом, на примере которого продемонстрированы функции Go с именованными возвращаемыми значениями, называется `returnNames.go`. Рассмотрим его, разделив на три части.

Первая часть программы `returnNames.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)

func namedMinMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return
}
```

В этом фрагменте кода вы видите реализацию функции `namedMinMax()` с именованными возвращаемыми параметрами. Однако здесь есть сложный момент: явно

в операторе `return` функция `namedMinMax()` не возвращает никаких переменных или значений. Тем не менее, поскольку возвращаемые значения перечислены в сигнатуре этой функции, параметры `min` и `max` возвращаются автоматически в том порядке, в котором они объявлены в определении функции.

Второй фрагмент кода `returnNames.go` выглядит так:

```
func minMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return min, max
}
```

В функции `minMax()` также использованы именованные возвращаемые значения, однако в ее операторе `return` явно определены как возвращаемые переменные, так и последовательность, в которой они будут возвращены.

Последняя часть `returnNames.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) < 3 {
        fmt.Println("The program needs at least 2 arguments!")
        return
    }

    a1, _ := strconv.Atoi(arguments[1])
    a2, _ := strconv.Atoi(arguments[2])

    fmt.Println(minMax(a1, a2))
    min, max := minMax(a1, a2)
    fmt.Println(min, max)

    fmt.Println(namedMinMax(a1, a2))
    min, max = namedMinMax(a1, a2)
    fmt.Println(min, max)
}
```

Цель кода Go в функции `main()` — убедиться, что все методы дают одинаковые результаты. Выполнение `returnNames.go` приведет к следующим результатам:

```
$ go run returnNames.go -20 1
-20 1
-20 1
-20 1
-20 1
```



От научного редактора: использование именованных переменных очень полезно, если вам нужно вернуть значение из функции после обработки паники. Например:

```
func Run() (err error) {
    defer func() {
        if r := recover(); r != nil {
            switch t := r.(type) {
            case string:
                err = error.New(t)
            case error:
                err = t
            }
        }
    }()
    panic("A panic")
}
```

Функции, принимающие указатели

Функция может принимать в качестве параметра указатель при условии, что это допускает ее сигнатура. Мы рассмотрим использование указателей в качестве параметров функции на примере кода Go из файла `ptrFun.go`.

Первая часть `ptrFun.go` выглядит так:

```
package main

import (
    "fmt"
)

func getPtr(v *float64) float64 {
    return *v * *v
}
```

Как видите, функция `getPtr()` принимает в качестве параметра указатель, который ссылается на значение типа `float64`.

Вторая часть программы содержит следующий код Go:

```
func main() {
    x := 12.2
    fmt.Println(getPtr(&x))
    x = 12
    fmt.Println(getPtr(&x))
}
```

Здесь сложность заключается в том, чтобы передать адрес переменной в функцию `getPtr()`, поскольку для этого требуется параметр указателя. Это можно

сделать, поставив перед переменной знак амперсанда (&x). Выполнение `ptrFun.go` приведет к следующим результатам:

```
$ go run ptrFun.go
148.83999999999997
144
```

Если попытаться передать в `getPtr()` простое значение, такое как `12.12`, и вызвать функцию — например, `getPtr(12.12)`, — то компиляция программы завершится неудачно и будет выведено следующее сообщение об ошибке:

```
$ go run ptrFun.go
# command-line-arguments
./ptrFun.go:15:21: cannot use 12.12 (type float64) as type *float64 in argument
to getPtr
```

Функции, которые возвращают указатели

Как показано в программе `pointerStruct.go` в главе 4, целесообразно создавать новые структурные переменные с использованием отдельной функции и возвращать указатель на них из этой функции. Поэтому сценарий создания функций, возвращающих указатели, очень распространен. Вообще, такая функция упрощает структуру программы и позволяет разработчику сосредоточиться на более важном, вместо того чтобы раз за разом копировать один и тот же код Go. В этом разделе рассмотрен более простой пример, который вы найдете в файле `returnPtr.go`.

Первая `returnPtr.go` содержит следующий код Go:

```
package main

import (
    "fmt"
)

func returnPtr(x int) *int {
    y := x * x
    return &y
}
```

Помимо ожидаемой преамбулы, в этой части кода определена новая функция, которая возвращает указатель на переменную типа `int`. Единственное, что нужно помнить, — это использовать `&y` в операторе `return`, чтобы этот оператор возвращал адрес памяти переменной `y`.

Вторая часть `returnPtr.go` выглядит так:

```
func main() {
    sq := returnPtr(10)
    fmt.Println("sq value:", *sq)
```

Символ `*` разыменовывает переменную указателя, то есть возвращает не сам адрес памяти, а фактическое значение, которое хранится по этому адресу.

Последний фрагмент `returnPtr.go` содержит следующий код Go:

```
fmt.Println("sq memory address:", sq)
}
```

Этот код возвращает адрес памяти переменной `sq`, а не значение типа `int`, хранящееся в этой переменной.

Если выполнить `returnPtr.go`, то получим следующий результат (у вас адрес памяти будет другим):

```
$ go run returnPtr.go
sq value: 100
sq memory address: 0xc00009a000
```

Функции, которые возвращают другие функции

В этом разделе вы узнаете, как реализовать функцию Go, которая возвращает другую функцию. В качестве примера рассмотрим код Go `returnFunction.go`, который разделим на три части. Первый фрагмент кода `returnFunction.go` выглядит так:

```
package main

import (
    "fmt"
)

func funReturnFun() func() int {
    i := 0
    return func() int {
        i++
        return i * i
    }
}
```

Как видно из реализации функции `funReturnFun()`, ее возвращаемое значение является анонимной функцией (`func() int`).

Второй фрагмент `returnFunction.go` содержит следующий код:

```
func main() {
    i := funReturnFun()
    j := funReturnFun()
```

В этом коде дважды вызывается функция `funReturnFun()`, и возвращаемое ею значение, которое является функцией, присваивается двум переменным с именами `i` и `j`. Как станет ясно из результатов работы программы, эти две переменные совершенно не связаны между собой.

Последний фрагмент кода `returnFunction.go` выглядит так:

```
fmt.Println("1:", i())
fmt.Println("2:", i())
fmt.Println("j1:", j())
fmt.Println("j2:", j())
fmt.Println("3:", i())
}
```

В данном коде Go переменная `i` трижды использована как `i()`, а переменная `j` — дважды как `j()`. Здесь важно то, что, хотя `i` и `j` были созданы посредством вызова одной и той же функции `funReturnFun()`, они абсолютно независимы и не имеют между собой ничего общего.

Выполнение `returnFunction.go` приведет к следующим результатам:

```
$ go run returnFunction.go
1: 1
2: 4
j1: 1
j2: 4
3: 9
```

Как видно из результатов работы `returnFunction.go`, значение `i` в функции `funReturnFun()` продолжает увеличиваться и не становится равным `0` после каждого вызова `i()` или `j()`.

Функции, которые принимают другие функции в качестве параметров

Функции Go могут принимать в качестве параметров другие функции Go. Это свойство лишь расширяет спектр того, что мы можем делать с функциями Go. Два самых распространенных способа использования этого свойства — функции сортировки элементов и функция `filepath.Walk()`.

Однако в представленном здесь примере, который называется `funFun.go`, мы рассмотрим более простой случай, который работает с целочисленными значениями. Соответствующий код будет представлен в трех частях.

Первый фрагмент кода `funFun.go` содержит следующий код Go:

```
package main

import "fmt"

func function1(i int) int {
    return i + i
}
```

```
func function2(i int) int {
    return i * i
}
```

Итак, у нас есть две функции, каждая из которых принимает значение типа `int` и возвращает значение типа `int`. Мы будем использовать эти функции в качестве параметров для другой функции.

Второй фрагмент `funFun.go` содержит следующий код:

```
func funFun(f func(int) int, v int) int {
    return f(v)
}
```

Функция `funFun()` принимает два параметра: функцию с именем `f` и значение типа `int`. Параметр `f` должен быть функцией, которая принимает один аргумент типа `int` и возвращает значение типа `int`.

Последний фрагмент кода `funFun.go` выглядит так:

```
func main() {
    fmt.Println("function1:", funFun(function1, 123))
    fmt.Println("function2:", funFun(function2, 123))
    fmt.Println("Inline:", funFun(func(i int) int {return i * i * i}, 123))
}
```

В первом вызове `fmt.Println()` используется `funFun()`, где в качестве начального параметра указано имя функции `function1` без скобок, а во втором вызове `fmt.Println()` используется `funFun()`, где первым параметром является `function2`.

В последнем выражении `fmt.Println()` происходит нечто волшебное: реализация функции-параметра находится внутри вызова `funFun()`. Этот метод отлично работает для простых и коротких функций-параметров, однако в случае функций с большим количеством строк кода Go он может выглядеть гораздо хуже.

Выполнение `funFun.go` приведет к следующим результатам:

```
$ go run funFun.go
function1: 246
function2: 15129
Inline: 1860867
```

Функции с переменным числом параметров

Go также поддерживает функции, которые принимают переменное число аргументов. Наиболее популярные функции с переменным числом аргументов входят в пакет `fmt`. Мы рассмотрим такие функции на примере программы `variadic.go`. Разделим ее на три части.

Первая часть `variadic.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
)

func varFunc(input ...string) {
    fmt.Println(input)
}
```

В этой части кода представлена реализация функции `varFunc()` с переменным числом аргументов строкового типа. Аргумент `input` — срез, который будет обрабатываться внутри функции `varFunc()` именно как срез. Оператор `...`, используемый как `...Type`, называется *оператором упаковки*. Соответствующий *оператор распаковки* начинается со среза и заканчивается знаком `...`. В функции с переменным числом аргументов оператор упаковки может использоваться только один раз.

Вторая часть `variadic.go` содержит следующий код Go:

```
func oneByOne(message string, s ...int) int {
    fmt.Println(message)
    sum := 0
    for i, a := range s {
        fmt.Println(i, a)
        sum = sum + a
    }
    s[0] = -1000
    return sum
}
```

Здесь вы видите еще одну функцию с переменным числом аргументов, которая называется `oneByOne()`. Эта функция принимает одну строку и переменное число целочисленных аргументов. Аргумент функции `s` является срезом.

Последняя часть `variadic.go` выглядит так:

```
func main() {
    arguments := os.Args
    varFunc(arguments...)
    sum := oneByOne("Adding numbers...", 1, 2, 3, 4, 5, -1, 10)
    fmt.Println("Sum:", sum)
    s := []int{1, 2, 3}
    sum = oneByOne("Adding numbers...", s...)
    fmt.Println(s)
}
```

Функция `main()` вызывает и использует две функции с переменным числом аргументов. Поскольку во втором вызове `oneByOne()` используется срез, то любые

изменения, которые вносятся в срез внутри функции с переменным числом параметров, после выхода из функции сохраняются.

Компиляция и выполнение `variadic.go` приведут к следующим результатам:

```
$ ./variadic 1 2
[./variadic 1 2]
Adding numbers...
0 1
1 2
2 3
3 4
4 5
5 -1
6 10
Sum: 24
Adding numbers...
0 1
1 2
2 3
[-1000 2 3]
```

Разработка Go-пакетов

В исходный код Go-пакета может входить несколько файлов и несколько каталогов. Он обычно размещается в одном каталоге, его имя совпадает с именем пакета, за единственным очевидным исключением — пакет `main` может размещаться где угодно.

В этом разделе мы разработаем простой Go-пакет с именем `aPackage`. Файл пакета называется `aPackage.go`, а его исходный код мы рассмотрим, разделив на две части.

Первая часть `aPackage.go` содержит следующий код Go:

```
package aPackage

import (
    "fmt"
)

func A() {
    fmt.Println("This is function A!")
}
```

Обратите внимание, что использование заглавных букв в именах Go-пакетов не принято — имя `aPackage` используется только в качестве примера.

Второй фрагмент кода `aPackage.go` выглядит так:

```
func B() {
    fmt.Println("privateConstant:", privateConstant)
}
```

```
const MyConstant = 123
const privateConstant = 21
```

Как видим, создать Go-пакет с нуля очень просто. Правда, сейчас мы еще не можем использовать этот пакет сам по себе, сначала нужно создать пакет с именем `main` и функцией `main()`, который позволит скомпилировать исполняемый файл. В данном случае программа, которая будет использовать `aPackage`, называется `useAPackage.go`, и в ней содержится следующий код Go:

```
import (
    "aPackage"
    "fmt"
)

func main() {
    fmt.Println("Using aPackage!")
    aPackage.A()
    aPackage.B()
    fmt.Println(aPackage.MyConstant)
}
```

Но если вы сейчас попытаетесь выполнить `useAPackage.go`, то получите сообщение об ошибке, которое говорит о том, что мы еще что-то не доделали:

```
$ go run useAPackage.go
useAPackage.go:4:2: cannot find package "aPackage" in any of:
    /usr/local/Cellar/go/1.9.2/libexec/src/aPackage (from $GOROOT)
    /Users/mtsouk/go/src/aPackage (from $GOPATH)
```

Есть еще кое-что, с чем вам нужно справиться. Как вы уже знаете из главы 1, для установки всех внешних пакетов в Go следует выполнить некоторые команды из оболочки UNIX. Сюда также входят пакеты, разработанные вами. Поэтому нужно поместить предыдущий пакет в соответствующий каталог и сделать его доступным для текущего пользователя UNIX. Чтобы установить разработанный вами пакет, необходимо открыть любую оболочку UNIX и выполнить следующие команды:

```
$ mkdir ~/go/src/aPackage
$ cp aPackage.go ~/go/src/aPackage/
$ go install aPackage
$ cd ~/go/pkg/darwin_amd64/
$ ls -l aPackage.a
-rw-r--r-- 1 mtsouk staff 4980 Dec 22 06:12 aPackage.a
```



Если каталога `~/go` еще нет, то его нужно создать с помощью команды `mkdir(1)`. В данном случае вам также нужно будет проделать эту операцию для каталога `~/go/src`.

Выполнение `useAPackage.go` приведет к следующим результатам:

```
$ go run useAPackage.go
Using aPackage!
This is function A!
privateConstant: 21
123
```

Компиляция Go-пакета

Несмотря на то что мы не можем выполнить Go-пакет, если он не содержит функцию `main()`, мы можем скомпилировать этот пакет и создать объектный файл:

```
$ go tool compile aPackage.go
$ ls -l aPackage.*
-rw-r--r--@ 1 mtsouk staff 201 Jan 10 22:08 aPackage.go
-rw-r--r-- 1 mtsouk staff 16316 Mar 4 20:01 aPackage.o
```

Закрытые переменные и функции

Приватные переменные и функции отличаются от публичных тем, что первые могут использоваться и вызываться только внутри пакета. Управление тем, какие функции, константы и переменные являются публичными, а какие — приватными, также называется *инкапсуляцией*.

В Go действует простое правило, согласно которому функции, переменные, типы и т. д., имена которых начинаются с заглавной буквы, являются открытыми. А функции, переменные, типы и т. д., имена которых начинаются со строчной буквы, являются закрытыми. Именно поэтому функция `fmt.Println()` называется `Println()`, а не `println()`. Однако это правило не распространяется на имена пакетов, которые могут начинаться как с прописных, так и со строчных букв.

Функция `init()`

В каждом Go-пакете может присутствовать закрытая функция с именем `init()`, которая автоматически выполняется в начале выполнения пакета.



Функция `init()` — это закрытая функция, то есть ее нельзя вызвать извне пакета, к которому она принадлежит. Кроме того, поскольку пользователь пакета не имеет контроля над функцией `init()`, вам следует хорошо подумать, прежде чем использовать ее в общедоступных пакетах или изменять в ней любое глобальное состояние.

Сейчас я приведу пример кода с несколькими функциями `init()` из разных Go-пакетов. Рассмотрим код простейшего Go-пакета, который называется просто `a`:

```
package a

import (
    "fmt"
)

func init() {
    fmt.Println("init() a")
}

func FromA() {
    fmt.Println("fromA()")
}
```

В этом пакете реализована `init()` и публичная функция с именем `FromA()`.

После этого из оболочки UNIX нужно выполнить следующие команды, чтобы пакет стал доступен текущему пользователю UNIX:

```
$ mkdir ~/go/src/a
$ cp a.go ~/go/src/a/
$ go install a
```

Теперь рассмотрим код Go-пакета, который называется `b`:

```
package b

import (
    "a"
    "fmt"
)

func init() {
    fmt.Println("init() b")
}

func FromB() {
    fmt.Println("fromB()")
    a.FromA()
}
```

Что здесь происходит? Пакет `a` использует стандартный Go-пакет `fmt`. Однако пакет `b` должен импортировать пакет `a`, поскольку в нем применяется функция `a.FromA()`. И в `a`, и в `b` есть функция `init()`.

Как и раньше, нам нужно установить этот пакет и сделать его доступным для текущего пользователя UNIX, выполнив из оболочки UNIX следующие команды:

```
$ mkdir ~/go/src/b/
$ cp b.go ~/go/src/b
$ go install b
```

Таким образом, сейчас у нас есть два Go-пакета, в каждом из которых есть функция `init()`. Теперь попытайтесь угадать, что мы получим, если выполним файл `manyInit.go`, который содержит следующий код:

```
package main

import (
    "a"
    "b"
    "fmt"
)

func init() {
    fmt.Println("init() manyInit")
}

func main() {
    a.FromA()
    b.FromB()
}
```

В сущности, главный вопрос — сколько раз будут выполняться функции `init()` этих пакетов? Выполнение `manyInit.go` приведет к следующим результатам, что и будет ответом на наш вопрос:

```
$ go run manyInit.go
init() a
init() b
init() manyInit
fromA()
fromB()
fromA()
```

Как видите, функция `init()` для пакета `a` выполняется только один раз, несмотря на то что пакет импортируется дважды, двумя разными пакетами. Кроме того, поскольку сначала выполняется блок `import` из `manyInit.go`, функции `init()` пакетов `a` и `b` запускаются раньше, чем функция `init()` файла `manyInit.go`, что вполне оправданно. Основная причина этого поведения заключается в том, что функции `init()` файла `manyInit.go` разрешено использовать элемент из `a` или из `b`.

Функция `init()` может быть очень полезной, если нужно присвоить значения некоторым неэкспортированным внутренним переменным. Например, в `init()` можно определить текущий часовой пояс. Наконец, учтите, что в одном файле может быть много функций `init()`, однако в Go эта функция используется редко.

Go-модули

Модули были впервые представлены в версии Go 1.11. На момент написания этой книги последней версией Go была 1.13. Несмотря на то что общая идея модулей Go остается неизменной, некоторые из представленных здесь деталей в будущих версиях Go могут измениться.

Модуль Go похож на Go-пакет с версией. Для версий модулей в Go используется *семантическое управление версиями*. Это означает, что версии начинаются с буквы *v*, за которой следует номер. Поэтому у вас могут быть такие версии, как *v1.0.0*, *v1.0.5* и *v2.0.2*. Здесь *v1*, *v2* или *v3* обозначают основную версию Go-пакета, которая обычно не имеет обратной совместимости. Следовательно, если программа Go работает с версией *v1*, это не гарантирует, что она будет работать с *v2* или *v3* — возможно, программа и будет с ними работать, но рассчитывать на это не следует.

Второе число в версии указывает на свойства модуля. Обычно у *v1.1.0* больше функций, чем у *v1.0.2* или *v1.0.0*, но при этом *v1.1.0* сохраняет совместимость с более ранними версиями.

Наконец, третье число говорит об исправлении ошибок, без внесения каких-либо новых функций. Заметьте, что семантическое управление версиями также используется в именовании версий самого языка Go. Обратите также внимание, что Go-модули позволяют писать код за пределами GOPATH.

Создание и использование Go-модулей

В этом подразделе мы создадим первую версию простейшего модуля. Для этого нам понадобится репозиторий GitHub, чтобы хранить код Go. В моем случае хранилище GitHub находится по адресу <https://github.com/mactsouk/myModule>. Мы начнем с пустого репозитория GitHub, в котором будет только файл `README.md`. Итак, сначала нам нужно выполнить следующую команду, чтобы получить содержимое репозитория GitHub:

```
$ git clone git@github.com:mactsouk/myModule.git
Cloning into 'myModule'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
Resolving deltas: 100% (1/1), done.
```

Если вы выполните ту же команду на своем компьютере, то получите мой репозиторий GitHub, который к тому моменту, как вы будете читать эти строки, не будет пустым. Если хотите создать собственный Go-модуль с нуля, то вам нужно создать собственный пустой репозиторий GitHub.

Создание версии v1.0.0

Для того чтобы создать версию v1.0.0 простейшего Go-модуля, нужно выполнить следующие команды:

```
$ go mod init
go: creating new go.mod: module github.com/mactsouk/myModule
$ touch myModule.go
$ vi myModule.go
$ git add .
$ git commit -a -m "Initial version 1.0.0"
$ git push
$ git tag v1.0.0
$ git push -q origin v1.0.0
$ go list
github.com/mactsouk/myModule
$ go list -m
github.com/mactsouk/myModule
```

Содержимое файла `myModule.go` выглядит так:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.0.0")
}
```

Содержимое созданного ранее файла `go.mod` имеет следующий вид:

```
$ cat go.mod
module github.com/mactsouk/myModule
go 1.12
```

Использование версии v1.0.0

Здесь вы узнаете, как использовать версию v1.0.0 созданного нами модуля Go. Чтобы использовать модули Go, нужно создать программу Go, которая в данном случае называется `useModule.go` и содержит следующий код Go:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
)

func main() {
    v1.Version()
}
```


Нам нужно указать путь к модулю Go (github.com/mactsouk/myModule) — в данном случае у модуля Go также есть псевдоним (v1). Псевдонимы для пакетов в Go используются крайне редко; но в нашем примере это облегчает чтение кода. Тем не менее подобный прием не следует использовать в рабочем коде без уважительной причины.

Если просто попытаться выполнить файл `useModule.go`, который в данном случае будет помещен в каталог `/tmp`, то это не получится, потому что требуемый модуль отсутствует в системе:

```
$ pwd
/tmp
$ go run useModule.go
useModule.go:4:2: cannot find package "github.com/mactsouk/myModule" in any of:
    /usr/local/Cellar/go/1.12/libexec/src/github.com/mactsouk/myModule
(from $GOROOT)
    /Users/mtsouk/go/src/github.com/mactsouk/myModule (from $GOPATH)
```

Поэтому, чтобы получить необходимые модули Go и успешно выполнить `useModule.go`, нужно ввести следующие команды:

```
$ export GO111MODULE=on
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.0.0
go: downloading github.com/mactsouk/myModule v1.0.0
go: extracting github.com/mactsouk/myModule v1.0.0
Version 1.0.0
```

Таким образом, код `useModule.go` является корректной программой и может быть выполнен. Теперь пора привести все в порядок, присвоив `useModule.go` имя и выполнив сборку:

```
$ go mod init hello
go: creating new go.mod: module hello
$ go build
```

Последняя команда генерирует исполняемый файл и размещает его в каталоге `/tmp`, а также создает еще два дополнительных файла с именами `go.sum` и `go.mod`. Содержимое `go.sum` будет следующим:

```
$ cat go.sum
github.com/mactsouk/myModule v1.0.0
h1:eTCn2JewnaJw0REKONrVhHmeDEJ0Q5TAZ0xsSbh8kFs=
github.com/mactsouk/myModule v1.0.0/go.mod
h1:s3ziarTDDvaXaHWYOf/ULi97aoBd6JfnnvAkM8rSuzg=
```

В файле `go.sum` хранится контрольная сумма всех загруженных модулей.

Содержимое `go.mod` выглядит так:

```
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.0.0
```



Обратите внимание: если в файле `go.mod`, принадлежащем нашему проекту, указана версия Go `v1.3.0`, то Go будет использовать версию `v1.3.0`, даже если на данный момент доступна более новая версия Go.

Создание версии v1.1.0

В этом пункте мы создадим новую версию `myModule`, используя другой *тег*. Однако на этот раз, в отличие от предыдущего, нет необходимости выполнять команду `go mod init`. Нам нужно лишь выполнить следующие команды:

```
$ vi myModule.go
$ git commit -a -m "v1.1.0"
[master ddd0742] v1.1.0
    1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
$ git tag v1.1.0
$ git push -q origin v1.1.0
```

В этой версии `myModule.go` содержится следующий код:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.1.0")
}
```

Использование версии v1.1.0

В этом пункте вы узнаете, как использовать версию `v1.1.0` созданного нами Go-модуля. Воспользуемся *образом* Docker, чтобы обеспечить максимальную независимость от машины, которую мы использовали для разработки модуля. Чтобы получить образ Docker и перейти в его оболочку UNIX, мы воспользуемся следующей командой:

```
$ docker run --rm -it golang:latest
root@884c0d188694:/go# cd /tmp
root@58c5688e3ee0:/tmp# go version
go version go1.13 linux/amd64
```

Как видим, образ Docker использует последнюю версию Go — на момент написания книги это была версия 1.13. Чтобы использовать один или несколько Go-модулей, нужно создать программу Go, которая называется `useUpdatedModule.go`. Она содержит следующий код Go:

```
package main

import (
```

```

    v1 "github.com/mactsouk/myModule"
}

func main() {
    v1.Version()
}

```

Код Go из файла `useUpdatedModule.go` совпадает с кодом Go в файле `useModule.go`. Его преимущество в том, что вы автоматически получите последнее обновление версии `v1`.

Когда программа в образе Docker будет написана, нужно будет сделать следующее:

```

root@58c5688e3ee0:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 91 Mar 2 19:59 useUpdatedModule.go
root@58c5688e3ee0:/tmp# export GO111MODULE=on
root@58c5688e3ee0:/tmp# go run useUpdatedModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0

```

Как видим, `useUpdatedModule.go` автоматически использует последнюю версию `v1` Go-модуля. Крайне важно выполнить команду `export GO111MODULE=on`, чтобы активизировать поддержку модуля.

Если попытаться выполнить программу `useModule.go`, которая находится в каталоге `/tmp` на вашем локальном компьютере, то получим следующий результат:

```

$ ls -l go.mod go.sum useModule.go
-rw----- 1 mtsouk wheel 67 Mar 2 21:29 go.mod
-rw----- 1 mtsouk wheel 175 Mar 2 21:29 go.sum
-rw-r--r-- 1 mtsouk wheel 92 Mar 2 21:12 useModule.go
$ go run useModule.go
Version 1.0.0

```

Это означает, что `useModule.go`, как и раньше, использует более старую версию Go-модуля. Если вы хотите, чтобы в программе `useModule.go` использовалась последняя версия Go-модуля, можете сделать следующее:

```

$ rm go.mod go.sum
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0

```

Если вы захотите вернуться к использованию версии `v1.0.0` модуля, то можете сделать следующее:

```

$ go mod init hello
go: creating new go.mod: module hello

```

```
$ go build
$ go run useModule.go
Version 1.1.0
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.1.0
$ vi go.mod
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.0.0
$ go run useModule.go
Version 1.0.0
```

В следующем пункте мы создадим старшую версию Go-модуля. Это означает, что вместо нового тега нам потребуется использовать другую ветку GitHub.

Создание версии v2.0.0

В этом пункте мы создадим вторую старшую версию `myModule`. Обратите внимание, что старшие версии нужно явно указать в операторах `import`.

Таким образом, вместо `github.com/mactsouk/myModule` у нас будет `github.com/mactsouk/myModule/v2` для версии v2 и `github.com/mactsouk/myModule/v3` для версии v3.

Прежде всего мы создадим новую ветку GitHub:

```
$ git checkout -b v2
Switched to a new branch 'v2'
$ git push --set-upstream origin v2
```

Затем сделаем следующее:

```
$ vi go.mod
$ cat go.mod
module github.com/mactsouk/myModule/v2
go 1.12
$ git commit -a -m "Using 2.0.0"
[v2 5af2269] Using 2.0.0
2 files changed, 2 insertions(+), 2 deletions(-)
$ git tag v2.0.0
$ git push --tags origin v2
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 441 bytes | 441.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:mactsouk/myModule.git
 * [new branch]      v2 -> v2
 * [new tag]         v2.0.0 -> v2.0.0
```

```
$ git --no-pager branch -a
      master
* v2
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/v2
```

В эту старшую версию `myModule.go` мы поместим следующий код:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.0.0")
}
```

Использование версии v2.0.0

Для того чтобы использовать Go-модули, мы снова создадим программу Go, которая на этот раз будет называться `useV2.go` и содержать следующий код Go:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

Мы будем использовать образ Docker, как самый удобный способ оперирования с Go-модулями, поскольку начнем с чистой установки Go:

```
$ docker run --rm -it golang:latest
root@191d84fc5571:/go# cd /tmp
root@191d84fc5571:/tmp# cat > useV2.go
package main
import (
    v "github.com/mactsouk/myModule/v2"
)
func main() {
    v.Version()
}
root@191d84fc5571:/tmp# export GO111MODULE=on
root@191d84fc5571:/tmp# go run useV2.go
go: finding github.com/mactsouk/myModule/v2 v2.0.0
go: downloading github.com/mactsouk/myModule/v2 v2.0.0
```

```
go: extracting github.com/mactsouk/myModule/v2 v2.0.0
Version 2.0.0
```

Все работает нормально — образ Docker использует модуль `myModule` версии 2.0.0.

Создание версии v2.1.0

Теперь мы создадим обновленную версию `myModule.go`, которая будет использовать другой тег GitHub. Для этого выполним следующие команды:

```
$ vi myModule.go
$ git commit -a -m "v2.1.0"
$ git push
$ git tag v2.1.0
$ git push -q origin v2.1.0
```

Измененное содержимое `myModule.go` будет выглядеть так:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.1.0")
}
```

Использование версии v2.1.0

Как вы уже знаете, чтобы использовать Go-модули, нам нужно создать программу Go, которая будет называться `useUpdatedV2.go`. Она содержит следующий код Go:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

Однако здесь нет необходимости объявлять, что мы хотим использовать последнюю версию v2 Go-модуля, — об этом позаботится сам Go. Именно поэтому программы `useUpdatedV2.go` и `useV2.go` абсолютно одинаковы.

И снова для простоты мы воспользуемся образом Docker. Причина использования команды `cat(1)` для создания `useUpdatedV2.go` заключается в том, что данный образ Docker поставляется без предустановленной утилиты `vi(1)`.

```
$ docker run --rm -it golang:1.12
root@ccfcd675e333:/go# cd /tmp/
root@ccfcd675e333:/tmp# cat > useUpdatedV2.go
package main
import (
    v "github.com/mactsouk/myModule/v2"
)
func main() {
    v.Version()
}
root@ccfcd675e333:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 92 Mar 2 20:34 useUpdatedV2.go
root@ccfcd675e333:/tmp# go run useUpdatedV2.go
useUpdatedV2.go:4:2: cannot find package "github.com/mactsouk/myModule/v2" in any of:
    /usr/local/go/src/github.com/mactsouk/myModule/v2 (from $GOROOT)
    /go/src/github.com/mactsouk/myModule/v2 (from $GOPATH)
root@ccfcd675e333:/tmp# export GO111MODULE=on
root@ccfcd675e333:/tmp# go run useUpdatedV2.go
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 2.1.0
```



Подробнее о `git(1)` и GitHub вы прочитаете в главе 7.

Использование двух версий одного и того же Go-модуля

В этом подразделе вы увидите, как использовать две старшие версии одного и того же Go-модуля в одной программе Go. Этот подход применим и в том случае, если вы хотите использовать одновременно более двух старших версий Go-модуля.

Мы создадим следующую программу Go и сохраним ее в файле `useTwo.go`:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
    v2 "github.com/mactsouk/myModule/v2"
)
```

```
func main() {
    v1.Version()
    v2.Version()
}
```

Как видим, нам нужно лишь явно импортировать старшие версии Go-модуля, которые мы хотим использовать, и присвоить им разные псевдонимы.

Выполнение `useTwo.go` приведет к следующим результатам:

```
$ export GO111MODULE=on
$ go run useTwo.go
go: creating new go.mod: module github.com/PacktPublishing/Mastering-Go-Second-Edition
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 1.1.0
Version 2.1.0
```

Где хранятся Go-модули

В этом подразделе мы узнаем, где и как Go хранит код и информацию о Go-модулях, которые мы используем, на примере созданного нами Go-модуля. После использования этого Go-модуля на моем компьютере с MacOS Mojave содержимое каталога `~/go/pkg/mod/github.com/mactsouk` выглядело так:

```
$ ls -lR ~/go/pkg/mod/github.com/mactsouk
total 0
drwxr-xr-x  3 mtsouk  staff    96B Mar  2 22:38 my!module
dr-x-----  6 mtsouk  staff   192B Mar  2 21:18 my!module@v1.0.0
dr-x-----  6 mtsouk  staff   192B Mar  2 22:07 my!module@v1.1.0
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module:
total 0
dr-x-----  6 mtsouk  staff   192B Mar  2 22:38 v2@v2.1.0
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module/v2@v2.1.0:
total 24
-r--r--r--  1 mtsouk  staff    28B Mar  2 22:38 README.md
-r--r--r--  1 mtsouk  staff    48B Mar  2 22:38 go.mod
-r--r--r--  1 mtsouk  staff    86B Mar  2 22:38 myModule.go
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.0.0:
total 24
-r--r--r--  1 mtsouk  staff    28B Mar  2 21:18 README.md
-r--r--r--  1 mtsouk  staff    45B Mar  2 21:18 go.mod
-r--r--r--  1 mtsouk  staff    86B Mar  2 21:18 myModule.go
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.1.0:
total 24
-r--r--r--  1 mtsouk  staff    28B Mar  2 22:07 README.md
-r--r--r--  1 mtsouk  staff    45B Mar  2 22:07 go.mod
-r--r--r--  1 mtsouk  staff    86B Mar  2 22:07 myModule.go
```




Лучший способ научиться разрабатывать и использовать Go-модули — экспериментировать. Без Go-модулей вам не обойтись, так что начинайте применять их прямо сейчас.

Команда go mod vendor

Иногда необходимо хранить все зависимости в одном месте, рядом с файлами проекта. Именно это позволяет сделать команда `go mod vendor`:

```
$ cd useTwoVersions
$ go mod init useV1V2
go: creating new go.mod: module useV1V2
$ go mod vendor
$ ls -l
total 24
-rw----- 1 mtsouk staff 114B Mar  2 22:43 go.mod
-rw----- 1 mtsouk staff 356B Mar  2 22:43 go.sum
-rw-r--r--@ 1 mtsouk staff 143B Mar  2 19:36 useTwo.go
drwxr-xr-x 4 mtsouk staff 128B Mar  2 22:43 vendor
$ ls -l vendor/github.com/mactsouk/myModule
total 24
-rw-r--r-- 1 mtsouk staff 28B Mar  2 22:43 README.md
-rw-r--r-- 1 mtsouk staff 45B Mar  2 22:43 go.mod
-rw-r--r-- 1 mtsouk staff 86B Mar  2 22:43 myModule.go
drwxr-xr-x 6 mtsouk staff 192B Mar  2 22:43 v2
$ ls -l vendor/github.com/mactsouk/myModule/v2
total 24
-rw-r--r-- 1 mtsouk staff 28B Mar  2 22:43 README.md
-rw-r--r-- 1 mtsouk staff 48B Mar  2 22:43 go.mod
-rw-r--r-- 1 mtsouk staff 86B Mar  2 22:43 myModule.go
```

Главное здесь — выполнение команды `go mod init <имя пакета>` перед командой `go mod vendor`.

Как писать хорошие Go-пакеты

В этом разделе вы найдете полезные советы, которые помогут вам разрабатывать хорошие Go-пакеты. Как вы уже знаете, Go-пакеты хранятся в каталогах, могут содержать публичные и приватные элементы. Публичные элементы могут использоваться как внутри пакета, так и извне, другими пакетами, тогда как приватные могут использоваться только внутри пакета.

Вот несколько хороших правил, которые позволят вам создавать отличные Go-пакеты:

- ❑ первое негласное правило хорошего пакета — то, что его элементы должны быть каким-то образом взаимосвязаны. Например, вы можете написать пакет для автомобилей, а вот сделать единый пакет для поддержки и автомобилей, и велосипедов — плохая идея. Проще говоря, лучше лишний раз разделить функциональность между несколькими пакетами, чем вместить слишком много функций в один Go-пакет. Кроме того, пакеты должны быть простыми и стильными, но не слишком простыми и нестабильными;
- ❑ второе правило заключается в том, что следует сначала в течение какого-то времени использовать собственные пакеты, прежде чем выставлять их на всеобщее обозрение. Это поможет обнаружить ошибки и убедиться, что ваши пакеты работают должным образом. После этого дайте их другим разработчикам для дополнительного тестирования и только потом сделайте общедоступными;
- ❑ попробуйте представить, каким пользователям понравятся ваши пакеты, и убедитесь, что, решив одни проблемы, эти пакеты не создадут новые;
- ❑ пакет не должен экспортировать бесчисленные функции без веских на то причин. Пакеты с кратким списком экспортируемых функций понятнее и проще в использовании. После этого постарайтесь дать своим функциям описательные, но не очень длинные имена;
- ❑ *интерфейсы* способны повысить полезность ваших функций. Поэтому, если вы считаете это целесообразным, используйте интерфейсы вместо отдельных типов в качестве параметров и возвращаемых значений функций;
- ❑ обновляя один из своих пакетов, постарайтесь ничего не поломать, а также не создать несовместимость со старыми версиями, если только в этом нет крайней необходимости;
- ❑ при разработке нового Go-пакета старайтесь использовать несколько файлов, чтобы группировать похожие задачи или концепции;
- ❑ также старайтесь следовать правилам, принятым для разработки Go-пакетов из стандартной библиотеки. В этом вам поможет чтение кода Go-пакетов, входящих в стандартную библиотеку;
- ❑ не создавайте новый пакет, если похожий уже существует. Внесите изменения в существующий пакет или, возможно, создайте собственную версию;
- ❑ никому не понравится, если Go-пакет будет выводить на экран информацию о регистрации. Более профессионально создать флаг, который бы позволил активизировать регистрацию, когда это необходимо;
- ❑ код Go ваших пакетов должен соответствовать коду Go ваших программ. Посмотрите на программу, в которой используются ваши пакеты: если имена функций плохо выделяются в коде, лучше их изменить. Поскольку имя пакета используется практически везде, постарайтесь давать пакетам краткие и выразительные имена;

- ❑ будет удобнее, если вы поместите новые определения типов Go рядом с тем местом, где они будут впервые использоваться: никто, включая вас, не захочет искать в исходных файлах определения новых типов данных;
- ❑ постарайтесь создать для каждого пакета тестовые файлы. Пакеты с тестовыми файлами считаются более профессиональными, чем без них; мелкие детали имеют решающее значение, они дают людям понять, что вы серьезный разработчик! Обратите внимание, что написание тестов для пакетов не является обязательным, но следует избегать использования пакетов, которые не включают в себя тесты. Подробнее о тестировании вы прочитаете в главе 11;
- ❑ наконец, не пишите Go-пакет только потому, что у вас нет ничего лучше, — пишите что-нибудь более подходящее, не тратьте время зря!



Всегда помните, что готовый код Go в пакете не просто не должен содержать ошибок — следующим по важности свойством успешного пакета является его документация, а также несколько примеров кода, в которых демонстрируется его применение и показаны характерные особенности функционирования пакета.

Пакет `syscall`

В этом разделе описана лишь небольшая часть функциональности стандартного Go-пакета `syscall`. Обратите внимание, что в состав пакета `syscall` входит множество функций и типов, связанных с низкоуровневыми примитивами операционной системы. Кроме того, пакет `syscall` широко используется другими Go-пакетами, такими как `os`, `net` и `time`, которые предоставляют кросс-платформенный интерфейс для операционной системы. Таким образом, пакет `syscall` не самый переносимый пакет в библиотеке Go — это и не входит в его задачу.

Несмотря на то что у систем UNIX есть много общего, в них также есть всевозможные различия, особенно если говорить о внутренних системах. Задача пакета `syscall` — как можно осторожнее сгладить все эти несовместимости. Это вовсе не секрет, пакет хорошо задокументирован, и именно поэтому `syscall` так успешно применяется.

Системный вызов (system call) — это способ, позволяющий приложению программно получить что-то от ядра операционной системы. Вследствие этого системные вызовы отвечают за доступ и работу с большинством низкоуровневых элементов UNIX, таких как процессы, устройства хранения данных, печать данных, сетевые интерфейсы и всевозможные файлы. Проще говоря, вы не можете работать в системе UNIX, не используя системные вызовы. Чтобы проверить системные

вызовы процесса UNIX, можно воспользоваться такими утилитами, как `strace(1)` и `dtrace(1)` из главы 2.

Мы рассмотрим использование пакета `syscall` на примере программы `useSyscall.go`, которую разделим на четыре части.



Вам может и не понадобиться использовать пакет `syscall` напрямую, если только вы не работаете над низкоуровневыми программами. Есть Go-пакеты «не для всех», и `syscall` — один из них!

Первая часть кода `useSyscall.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "syscall"
)
```

Это простая часть программы, где мы просто импортируем нужные Go-пакеты. Вторая часть `useSyscall.go` содержит следующий код Go:

```
func main() {
    pid, _, _ := syscall.Syscall(39, 0, 0, 0)
    fmt.Println("My pid is", pid)
    uid, _, _ := syscall.Syscall(24, 0, 0, 0)
    fmt.Println("User ID:", uid)
```

В этой части мы с помощью двух вызовов функции `syscall.Syscall()` получаем информацию об идентификаторах процесса и пользователя. Первый параметр в вызове `syscall.Syscall()` определяет запрашиваемую информацию.

Третий фрагмент `useSyscall.go` содержит следующий код Go:

```
message := []byte{'n', 'e', 'l', 'l', 'o', '!', '\n'}
fd := 1
syscall.Write(fd, message)
```

В этой части мы выводим на экран сообщение с помощью `syscall.Write()`. Первый параметр этой функции — дескриптор файла, в который будут записываться данные, а второй — байтовый срез, в котором, собственно, и хранится выводимое сообщение. Функция `syscall.Write()` является переносимой.

Последняя часть программы `useSyscall.go` содержит такой код:

```
fmt.Println("Using syscall.Exec()")
command := "/bin/ls"
env := os.Environ()
syscall.Exec(command, []string{"ls", "-a", "-x"}, env)
}
```

В последней части программы показано, как можно использовать функцию `syscall.Exec()` для выполнения внешней команды. Однако мы не можем контролировать данные, выводимые командой, — они выводятся на экран автоматически.

Выполнение `useSyscall.go` в операционной системе macOS Mojave приводит к следующим результатам:

```
$ go run useSyscall.go
My pid is 14602
User ID: 501
Hello!
Using syscall.Exec()
.      ..      a.go
funFun.go      functions.go      html.gohtml
htmlT.db      htmlT.go      manyInit.go
ptrFun.go      returnFunction.go      returnNames.go
returnPtr.go      text.gotext      textT.go
useAPackage.go      useSyscall.go
```

Выполнение этой же программы на компьютере с Debian Linux привело к следующим результатам:

```
$ go run useSyscall.go
My pid is 20853
User ID: 0
Hello!
Using syscall.Exec()
.      ..      a.go
funFun.go      functions.go      html.gohtml
htmlT.db      htmlT.go      manyInit.go
ptrFun.go      returnFunction.go      returnNames.go
returnPtr.go      text.gotext      textT.go
useAPackage.go      useSyscall.go
```

Таким образом, несмотря на то что большая часть выходных данных не изменилась, вызов `syscall.Syscall(39, 0, 0, 0)` в Linux не работает, поскольку идентификатор пользователя в Linux не может быть равен 0. Это означает, что данная команда не является переносимой.

Чтобы узнать, какие стандартные Go-пакеты используют пакет `syscall`, можно выполнить из оболочки UNIX следующую команду:

```
$ grep \"syscall\" `find /usr/local/Cellar/go/1.12/libexec/src -name \"*.go\"`
```

Только не забудьте заменить в ней `/usr/local/Cellar/go/1.12/libexec/src` на правильный путь к каталогу в вашей системе.

Как на самом деле работает `fmt.Println()`

Если вы хотите понять, в чем заключается настоящая польза от пакета `syscall`, для начала прочитайте этот подраздел. Реализация функции `fmt.Println()`, которая находится в файле <https://golang.org/src/fmt/print.go>, выглядит следующим образом:

```
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

Таким образом, для выполнения своей работы функция `fmt.Println()` вызывает `fmt.Fprintln()`. Реализация `fmt.Fprintln()`, которая находится в том же файле, выглядит так:

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintln(a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

Как видим, на самом деле операцию записи в `fmt.Fprintln()` выполняет функция `Write()` из интерфейса `io.Writer`. В данном случае интерфейс `io.Writer` — это `os.Stdout`, который описан в файле <https://golang.org/src/os/file.go> следующим образом:

```
var (
    Stdin = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Теперь рассмотрим реализацию функции `NewFile()`, которая находится в файле https://golang.org/src/os/file_plan9.go:

```
func NewFile(fd uintptr, name string) *File {
    fdi := int(fd)
    if fdi < 0 {
        return nil
    }
    f := &File{&file{fd: fdi, name: name}}
    runtime.SetFinalizer(f.file, (*file).close)
    return f
}
```

Увидев исходный файл Go с именем `file_plan9.go`, мы можем предположить, что в нем содержатся команды, специфичные для разных вариантов UNIX, следовательно, в этом файле хранится код, который не является переносимым.

На самом деле здесь показан тип структуры `file`, который встроен в тип `File` и который, судя по его имени, экспортируется. Итак, наши поиски функций, которые применяются к структуре `File` или к указателю на структуру `File` и позволяют записывать данные, мы начнем с файла https://golang.org/src/os/file_plan9.go. Поскольку функция, которую мы ищем, называется `Write()` — посмотрите еще раз на реализацию `Fprintln()`, — чтобы найти ее, нам понадобится пересмотреть все исходные файлы пакета `os`:

```
$ grep "func (f *File) Write(" *.go
file.go:func (f *File) Write(b []byte) (n int, err error) {
```

Реализация функции `Write()` находится в файле <https://golang.org/src/os/file.go> и выглядит следующим образом:

```
func (f *File) Write(b []byte) (n int, err error) {
    if err := f.checkValid("write"); err != nil {
        return 0, err
    }
    n, e := f.write(b)
    if n < 0 {
        n = 0
    }
    if n != len(b) {
        err = io.ErrShortWrite
    }

    epipecheck(f, e)

    if e != nil {
        err = f.wrapErr("write", e)
    }

    return n, err
}
```

Таким образом, теперь нам нужно найти функцию `write()`. Результаты поиска строки `write` в https://golang.org/src/os/file_plan9.go сообщают нам, что в файле https://golang.org/src/os/file_plan9.go есть следующая функция:

```
func (f *File) write(b []byte) (n int, err error) {
    if len(b) == 0 {
        return 0, nil
    }
    return fixCount(syscall.Write(f.fd, b))
}
```

Таким образом, вызов функции `fmt.Println()` реализован с использованием вызова `syscall.Write()`. Этот пример наглядно демонстрирует, насколько полезен и важен пакет `syscall`.

Пакеты `go/scanner`, `go/parser` и `go/token`

В этом разделе вы познакомитесь с пакетами `go/scanner`, `go/parser` и `go/token`, а также `go/ast`. Вы узнаете, как в Go на низком уровне выполняется сканирование и синтаксический анализ кода, а это поможет вам понять, как работает Go. Впрочем,

вы можете пропустить этот раздел, если считаете, что низкоуровневые операции не для вас.

Синтаксический анализ языка выполняется в два этапа. Первый из них заключается в разбиении входных данных на лексемы (*лексический анализ*), а второй — в передаче этих лексем синтаксическому анализатору, чтобы убедиться, что эти лексемы имеют смысл и находятся в правильном порядке (*семантический анализ*), ведь, если произвольно сочетать английские слова, не всегда получится правильное предложение.

Пакет go/ast

Абстрактное синтаксическое дерево (Abstract Syntax Tree, AST) — структурированное представление исходного кода программы Go. Это дерево строится в соответствии с правилами, указанными в спецификации языка. Пакет go/ast используется для объявления типов данных, необходимых для представления AST в Go. Если вы хотите больше узнать о типе ast.*, то пакет go/ast будет наилучшим источником такой информации.

Пакет go/scanner

Сканер — это то, что читает программу, написанную на языке программирования, и генерирует лексемы. В данном случае это код Go, который читает программу, созданную на языке Go.

Пакет go/scanner используется для чтения программ Go и генерации последовательности лексем. Продемонстрируем использование пакета go/scanner на примере программы goScanner.go. Разделим ее на три части.

Первая часть goScanner.go выглядит так:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
```


В пакете `go/token` определены константы, которые представляют лексемы языка программирования Go.

Во второй части `goScanner.go` содержится следующий код Go:

```
for _, file := range os.Args[1:] {
    fmt.Println("Processing:", file)
    f, err := ioutil.ReadFile(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    One := token.NewFileSet()
    files := one.AddFile(file, one.Base(), len(f))
```

Исходный файл, который разбивается на лексемы, хранится в переменной `file`, а его содержимое — в переменной `f`.

Последняя часть `goScanner.go` выглядит так:

```
var myScanner scanner.Scanner
myScanner.Init(files, f, nil, scanner.ScanComments)

for {
    pos, tok, lit := myScanner.Scan()
    if tok == token.EOF {
        break
    }
    fmt.Printf("%s\t%s\t%q\n", one.Position(pos), tok, lit)
}
}
```

Цикл `for` используется здесь для прохода по входному файлу. Конец файла с исходным кодом представлен значением `token.EOF`, которое является признаком завершения цикла `for`. Метод `scanner.Scan()` возвращает текущую позицию в файле, лексему и литерал. Использование константы `scanner.ScanComments` в `scanner.Init()` указывает сканеру возвращать комментарии как лексемы `COMMENT`. Вместо `scanner.ScanComments` можно поставить `1` или `0`, если вы хотите исключить все лексемы `COMMENT` из выходных данных.

Сборка и выполнение `goScanner.go` даст следующие результаты:

```
$ ./goScanner a.go
Processing: a.go
a.go:1:1    package "package"
a.go:1:9    IDENT    "a"
a.go:1:10   ;        "\n"
a.go:3:1    import   "import"
a.go:3:8    (        ""
a.go:4:2    STRING  "\"fmt\""
```

```

a.go:4:7 ;      "\n"
a.go:5:1 )      ""
a.go:5:2 ;      "\n"
a.go:7:1 func   "func"
a.go:7:6 IDENT  "init"
a.go:7:10 (     ""
a.go:7:11 )     ""
a.go:7:13 {     ""
a.go:8:2 IDENT  "fmt"
a.go:8:5 .      ""
a.go:8:6 IDENT  "Println"
a.go:8:13 (     ""
a.go:8:14 STRING "\init() a\""
a.go:8:24 )     ""
a.go:8:25 ;     "\n"
a.go:9:1 }      ""
a.go:9:2 ;     "\n"
a.go:11:1 func  "func"
a.go:11:6 IDENT "FromA"
a.go:11:11 (    ""
a.go:11:12 )    ""
a.go:11:14 {    ""
a.go:12:2 IDENT "fmt"
a.go:12:5 .     ""
a.go:12:6 IDENT "Println"
a.go:12:13 (    ""
a.go:12:14 STRING "\"fromA()\""
a.go:12:23 )    ""
a.go:12:24 ;    "\n"
a.go:13:1 }     ""
a.go:13:2 ;     "\n"

```

Результат `goScanner.go` предельно прост. Обратите внимание, что `goScanner.go` может сканировать файлы любого типа, даже двоичные. Однако при сканировании двоичного файла можно получить менее читаемый результат. Как видно из результатов работы `goScanner.go`, сканер Go добавляет точки с запятой автоматически. Обратите внимание, что `IDENT` уведомляет идентификатор о том, какой тип лексем является наиболее распространенным.

Следующий подраздел посвящен процессу синтаксического анализа.

Пакет go/parser

Синтаксический анализатор получает результаты работы сканера (лексемы) и синтезирует из них структуру. Используя грамматику, которая описывает язык программирования, синтаксический анализатор проверяет, образуют ли полученные лексемы допустимую программу. Созданная структура имеет вид дерева AST.

Применение пакета `go/parser`, который обрабатывает выходные данные, полученные от `go/token`, продемонстрировано на примере программы `goParser.go`. Разделим ее на четыре части.

Первая часть `goParser.go` выглядит так:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strings"
)

type visitor int
```

Вторая часть `goParser.go` содержит следующий код Go:

```
func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }
    fmt.Printf("%sT\n", strings.Repeat("\t", int(v)), n)
    return v + 1
}
```

Метод `Visit()` будет вызываться для каждого узла AST.

Третья часть `goParser.go` выглядит так:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
}
```

Последняя часть `goParser.go` выглядит следующим образом:

```
for _, file := range os.Args[1:] {
    fmt.Println("Processing:", file)
    one := token.NewFileSet()
    var v visitor
    f, err := parser.ParseFile(one, file, nil, parser.AllErrors)
    if err != nil {
        fmt.Println(err)
        return
    }
    ast.Walk(v, f)
}
}
```

Функция `walk()`, вызываемая рекурсивно, перебирает все узлы дерева AST в порядке поиска в глубину.

Сборка и выполнение `goParser.go` с целью построения AST для простого не-большого Go-модуля приводят к следующим результатам:

```
$ ./goParser a.go
Processing: a.go
*ast.File
  *ast.Ident
  *ast.GenDecl
    *ast.ImportSpec
      *ast.BasicLit
  *ast.FuncDecl
    *ast.Ident
    *ast.FuncType
      *ast.FieldList
    *ast.BlockStmt
      *ast.ExprStmt
        *ast.CallExpr
          *ast.SelectorExpr
            *ast.Ident
            *ast.Ident
          *ast.BasicLit
  *ast.FuncDecl
    *ast.Ident
    *ast.FuncType
      *ast.FieldList
    *ast.BlockStmt
      *ast.ExprStmt
        *ast.CallExpr
          *ast.SelectorExpr
            *ast.Ident
            *ast.Ident
          *ast.BasicLit
```

Результат работы `goParser.go` предельно прост. Однако он совершенно не похож на то, что дает на выходе `goScanner.go`.

Теперь, когда вы знаете, как выглядят результаты работы сканера и синтаксического анализатора Go, вы готовы к тому, чтобы рассмотреть несколько практических примеров.

Практический пример

В этом подразделе мы напишем программу Go, которая подсчитывает, сколько раз ключевое слово встречается во входных файлах. В данном случае будет подсчитано количество вхождений ключевого слова `var`. Используемая для этого утилита

называется `varTimes.go`. Рассмотрим ее, разделив на четыре части. Первая часть `varTimes.go` выглядит так:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

var KEYWORD = "var"
var COUNT = 0
```

Мы можем искать любое ключевое слово Go, какое захотим, и даже можем изменять значение глобальной переменной `KEYWORD` во время выполнения программы, внося соответствующие изменения в код `varTimes.go`.

Вторая часть `varTimes.go` содержит следующий код Go:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }

    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        f, err := ioutil.ReadFile(file)
        if err != nil {
            fmt.Println(err)
            return
        }
        one := token.NewFileSet()
        files := one.AddFile(file, one.Base(), len(f))
```

Третья часть `varTimes.go` выглядит так:

```
var myScanner scanner.Scanner
myScanner.Init(files, f, nil, scanner.ScanComments)

localCount := 0
for {
    _, tok, lit := myScanner.Scan()
    if tok == token.EOF {
        break
    }
```

В данном случае позиция, в которой была обнаружена лексема, игнорируется, поскольку это не имеет значения. Однако переменная `tok` все равно необходима, чтобы определить конец файла.

Последняя часть `varTimes.go` выглядит так:

```
        if lit == KEYWORD {
            COUNT++
            localCount++
        }
    }
    fmt.Printf("Found _%s_ %d times\n", KEYWORD, localCount)
}
fmt.Printf("Found _%s_ %d times in total\n", KEYWORD, COUNT)
}
```

Компиляция и выполнение `varTimes.go` приведут к следующим результатам:

```
$ go build varTimes.go
$ ./varTimes varTimes.go variadic.go a.go
Processing: varTimes.go
Found _var_ 3 times
Processing: variadic.go
Found _var_ 0 times
Processing: a.go
Found _var_ 0 times
Found _var_ 3 times in total
```

Поиск имен переменных заданной длины

В этом подразделе рассмотрим еще один практический пример, более сложный, чем тот, что был представлен в программе `varTimes.go`. Вы узнаете, как найти имена переменных заданной длины, — для этого можно задать любую длину строки, какую пожелаете. Кроме того, программа будет различать *глобальные* и *локальные переменные*.



Локальная переменная определена внутри функции, тогда как глобальная — вне функции. Глобальные переменные также называются переменными пакета.

Утилита, которую мы рассмотрим, называется `varSize.go`. Разделим ее на четыре части. Первая часть `varSize.go` выглядит так:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strconv"
)
```

```

var SIZE = 2
var GLOBAL = 0
var LOCAL = 0

type visitor struct {
    Package map[*ast.GenDecl]bool
}

func makeVisitor(f *ast.File) visitor {
    k1 := make(map[*ast.GenDecl]bool)
    for _, aa := range f.Decls {
        v, ok := aa.(*ast.GenDecl)
        if ok {
            k1[v] = true
        }
    }

    return visitor{k1}
}

```

Поскольку мы хотим считать локальные и глобальные переменные отдельно, определим две глобальные переменные с именами `GLOBAL` и `LOCAL`, чтобы хранить эти значения. Структура `visitor` поможет нам различать локальные и глобальные переменные, поэтому создадим в структуре `visitor` поле `map`. Метод `makeVisitor()` используется для инициализации активной структуры `visitor` в соответствии со значениями ее параметра, который является узлом `File`, представляющим весь файл целиком.

Вторая часть `varSize.go` содержит реализацию метода `Visit()`:

```

func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }

    switch d := n.(type) {
    case *ast.AssignStmt:
        if d.Tok != token.DEFINE {
            return v
        }

        for _, name := range d.Lhs {
            v.isItLocal(name)
        }
    case *ast.RangeStmt:
        v.isItLocal(d.Key)
        v.isItLocal(d.Value)
    case *ast.FuncDecl:
        if d.Recv != nil {
            v.CheckAll(d.Recv.List)
        }
    }
}

```

```

    v.CheckAll(d.Type.Params.List)
    if d.Type.Results != nil {
        v.CheckAll(d.Type.Results.List)
    }
case *ast.GenDecl:
    if d.Tok != token.VAR {
        return v
    }
    for _, spec := range d.Specs {
        value, ok := spec.(*ast.ValueSpec)
        if ok {
            for _, name := range value.Names {
                if name.Name == "_" {
                    continue
                }
                if v.Package[d] {
                    if len(name.Name) == SIZE {
                        fmt.Printf("*** %s\n", name.Name)
                        GLOBAL++
                    }
                } else {
                    if len(name.Name) == SIZE {
                        fmt.Printf("* %s\n", name.Name)
                        LOCAL++
                    }
                }
            }
        }
    }
}
}
return v
}

```

Главная задача функции `Visit()` — определить тип узла, с которым она работает, и действовать соответственно. Это выполняется с помощью оператора `switch`.

Узел `ast.AssignStmt` — присвоения или короткие объявления переменных. Узел `ast.RangeStmt` является структурным типом для представления оператора `for` с условием `range` — еще одно место, где могут быть созданы локальные переменные.

Узел `ast.FuncDecl` является структурным типом для представления объявлений функций — переменные, определенные внутри функций, являются локальными. Наконец, узел `ast.GenDecl` — структурный тип для представления объявлений импорта, констант, типа и переменных. Однако нас интересуют только лексемы `token.VAR`.

Третья часть `varSize.go` выглядит так:

```

func (v visitor) isItLocal(n ast.Node) {
    identifier, ok := n.(*ast.Ident)
    if ok == false {

```



```

        return
    }

    if identifier.Name == "_" || identifier.Name == "" {
        return
    }

    if identifier.Obj != nil && identifier.Obj.Pos() == identifier.Pos() {
        if len(identifier.Name) == SIZE {
            fmt.Printf("* %s\n", identifier.Name)
            LOCAL++
        }
    }
}

func (v visitor) CheckAll(fs []*ast.Field) {
    for _, f := range fs {
        for _, name := range f.Names {
            v.isItLocal(name)
        }
    }
}

```

Эти две функции являются вспомогательными методами. Первый из них решает, является ли узел идентификатора локальной переменной, а второй посещает узел `ast.Field` и проверяет его содержимое на наличие локальных переменных.

Последняя часть `varSize.go` выглядит таким образом:

```

func main() {
    if len(os.Args) <= 2 {
        fmt.Println("Not enough arguments!")
        return
    }

    temp, err := strconv.Atoi(os.Args[1])
    if err != nil {
        SIZE = 2
        fmt.Println("Using default SIZE:", SIZE)
    } else {
        SIZE = temp
    }

    var v visitor
    all := token.NewFileSet()
    for _, file := range os.Args[2:] {
        fmt.Println("Processing:", file)
        f, err := parser.ParseFile(all, file, nil, parser.AllErrors)
        if err != nil {
            fmt.Println(err)
        }
    }
}

```

```

        continue
    }

    v = makeVisitor(f)
    ast.Walk(v, f)
}
fmt.Printf("Local: %d, Global:%d with a length of %d.\n", LOCAL, GLOBAL, SIZE)
}

```

Программа генерирует дерево AST на основе полученных входных данных и затем обрабатывает это дерево, извлекая оттуда нужную информацию. Кроме метода `Visit()`, который является частью интерфейса, остальное выполняется в функции `main()` с помощью метода `ast.Walk()`, который автоматически обходит все узлы дерева AST каждого обрабатываемого файла.

Сборка и выполнение `varSize.go` приведут к результатам следующего вида:

```

$ go build varSize.go
$ ./varSize
Not enough arguments!
$ ./varSize 2 varSize.go variadic.go
Processing: varSize.go
* k1
* aa
* ok
* ok
* ok
* fs
Processing: variadic.go
Local: 6, Global:0 with a length of 2.
$ ./varSize 3 varSize.go variadic.go
Processing: varSize.go
* err
* all
* err
Processing: variadic.go
* sum
* sum
Local: 5, Global:0 with a length of 3.
$ ./varSize 7 varSize.go variadic.go
Processing: varSize.go
Processing: variadic.go
* message
Local: 1, Global:0 with a length of 7.

```

Чтобы получить более упорядоченный результат, можно удалить из `varSize.go` многочисленные вызовы `fmt.Println()`.

Go позволяет создавать уникальные решения: если уметь анализировать программы, при желании можно даже написать свой анализатор для собственного языка программирования! Если вы действительно хотите разобраться в том, как

работают синтаксические анализаторы, очень советую заглянуть на страницу документации пакета `go/ast`, которую вы найдете по адресу <https://golang.org/pkg/go/ast/>, а также почитать его исходный код, расположенный по адресу <https://github.com/golang/go/tree/master/src/go/ast>.

Шаблоны для текста и HTML

Тема этого раздела, вероятно, вас приятно удивит: оба представленных здесь пакета обеспечивают потрясающую гибкость. Я уверен, вы найдете множество творческих способов ее применения. *Шаблоны* — это то, что обычно позволяет отделить форматирование от выводимых данных. Обратите внимание, что шаблон Go может быть как файлом, так и строкой. Общая идея такова: следует использовать встроенные строки для небольших шаблонов и внешние файлы для шаблонов крупнее.



Нельзя импортировать в одну и ту же программу Go одновременно и `text/template`, и `html/template`, поскольку данные пакеты имеют одно и то же имя (`template`). Если это крайне необходимо, то следует присвоить одному из них псевдоним. Как это сделать, показано в коде `useStrings.go` из главы 4.

Текст обычно выводится прямо на экран, тогда как HTML-страницы можно увидеть с помощью браузера. Но текст обычно предпочтительнее, чем HTML, если вы планируете обрабатывать его с помощью утилит Go, а также других утилит командной строки UNIX. Поэтому лучше использовать пакет `text/template`, а не `html/template`.

Заметьте, что пакеты `text/template` и `html/template` являются хорошими примерами того, насколько сложным может быть пакет Go. Как вы вскоре узнаете, каждый из них поддерживает собственный язык программирования — благодаря хорошему программному обеспечению сложное выглядит простым и элегантным.

Вывод простого текста

Если вам нужно реализовать вывод простого текста, то для этого отлично подойдет пакет `text/template`. Мы продемонстрируем использование пакета `text/template` на примере программы `textT.go`, которую разделим на пять частей.

Поскольку шаблоны обычно хранятся во внешних файлах, в этом примере используем файл шаблона `text.gotext`. Рассмотрим его, также разделив на три части. Обычно данные считываются из текстовых файлов или из Интернета. Однако в этом примере мы для простоты жестко закодируем данные в программе `textT.go`, представив их в виде среза.

Сначала рассмотрим код `textT.go`. Первая часть кода `textT.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "text/template"
)
```

Второй фрагмент кода `textT.go`:

```
type Entry struct {
    Number int
    Square int
}
```

Вам нужно определить новый тип для хранения данных, если только вы не имеете дело с очень простыми данными.

В третьей части `textT.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need the template file!")
        return
    }

    tFile := arguments[1]
    DATA := [][]int{{-1, 1}, {-2, 4}, {-3, 9}, {-4, 16}}
```

Исходная версия данных хранится в переменной `DATA`, которая представляет собой двумерный срез.

В четвертой части `textT.go` содержится следующий код Go:

```
var Entries []Entry

for _, i := range DATA {
    if len(i) == 2 {
        temp := Entry{Number: i[0], Square: i[1]}
        Entries = append(Entries, temp)
    }
}
```

Этот код создает *срез структур* на основе переменной `DATA`.

Последний фрагмент кода `textT.go` выглядит так:

```
t := template.Must(template.ParseGlob(tFile))
t.Execute(os.Stdout, Entries)
}
```

Для выполнения всех необходимых инициализаций используется функция `template.Must()`. Она возвращает данные типа `Template` — эта структура представля-

ет собой структуру, которая содержит в себе шаблон, обработанный синтаксическим анализатором. Функция `template.ParseGlob()` читает внешний файл шаблона. Забудьте, что я предпочитаю использовать для файлов шаблонов расширение `gohtml`, но вы можете выбрать любое другое расширение — только будьте последовательны.

Наконец, функция `template.Execute()` выполняет всю остальную работу, которая включает в себя обработку данных и вывод результатов в нужный файл — в данном случае это `os.Stdout`.

Теперь рассмотрим код в файле шаблона. Первая часть файла текстового шаблона выглядит так:

```
Calculating the squares of some integers
```

Обратите внимание, что пустые строки в файле текстового шаблона очень важны: они будут отображаться как пустые строки при итоговом выводе данных.

Вторая часть шаблона выглядит так:

```
{{ range . }} The square of {{ printf "%d" .Number }} is {{ printf "%d" .Square }}
```

Здесь много интересного. Ключевое слово `range` позволяет перебрать строки входных данных, которые представлены в виде среза структур. Обычный текст выводится как текст, а переменные и динамический текст должны заключаться в двойные фигурные скобки: `{{ ... }}`. Поля структуры доступны под именами `.Number` и `.Square`. Обратите внимание на знак точки перед именем поля типа данных `Entry`. Наконец, для форматирования окончательных результатов используется команда `printf`.

Третья часть файла `text.gotext` выглядит так:

```
{{ end }}
```

Выполнение команды `{{ range }}` заканчивается командой `{{ end }}`. Если случайно поставить `{{ end }}` не в том месте, это изменит весь вывод данных. Подчеркну: пустые строки в файлах текстовых шаблонов имеют большое значение и будут отражаться при итоговом выводе.

Выполнение `textT.go` приведет к результатам следующего вида:

```
$ go run textT.go text.gotext
Calculating the squares of some integers
The square of -1 is 1
The square of -2 is 4
The square of -3 is 9
The square of -4 is 16
```

Вывод текста в формате HTML

В этом разделе продемонстрировано использование пакета `html/template` на примере программы `htmlT.go`. Мы разделим код этой программы на шесть частей.

Принцип пакета `html/template` такой же, как и у `text/template`. Главное различие между ними состоит в том, что `html/template` генерирует текст в формате HTML, защищенный от атак типа «внедрение кода».



Текст в формате HTML возможно создать и с помощью пакета `text/template`, в конце концов, HTML — это лишь текст. Однако если вы хотите создать именно HTML, то лучше вместо этого пакета использовать `html/template`.

Для простоты мы будем считывать представленные ниже данные из базы данных SQLite. Однако вы можете использовать любую другую базу данных по своему выбору, если у вас есть или если вы сможете написать соответствующие драйверы для Go. Для упрощения будем заполнять таблицу базы данных в нашем примере, перед тем как выполнить чтение из нее.

Первая часть кода `htmlT.go` выглядит так:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
    "html/template"
    "net/http"
    "os"
)

type Entry struct {
    Number int
    Double int
    Square int
}

var DATA []Entry
var tFile string
```

Как видим, в блоке `import` появился новый пакет с именем `net/http` — этот пакет используется для создания в Go серверов и клиентов HTTP. Подробнее о сетевом программировании на Go и об использовании стандартных Go-пакетов `net` и `net/http` вы узнаете из глав 12 и 13.

Кроме пакета `net/http`, мы видим здесь определение типа данных `Entry`, в котором представлены записи, прочитанные из таблицы SQLite3, а также две глобальные переменные с именами `DATA` и `tFile`, в которых хранятся данные, предназначенные для передачи в файл шаблона, и имя файла шаблона соответственно.

Наконец, мы видим здесь использование пакета <https://github.com/mattn/go-sqlite3> для связи с базой данных SQLite3 посредством интерфейса `database/sql`.

Вторая часть `htmlT.go` выглядит так:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Host: %s Path: %s\n", r.Host, r.URL.Path)
    myT := template.Must(template.ParseGlob(tFile))
    myT.ExecuteTemplate(w, tFile, DATA)
}
```

Функция `myHandler()` феноменально проста и эффективна, особенно если учесть ее размер! Функция `template.ExecuteTemplate()` выполняет всю работу. Ее первый параметр — это переменная, в которой хранится соединение с HTTP-клиентом, второй параметр — файл шаблона, который будет использоваться для форматирования данных, а третий параметр — срез структур с данными.

Третий фрагмент `htmlT.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Need Database File + Template File!")
        return
    }

    database := arguments[1]
    tFile = arguments[2]
```

В четвертой части кода `htmlT.go` мы начинаем работать с базой данных:

```
db, err := sql.Open("sqlite3", database)
if err != nil {
    fmt.Println(nil)
    return
}

fmt.Println("Emptying database table.")
_, err = db.Exec("DELETE FROM data")
if err != nil {
    fmt.Println(nil)
    return
}

fmt.Println("Populating", database)
stmt, _ := db.Prepare("INSERT INTO data(number, double, square) values(?,?,?)")
for i := 20; i < 50; i++ {
    _, _ = stmt.Exec(i, 2*i, i*i)
}
```

Функция `sql.Open()` открывает соединение с базой данных. С помощью функции `db.Exec()` мы можем выполнять команды базы данных, не ожидая от нее об-

ратной связи. Наконец, `db.Prepare()` позволяет многократно выполнять команду базы данных, изменяя только ее параметры и вызывая `Exec()`.

В пятой части `htmlT.go` содержится следующий код Go:

```
rows, err := db.Query("SELECT * FROM data")
if err != nil {
    fmt.Println(nil)
    return
}

var n int
var d int
var s int
for rows.Next() {
    err = rows.Scan(&n, &d, &s)
    temp := Entry{Number: n, Double: d, Square: s}
    DATA = append(DATA, temp)
}
```

В этой части программы мы считываем данные из заданной таблицы, используя функцию `db.Query()`, многократные вызовы `Next()` и `Scan()`. Считывая данные, мы помещаем их в срез структур и на этом заканчиваем работу с базой данных.

Последняя часть программы посвящена настройке веб-сервера и содержит следующий код Go:

```
http.HandleFunc("/", myHandler)
err = http.ListenAndServe(":8080", nil)
if err != nil {
    fmt.Println(err)
    return
}
}
```

Здесь функция `http.HandleFunc()` сообщает веб-серверу, какие URL-адреса и какой функцией-обработчиком (`myHandler()`) будут поддерживаться. Текущий обработчик поддерживает адрес `/URL`, что в языке Go соответствует всем URL. Это избавляет нас от необходимости создавать какие-либо дополнительные статические или динамические страницы.

Код программы `htmlT.go` делится на две виртуальные части. Первая из них получает данные из базы и помещает их в срез структур, а вторая, похожая на `textT.go`, предназначена для отображения данных в браузере.



У SQLite есть два больших преимущества: вам не приходится запускать серверный процесс для сервера баз данных, и базы данных SQLite хранятся в автономных файлах. Последнее означает, что вся база данных SQLite может храниться в одном файле.

Обратите внимание: чтобы сократить код Go и обеспечить возможность многократного запуска программы `htmlT.go`, придется вручную создать таблицу базы данных и базу данных SQLite3. Для этого нужно лишь выполнить следующие команды:

```
$ sqlite3 htmlT.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE TABLE data (
...> number INTEGER PRIMARY KEY,
...> double INTEGER,
...> square INTEGER );
sqlite> ^D
$ ls -l htmlT.db
-rw-r--r-- 1 mtsouk staff 8192 Dec 26 22:46 htmlT.db
```

Первая команда выполняется из оболочки UNIX, она создает файл базы данных. Вторая команда выполняется из оболочки SQLite3 и создает таблицу базы данных с именем `data`. У этой таблицы три поля с именами `number`, `double` и `square`.

Кроме того, нам понадобится внешний файл шаблона, который называется `html.gohtml`. Мы будем использовать этот файл для генерации выходных данных программы.

Первая часть `html.gohtml` выглядит так:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Doing Maths in Go!</title>
  <style>
    html {
      font-size: 14px;
    }
    table, th, td {
      border: 2px solid blue;
    }
  </style>
</head>
<body>
```

В основе HTML-кода, который получит браузер, лежит содержимое файла `html.gohtml`. Это значит, что вам нужно сформировать правильную итоговую HTML-страницу, для чего и нужен показанный выше код HTML со встроенными в него стилями CSS для форматирования создаваемой таблицы HTML.

Во второй части `html.gohtml` содержится следующий код:

```
<table>
  <thead>
```

```

        <tr>
            <th>Number</th>
            <th>Double</th>
            <th>Square</th>
        </tr>
    </thead>
    <tbody>
        {{ range . }}
        <tr>
            <td> {{ .Number }} </td>
            <td> {{ .Double }} </td>
            <td> {{ .Square }} </td>
        </tr>
        {{ end }}
    </tbody>
</table>

```

Как видно из этого кода, нам все равно приходится использовать `{{ range }}` и `{{ end }}`, чтобы перебрать все элементы среза структур, который был передан в `template.ExecuteTemplate()`. Однако на этот раз в файле шаблона `html.gohtml` содержится много HTML-кода для представления данных из среза структур в заданном формате.

Последняя часть файла шаблона HTML выглядит так:

```

</body>
</html>

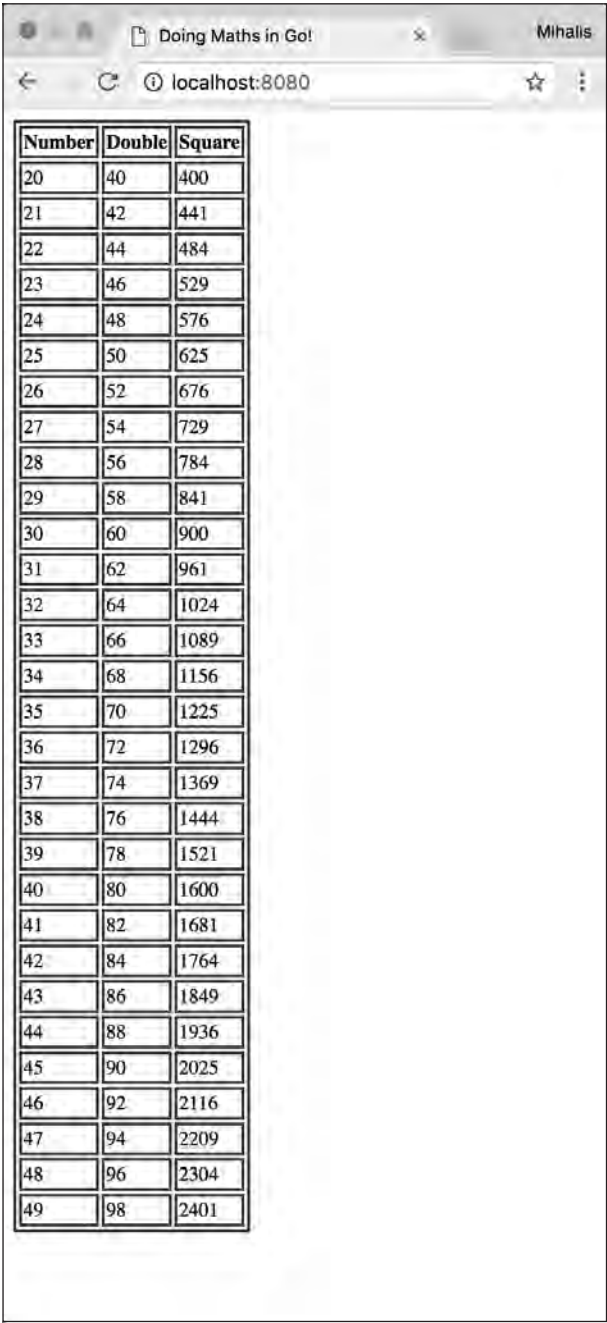
```

Эта часть файла `html.gohtml` предназначена главным образом для правильного завершения сформированного HTML-кода в соответствии со стандартами HTML. Прежде чем скомпилировать и выполнить `htmlT.go`, нам необходимо загрузить пакет, который обеспечит взаимодействие языка программирования Go и базы данных SQLite3. Для этого нужно выполнить следующую команду:

```
$ go get github.com/mattn/go-sqlite3
```

Как вы уже знаете, на компьютере с macOS исходный код загруженного пакета находится в каталоге `~/go/src`, а его скомпилированная версия — в каталоге `~/go/pkg/darwin_amd64`. Если у вас другая ОС, просмотрите содержимое каталога `~/go/pkg`, у которого может быть иная структура. Обратите внимание, что символом `~` обозначается домашний каталог текущего пользователя.

Следует учитывать, что есть и дополнительные пакеты Go, которые обеспечивают обмен данными с базой SQLite3. Однако тот пакет, который использован здесь, является единственным в настоящее время поддерживающим интерфейс `database/sql`. Выполнение `htmlT.go` приведет к тому, что в окне браузера появится таблица, представленная на рис. 6.1.



The image shows a web browser window with the title "Doing Maths in Go!". The address bar displays "localhost:8080". The browser's name "Mihalis" is visible in the top right corner. The main content area contains a table with three columns: "Number", "Double", and "Square". The table lists values for numbers from 20 to 49. Each row contains the number, its double, and its square.

Number	Double	Square
20	40	400
21	42	441
22	44	484
23	46	529
24	48	576
25	50	625
26	52	676
27	54	729
28	56	784
29	58	841
30	60	900
31	62	961
32	64	1024
33	66	1089
34	68	1156
35	70	1225
36	72	1296
37	74	1369
38	76	1444
39	78	1521
40	80	1600
41	82	1681
42	84	1764
43	86	1849
44	88	1936
45	90	2025
46	92	2116
47	94	2209
48	96	2304
49	98	2401

Рис. 6.1. Результат работы программы htmlT.go

Программа `htmlT.go` также выведет в оболочке UNIX результаты следующего вида — в основном это отладочная информация:

```
$ go run htmlT.go htmlT.db html.gohtml
Emptying database table.
Populating htmlT.db
Host: localhost:8080 Path: /
Host: localhost:8080 Path: /favicon.ico
Host: localhost:8080 Path: /123
```

Для того чтобы просмотреть сформированную программой HTML-страницу из оболочки UNIX, можно воспользоваться утилитой `wget(1)`:

```
$ wget -qO- http://localhost:8080
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8">
    <title>Doing Maths in Go!</title>
    <style>
      html {
        font-size: 14px;
      }
      table, th, td {
        border: 2px solid blue;
      }
    </style>
  </head>
  <body>
    <table>
      <thead>
        <tr>
```

И `text/template`, и `html/template` являются мощными пакетами, которые способны сэкономить вам много времени. Поэтому я советую использовать их в тех случаях, когда они соответствуют требованиям ваших приложений.

Дополнительные ресурсы

Следующие ресурсы будут для вас очень полезны:

- ❑ посетите страницу документации стандартного Go-пакета `syscall` по адресу <https://golang.org/pkg/syscall/>. Это одна из самых больших страниц документации Go, какие мне когда-либо встречались;
- ❑ посетите страницу документации пакета `text/template` по адресу <https://golang.org/pkg/text/template/>;
- ❑ также зайдите по ссылке на страницу <https://golang.org/pkg/html/template/>, где вы найдете документацию к пакету `html/template`;

- ❑ подробную информацию о пакете `go/token` вы узнаете по адресу <https://golang.org/pkg/go/token/>;
- ❑ чтобы больше узнать о пакете `go/parser`, зайдите по ссылке на страницу <https://golang.org/pkg/go/parser/>;
- ❑ чтобы больше узнать о пакете `go/scanner`, зайдите на страницу <https://golang.org/pkg/go/scanner/>;
- ❑ подробнее о пакете `go/ast` читайте по адресу <https://golang.org/pkg/go/ast/>;
- ❑ посетите домашнюю страницу SQLite3 по адресу <https://www.sqlite.org/>;
- ❑ посмотрите видео Мэта Райера (Mat Ryer) о том, как писать красивые пакеты на Go: <https://www.youtube.com/watch?v=cAWlv2SeQus>;
- ❑ если вы хотите узнать, что такое *Plan 9*, зайдите сюда: <https://plan9.io/plan9/>;
- ❑ не пожалейте времени на то, чтобы познакомиться с инструментом командной строки `find(1)`, посетив страницу его документации (`man 1 find`).

Упражнения

- ❑ Найдите дополнительную информацию о том, как в действительности реализована функция `fmt.Printf()`.
- ❑ Сможете ли вы написать функцию, которая сортирует три значения типа `int`? Попробуйте написать две версии такой функции: с именованными возвращаемыми значениями и без таких значений. Какая из них, по-вашему, лучше?
- ❑ Сможете ли вы изменить код Go программы `htmlT.go` так, чтобы использовать в ней пакет `text/template` вместо `html/template`?
- ❑ Сможете ли вы изменить код Go программы `htmlT.go` так, чтобы использовать для связи с базой данных SQLite3 пакет <https://github.com/feyeleanor/gosqlite3> или <https://github.com/phf/go-sqlite3>?
- ❑ Создайте свой Go-модуль и разработайте для него три старшие версии.
- ❑ Напишите программу Go, подобную `htmlT.go`, которая бы получала данные из базы данных MySQL. Отметьте, какие изменения вы внесли в код.

Резюме

В этой главе вы познакомились с такими тремя основными темами, как функции Go, Go-пакеты и Go-модули. Главным преимуществом Go-модулей является то, что в них записаны точные требования к зависимостям, что позволяет легко создавать воспроизводимые сборки.

Здесь также приведены исчерпывающие советы по разработке хороших Go-пакетов. Затем вы изучили пакеты `text/template` и `html/template`, которые позволяют выводить на экран простой текст и текст в формате HTML на основе шаблонов, а также пакеты `go/token`, `go/parser` и `go/scanner`. Наконец, вы узнали о стандартном Go-пакете `syscall` и его дополнительных возможностях.

В следующей главе мы обсудим два важных свойства Go: интерфейсы и рефлексия. Кроме того, речь пойдет об объектно-ориентированном программировании в методах Go, методах отладки и типах Go. Это углубленные темы, и поначалу они могут показаться вам сложными. Однако, узнав о них больше, вы, несомненно, улучшите свои навыки программирования на Go.

Наконец, в следующей главе содержится краткое описание основных возможностей утилиты `git`, которая используется для создания Go-модулей.

7 Рефлексия и интерфейсы на все случаи жизни

Из предыдущей главы вы узнали, как разрабатывать пакеты, модули и функции в Go, а также как работать с текстовыми и HTML-шаблонами с помощью пакетов `text/template` и `html/template`. Кроме того, вы научились использовать пакет `syscall`.

В этой главе рассмотрены три очень интересные, удобные и более глубокие концепции Go: *рефлексия*, *интерфейсы* и методы *типов*. Интерфейсы в Go используются повсеместно, в отличие от рефлексии, так как обычно в этом нет необходимости. Вы также познакомитесь с операциями утверждения типа, отладчиком Delve и *объектно-ориентированным программированием* на Go. Наконец, в этой главе описаны основы работы с *git* и *GitHub*.

Таким образом, в данной главе рассмотрены следующие темы:

- ❑ методы типов;
- ❑ интерфейсы Go;
- ❑ операции утверждения типа;
- ❑ разработка и использование собственных интерфейсов;
- ❑ GitHub и git;
- ❑ основы работы с отладчиком Delve;
- ❑ объектно-ориентированное программирование на Go;
- ❑ рефлексия и стандартный Go-пакет `reflect`;
- ❑ рефлексия и библиотека `reflectwalk`.

Методы типов

Метод типа (type method) в Go — это функция со специальным аргументом-приемником. Такой метод объявляется как обычная функция с дополнительным параметром, который ставится перед именем функции. Данный параметр связывает функцию с типом этого дополнительного параметра. Именно этот параметр называется *приемником метода*.

Следующий код Go представляет собой реализацию функции `Close()`, которая находится в файле https://golang.org/src/os/file_plan9.go:

```
func (f *File) Close() error {
    if err := f.checkValid("close"); err != nil {
        return err
    }
    return f.file.close()
}
```

Функция `Close()` является методом типа `File`, поскольку перед ее именем, после ключевого слова `func`, стоит параметр `(f *File)`. Параметр `f` называется приемником метода. В терминологии объектно-ориентированного программирования этот процесс можно описать как передачу сообщения *объекту*. В Go приемнику метода присваивается обычное имя переменной, без специального ключевого слова, такого как `this` или `self`.

Теперь позвольте представить вам полный пример — код Go из файла `method.go`, который мы рассмотрим, разделив на четыре части.

Первая часть `method.go` содержит следующий код Go:

```
package main

import (
    "fmt"
)

type twoInts struct {
    X int64
    Y int64
}
```

В этом коде Go мы видим определение новой структуры `twoInts`, имеющей два поля.

Второй фрагмент кода `methods.go` выглядит так:

```
func regularFunction(a, b twoInts) twoInts {
    temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}
    return temp
}
```

В этой части мы определяем новую функцию с именем `normalFunction()`. Она принимает два параметра типа `twoInts` и возвращает одно значение типа `twoInts`.

В третьей части программы содержится следующий код Go:

```
func (a twoInts) method(b twoInts) twoInts {
    temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}
    return temp
}
```


Функция `method()` эквивалентна функции `normalFunction()`, определенной в предыдущей части `method.go`. Однако `method()` — это метод типа, и вскоре вы узнаете, как его вызывать.



Самое интересное, что у `method()` точно такая же реализация, как и у `regularFunction()`!

Последний фрагмент кода `method.go` выглядит так:

```
func main() {
    i := twoInts{X: 1, Y: 2}
    j := twoInts{X: -5, Y: -2}
    fmt.Println(regularFunction(i, j))
    fmt.Println(i.method(j))
}
```

Как видите, способ вызова метода типа (`i.method(j)`) отличается от способа вызова обычной функции (`regularFunction(i, j)`).

Выполнение `method.go` приведет к следующим результатам:

```
$ go run methods.go
{-4 0}
{-4 0}
```

Обратите внимание, что методы типа связаны с интерфейсами, о которых пойдет речь в следующем разделе. Скоро вы больше узнаете о методах типа.

Интерфейсы в Go

Интерфейсный тип в Go определяет поведение других типов путем предоставления списка методов, которые необходимо реализовать. Чтобы тип соответствовал интерфейсу, в нем необходимо реализовать все методы, предусмотренные этим интерфейсом, — обычно этих методов не очень много.

Проще говоря, интерфейс — это *абстрактный тип*, определяющий множество функций, которые необходимо реализовать для типа, чтобы его можно было считать экземпляром интерфейса. Если это сделано, можно сказать, что тип удовлетворяет данному интерфейсу. Таким образом, интерфейс — это две составляющие: набор методов и тип. Интерфейс используется для определения поведения других типов.

Главное преимущество, которое мы получаем благодаря наличию и использованию интерфейса, состоит в том, что мы можем передавать переменную типа, реализующего этот интерфейс, любой функции, которая ожидает параметр данного интерфейса. Без такой удивительной возможности интерфейсы были бы лишь формальностью без сколь-нибудь значительной практической пользы.



Обратите внимание: если определение и реализация интерфейса находятся в одном и том же Go-пакете, то, возможно, стоит пересмотреть концепцию. Дело не в том, что это технически некорректно, а в том, что это, скорее всего, неверно с логической точки зрения.

Из самых распространенных интерфейсов Go отметим два: `io.Reader` и `io.Writer`. Они используются в операциях ввода и вывода файлов. Точнее, `io.Reader` применяется для чтения из файла, тогда как `io.Writer` — для записи в файл любого типа.

Определение `io.Reader` содержится в файле <https://golang.org/src/io/io.go> и выглядит так:

```
type Reader interface {
    Read(p []byte) (n int, err error)
}
```

Следовательно, чтобы тип удовлетворял интерфейсу `io.Reader`, необходимо реализовать метод `Read()` согласно определению интерфейса.

Аналогично определение `io.Writer` находится в том же файле <https://golang.org/src/io/io.go> и выглядит так:

```
type Writer interface {
    Write(p []byte) (n int, err error)
}
```

Чтобы удовлетворить требованиям интерфейса `io.Writer`, достаточно реализовать всего один метод с именем `Write()`.

Оба интерфейса, и `io.Reader`, и `io.Writer`, требуют реализации лишь одного метода. Тем не менее оба эти интерфейса очень мощные — очевидно, их сила заключается в их простоте. Вообще говоря, большинство интерфейсов довольно просты.

В следующих подразделах показано, как определить интерфейс самостоятельно и как использовать его в других Go-пакетах. Обратите внимание, что интерфейс не обязательно должен быть необычным или впечатляющим — главное, чтобы он выполнял свое предназначение.



Проще говоря, интерфейсы следует использовать в тех случаях, когда необходимо гарантировать выполнение определенных условий и обеспечить определенное поведение, ожидаемое от элемента Go.

Операции утверждения типа

Операции утверждения типа — это записи вида `x.(T)`, где `x` — интерфейсный тип, а `T` — тип. Кроме того, реальное значение, хранящееся в `x`, должно иметь тип `T`, и `T` должен соответствовать интерфейсному типу `x`. Прочитав следующие несколько

абзацев, а также рассмотрев пример кода, вы лучше поймете это немного странное определение операции утверждения типа.

Операции утверждения типа помогают выполнить **две задачи**. Во-первых, они проверяют, соответствует ли значение интерфейса определенному типу. При таком использовании операция утверждения типа возвращает два значения: основное и логическое значение. Скорее всего, вам нужно именно основное значение, однако логическое значение указывает на то, была ли операция утверждения типа успешной.

Во-вторых, операции утверждения типа позволяют использовать конкретное значение, хранящееся в интерфейсе, или присвоить его новой переменной. Например, если в интерфейсе есть переменная типа `int`, то можно получить это значение, используя операцию утверждения типа.

Но если операция утверждения типа завершилась неудачно и этот сбой не был обработан, то программа запаникует. Рассмотрим код Go программы `assertion.go`, который мы разделим на две части. Первая часть выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    var myInt interface{} = 123

    k, ok := myInt.(int)
    if ok {
        fmt.Println("Success:", k)
    }

    v, ok := myInt.(float64)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("Failed without panicking!")
    }
}
```

Сначала мы объявили переменную `myInt`, которая имеет динамический тип `int` и значение `123`. Затем мы дважды использовали операцию утверждения типа для проверки интерфейса переменной `myInt`: один раз для типа `int` и еще один раз для типа `float64`.

Поскольку переменная `myInt` не содержит значения типа `float64`, операция утверждения типа `myInt.(float64)` завершится сбоем, если не будет обработана

должным образом. К счастью, в данном случае правильное использование переменной `ok` не даст программе уйти в панику.

Вторая часть содержит следующий код Go:

```
i := myInt.(int)
fmt.Println("No checking:", i)

j := myInt.(bool)
fmt.Println(j)
}
```

Здесь присутствуют две операции утверждения типа. Первая из них успешна, поэтому проблем с ней не возникает. Расскажу подробнее об этой операции утверждения типа. Здесь типом переменной `i` будет `int`, а ее значением — число `123`, которое хранится в `myInt`. Таким образом, поскольку тип `int` соответствует интерфейсу `myInt`, — в данном случае из-за того, что интерфейс `myInt` не требует реализации никаких функций, — значение `myInt.(int)` имеет тип `int`.

Однако вторая операция утверждения типа — `myInt.(bool)` — вызовет панику, поскольку основное значение `myInt` не является логическим (`bool`).

Поэтому выполнение `assertion.go` приведет к следующим результатам:

```
$ go run assertion.go
Success: 123
Failed without panicking!
No cheking: 123
panic: interface conversion: interface {} is int, not bool
goroutine 1 [running]:
main.main()
  /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch07/assertion.go:25 +0x1c1
exit status 2
```

Go четко указывает на причину паники: `interface {} is int, not bool`.

Вообще, где интерфейсы — там и операции утверждения типа. Еще больше операций утверждений типа встретится в программе `useInterface.go`, которую мы вскоре рассмотрим.

Как писать свои интерфейсы

В этом разделе вы узнаете, как разрабатывать собственные интерфейсы. Это сравнительно просто, главное — знать, что именно вы хотите создать.

Методика создания интерфейсов продемонстрирована на примере кода Go из файла `myInterface.go`, который мы вскоре рассмотрим. Интерфейс, который мы создадим, поможет нам работать с геометрическими фигурами на плоскости.

В `myInterface.go` содержится следующий код:

```
package myInterface

type Shape interface {
    Area() float64
    Perimeter() float64
}
```

Определение интерфейса `shape` действительно очень простое. Оно требует реализации всего двух функций — `Area()` и `Perimeter()`, которые возвращают значение типа `float64`. Первая из этих функций будет использоваться для расчета площади плоской фигуры, а вторая — для расчета периметра.

После этого нужно установить пакет `myInterface.go` и сделать его доступным для текущего пользователя. Как вы уже знаете, процесс установки включает в себя выполнение следующих команд UNIX:

```
$ mkdir ~/go/src/myInterface
$ cp myInterface.go ~/go/src/myInterface
$ go install myInterface
```

Использование интерфейса Go

В этом подразделе вы научитесь использовать интерфейс, созданный в `myInterface.go`. Для этого мы рассмотрим Go-программу `useInterface.go`, разделив ее на пять частей.

В первой части `useInterface.go` содержится следующий код Go:

```
package main

import (
    "fmt"
    "math"
    "myInterface"
)

type square struct {
    x float64
}

type circle struct {
    R float64
}
```

Поскольку нужный нам интерфейс определен в отдельном пакете, неудивительно, что мы импортируем этот пакет `myInterface`.

Вторая часть `useInterface.go` содержит следующий код:

```
func (s square) Area() float64 {
    return s.X * s.X
}

func (s square) Perimeter() float64 {
    return 4 * s.X
}
```

В этой части программы реализован интерфейс `shape` для типа `square`.

В третьей части программы содержится следующий код Go:

```
func (s circle) Area() float64 {
    return s.R * s.R * math.Pi
}

func (s circle) Perimeter() float64 {
    return 2 * s.R * math.Pi
}
```

Здесь реализован интерфейс `shape` для типа `circle`.

В четвертой части `useInterface.go` содержится такой код Go:

```
func Calculate(x myInterface.Shape) {
    _, ok := x.(circle)
    if ok {
        fmt.Println("Is a circle!")
    }

    v, ok := x.(square)
    if ok {
        fmt.Println("Is a square:", v)
    }

    fmt.Println(x.Area())
    fmt.Println(x.Perimeter())
}
```

Как видим, в этом коде реализована функция, для которой требуется один параметр типа `shape`, — `myInterface.Shape`. Эта магия станет более очевидной, когда вы поймете, что здесь подойдет любой параметр типа `shape`, то есть любой параметр, тип которого реализует интерфейс `shape`.

Код, представленный в начале функции, позволяет различать типы данных, которые реализуют желаемый интерфейс. Во втором блоке показано, как получить значения, хранящиеся в параметре `square`, — этот способ можно использовать для любого типа, который реализует интерфейс `myInterface.Shape`.

Последний фрагмент кода содержит следующий код:

```
func main() {
    x := square{X: 10}
    fmt.Println("Perimeter:", x.Perimeter())
    Calculate(x)
    y := circle{R: 5}
    Calculate(y)
}
```

В этой части программы показано, что параметрами реализованной ранее функции `Calculate()` могут служить как переменные типа `circle`, так и переменные типа `square`.

Если выполнить `useInterface.go`, то получим следующие результаты:

```
$ go run useInterface.go
Perimeter: 40
Is a square: {10}
100
40
Is a circle!
78.53981633974483
31.41592653589793
```

Использование переключателей для интерфейсов и типов данных

В этом подразделе показано, как использовать оператор `switch` для переключения в зависимости от типов данных. Мы рассмотрим это на примере кода Go из файла `switch.go`, который разделим на четыре части.

Код программы `switch.go` в некоторой степени повторяет `useInterface.go`, однако в нем добавлен еще один тип, `rectangle`, и здесь нет необходимости в реализации методов какого-либо интерфейса.

Первая часть программы выглядит так:

```
package main

import (
    "fmt"
)
```

Поскольку код `switch.go` не работает с интерфейсом, определенным в `myInterface.go`, то нет необходимости импортировать пакет `myInterface`.

Во второй части определены три новых типа данных, которые будут использоваться в программе:

```
type square struct {
    x float64
}
```

```

type circle struct {
    R float64
}

type rectangle struct {
    X float64
    Y float64
}

```

Все эти три типа данных довольно просты.

Третий фрагмент `switch.go` содержит следующий код на Go:

```

func tellInterface(x interface{}) {
    switch v := x.(type) {
    case square:
        fmt.Println("This is a square!")
    case circle:
        fmt.Printf("%v is a circle!\n", v)
    case rectangle:
        fmt.Println("This is a rectangle!")
    default:
        fmt.Printf("Unknown type %T!\n", v)
    }
}

```

Здесь вы видите реализацию функции `tellInterface()` с одним параметром `x` и типом `interface{}`.

Этот прием поможет нам различать разные типы передаваемых функции параметров `x`. Вся магия происходит благодаря оператору `x.(type)`, который возвращает тип элемента `x`. Чтобы получить значение типа, в `fmt.Printf()` используется операция `%v`.

В последней части `switch.go` содержится реализация функции `main()`:

```

func main() {
    x := circle{R: 10}
    tellInterface(x)
    y := rectangle{X: 4, Y: 1}
    tellInterface(y)
    z := square{X: 4}
    tellInterface(z)
    tellInterface(10)
}

```

Выполнение `switch.go` приведет к результатам такого вида:

```

$ go run switch.go
{10} is a circle!
This is a rectangle!
This is a square!
Unknown type int!

```


Рефлексия

Рефлексия — это дополнительная функциональность Go, которая позволяет динамически, в процессе выполнения программы, узнавать тип произвольного объекта, а также получать информацию о его структуре. Для работы с рефлексией в Go создан пакет `reflect`. Советую запомнить одну вещь: едва ли вам придется часто использовать рефлексии при разработке программ на Go. Отсюда сразу следуют два вопроса: зачем нужна рефлексия и когда ее стоит использовать?

Рефлексия необходима для реализации таких пакетов, как `fmt`, `text/template` и `html/template`. В пакете `fmt` рефлексия избавляет нас от необходимости обрабатывать каждый из существующих типов данных по отдельности. Но даже если нам хватило терпения написать код для каждого из известных нам типов данных, мы все равно не смогли бы учесть все возможные типы! В этом случае благодаря рефлексии методы пакета `fmt` могут получить структуру любого типа данных и работать с ним.

Таким образом, рефлексия будет полезной в тех случаях, когда вы хотите обеспечить максимально возможную универсальность или когда хотите гарантировать возможность работы с типами данных, которые не существуют на момент написания кода, но могут появиться в будущем. Кроме того, рефлексия удобна при работе со значениями типов, которые не соответствуют общему интерфейсу.



Как видим, рефлексия помогает работать с неизвестными типами данных и неизвестными значениями типов. Однако за такую гибкость приходится дорого платить.

Звездами пакета `reflect` являются два типа данных — `reflect.Value` и `reflect.Type`. Первый из них используется для хранения значений любого типа, а второй — для представления типов Go.

Простой пример рефлексии

В этом подразделе приводится сравнительно простой пример рефлексии, чтобы вы могли уверенно себя чувствовать с расширенным функционалом Go.

Файл с исходным кодом Go для этого примера называется `reflection.go`. Рассмотрим его, разделив на четыре части. Цель `reflection.go` — исследовать «неизвестную» структурную переменную и получить дополнительную информацию о ней во время выполнения программы. Чтобы было еще интереснее, в программе определены два новых типа `struct`. На основании этих типов мы также добавим две новые переменные, однако исследуем только одну из них.

Если не передать программе аргументы командной строки, то она будет исследовать первую из этих переменных, в противном случае — вторую. На практике это означает, что программа не знает заранее (в процессе выполнения) тип структурной переменной, которую ей следует обработать.

В первой части `reflection.go` содержится следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type a struct {
    X int
    Y float64
    Z string
}

type b struct {
    F int
    G int
    H string
    I float64
}
```

Здесь мы видим определение типов данных `struct`, которые будут использоваться в программе.

Второй фрагмент кода `mirror.go` выглядит так:

```
func main() {
    x := 100
    xRefl := reflect.ValueOf(&x).Elem()
    xType := xRefl.Type()
    fmt.Printf("The type of x is %s.\n", xType)
```

Этот код Go — небольшой, наивный пример отражения. Сначала мы объявляем переменную `x` и вызываем функцию `reflect.ValueOf(&x).Elem()`. Затем мы вызываем `xRefl.Type()`, чтобы получить тип переменной, который хранится в `xType`. В этих трех строках кода показано, как можно получить тип переменной, используя рефлексия. Но если вам нужно узнать только тип переменной, можно просто вызвать `reflect.TypeOf(x)`.

Третья часть `mirror.go` содержит следующий код Go:

```
A := a{100, 200.12, "Struct a"}
B := b{1, 2, "Struct b", -1.2}
var r reflect.Value
```

```
arguments := os.Args
if len(arguments) == 1 {
    r = reflect.ValueOf(&A).Elem()
} else {
    r = reflect.ValueOf(&B).Elem()
}
```

Здесь мы объявляем две переменные — `A` и `B`. Переменная `A` относится к типу `a`, а переменная `B` — к типу `b`. Типом переменной `r` должен быть `reflect.Value`, поскольку функция `reflect.ValueOf()` возвращает значение именно этого типа. Метод `Elem()` возвращает значение, содержащееся в интерфейсе рефлексии (`reflect.Value`).

Последняя часть `reflection.go` выглядит так:

```
iType := r.Type()
fmt.Printf("i Type: %s\n", iType)
fmt.Printf("The %d fields of %s are:\n", r.NumField(), iType)
for i := 0; i < r.NumField(); i++ {
    fmt.Printf("Field name: %s ", iType.Field(i).Name)
    fmt.Printf("with type: %s ", r.Field(i).Type())
    fmt.Printf("and value %v\n", r.Field(i).Interface())
}
}
```

В этой части программы для получения необходимой информации используются соответствующие функции пакета `reflect`. Метод `NumField()` возвращает количество полей в структуре `reflect.Value`, а функция `Field()` возвращает поле структуры, переданной ей в качестве параметра. Функция `Interface()` возвращает значение поля структуры `reflect.Value` в качестве интерфейса.

Если дважды запустить `reflection.go`, то получим следующий результат:

```
$ go run reflection.go 1
The type of x is int.
i Type: main.b
The 4 fields of main.b are:
Field name: F with type: int and value 1
Field name: G with type: int and value 2
Field name: H with type: string and value Struct b
Field name: I with type: float64 and value -1.2
$ go run reflection.go
The type of x is int.
i Type: main.a
The 3 fields of main.a are:
Field name: X with type: int and value 100
Field name: Y with type: float64 and value 200.12
Field name: Z with type: string and value Struct a
```

Важно отметить, что в Go используется свое, внутреннее представление для вывода типов данных переменных `A` и `B` — `main.a` и `main.b` соответственно. Однако к переменной `x`, типом которой является `int`, это не относится.

Более сложный пример рефлексии

В этом подразделе мы рассмотрим более сложные способы использования рефлексии, которые продемонстрируем на сравнительно небольших примерах с использованием кода Go из файла `advRef1.go`.

Мы разделим программу `advRef1.go` на пять частей. Первая часть выглядит так:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type t1 int
type t2 int
```

Обратите внимание, что, хотя оба типа, `t1` и `t2`, основаны на типе `int` и, следовательно, по сути, относятся к одному и тому же типу `int`, для Go это совершенно разные типы. Их внутреннее представление после того, как Go проанализирует код программы, будет `main.t1` и `main.t2` соответственно.

Вторая часть кода `advRef1.go` выглядит так:

```
type a struct {
    X    int
    Y    float64
    Text string
}

func (a1 a) compareStruct(a2 a) bool {
    r1 := reflect.ValueOf(&a1).Elem()
    r2 := reflect.ValueOf(&a2).Elem()
    for i := 0; i < r1.NumField(); i++ {
        if r1.Field(i).Interface() != r2.Field(i).Interface() {
            return false
        }
    }
    return true
}
```

В этом коде определяется тип структуры Go с именем `a` и реализуется функция Go с именем `compareStruct()`. Цель этой функции — выяснить, являются ли две переменные типа `a` абсолютно одинаковыми. Как видите, для выполнения этой задачи в `compareStruct()` используется код Go из программы `mirror.go`.

Третий фрагмент `advRef1.go` содержит следующий код Go:

```
func printMethods(i interface{}) {
    r := reflect.ValueOf(i)
```

```

t := r.Type()
fmt.Printf("Type to examine: %s\n", t)

for j := 0; j < r.NumMethod(); j++ {
    m := r.Method(j).Type()
    fmt.Println(t.Method(j).Name, "-->", m)
}
}

```

Функция `printMethods()` выводит список методов переменной. Для демонстрации применения `printMethods()` в `advRef1.go` будет использоваться переменная типа `os.File`.

Четвертый фрагмент `advRef1.go` содержит следующий код Go:

```

func main() {
    x1 := t1(100)
    x2 := t2(100)
    fmt.Printf("The type of x1 is %s\n", reflect.TypeOf(x1))
    fmt.Printf("The type of x2 is %s\n", reflect.TypeOf(x2))

```

```

    var p struct{}
    r := reflect.New(reflect.ValueOf(&p).Type()).Elem()
    fmt.Printf("The type of r is %s\n", reflect.TypeOf(r))

```

Последняя часть кода из файла `advRef1.go` выглядит так:

```

a1 := a{1, 2.1, "A1"}
a2 := a{1, -2, "A2"}

if a1.compareStruct(a1) {
    fmt.Println("Equal!")
}

if !a1.compareStruct(a2) {
    fmt.Println("Not Equal!")
}

var f *os.File
printMethods(f)
}

```

Как вы вскоре узнаете, вызов `a1.compareStruct(a1)` возвращает `true`, поскольку мы сравниваем переменную `a1` с самой собой, тогда как вызов `a1.compareStruct(a2)` возвращает `false`, так как переменные `a1` и `a2` имеют разные значения.

Выполнение `advRef1.go` приведет к следующим результатам:

```

$ go run advRef1.go
The type of x1 is main.t1
The type of x2 is main.t2
The type of r is reflect.Value
Equal!
Not Equal!

```

```
Type to examine: *os.File
Chdir --> func() error
Chmod --> func(os.FileMode) error
Chown --> func(int, int) error
Close --> func() error
Fd --> func() uintptr
Name --> func() string
Read --> func([]uint8) (int, error)
ReadAt --> func([]uint8, int64) (int, error)
Readdir --> func(int) ([]os.FileInfo, error)
Readdirnames --> func(int) ([]string, error)
Seek --> func(int64, int) (int64, error)
Stat --> func() (os.FileInfo, error)
Sync --> func() error
Truncate --> func(int64) error
Write --> func([]uint8) (int, error)
WriteAt --> func([]uint8, int64) (int, error)
WriteString --> func(string) (int, error)
```

Как видим, типом переменной `r`, которая возвращается функцией `reflect.New()`, является `reflect.Value`. Кроме того, результат метода `printMethods()` говорит о том, что тип `*os.File` поддерживает множество методов, в том числе `Chdir()` и `Chmod()`.

Три недостатка рефлексии

Без сомнения, рефлексия в Go — мощная функциональность. Однако, как и любой другой инструмент, рефлексии следует использовать рационально, и на то есть три основные причины. Первая из них заключается в том, что из-за чересчур интенсивного использования рефлексии код становится трудно читать и поддерживать. Иногда эта проблема решается посредством хорошей документации, но, как известно, разработчикам обычно не хватает времени на написание необходимой документации.

Вторая причина состоит в том, что код Go, в котором используется рефлексия, замедляет выполнение программы. Вообще, код Go, предназначенный для работы с определенным типом данных, всегда будет быстрее, чем код Go, в котором используется рефлексия для динамической работы с любым типом данных. Кроме того, такой динамический код затрудняет работу для инструментов рефакторинга или анализа кода.

Последняя причина — ошибки рефлексии не распознаются на этапе сборки и обнаруживаются только в процессе выполнения, вызывая панику, так что ошибки рефлексии способны привести к сбою программ. Это может случиться через несколько месяцев или даже лет после разработки программы Go! Одно из решений данной проблемы — провести глубокое тестирование, прежде чем использовать вызов опасной функции. Однако это увеличивает количество кода Go в программах, что делает их еще медленнее.

Библиотека reflectwalk

Библиотека `reflewalk` позволяет перебирать сложные значения в Go, используя рефлексии, — подобно тому как мы перебираем файлы в файловой системе. В следующем примере программы `walkRef.go` показано, как перебрать все элементы структуры. Мы рассмотрим этот код, разделив его на пять частей.

Первая часть `walkRef.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/mitchellh/reflectwalk"
    "reflect"
)

type Values struct {
    Extra map[string]string
}
```

Поскольку `reflewalk` не является стандартным пакетом Go, нам нужно вызвать его, указав полный адрес.

Вторая часть `walkRef.go` выглядит так:

```
type WalkMap struct {
    MapVal reflect.Value
    Keys   map[string]bool
    Values map[string]bool
}

func (t *WalkMap) Map(m reflect.Value) error {
    t.MapVal = m
    return nil
}
```

Интерфейс, определенный в `reflewalk`, требует реализации функции `Map()`, которая используется для поиска в хеш-таблицах.

Третья часть `walkRef.go` выглядит так:

```
func (t *WalkMap) MapElem(m, k, v reflect.Value) error {
    if t.Keys == nil {
        t.Keys = make(map[string]bool)
        t.Values = make(map[string]bool)
    }

    t.Keys[k.Interface().(string)] = true
    t.Values[v.Interface().(string)] = true
    return nil
}
```

Четвертая часть `walkRef.go` выглядит следующим образом:

```
func main() {
    w := new(WalkMap)

    type S struct {
        Map map[string]string
    }

    data := &S{
        Map: map[string]string{
            "V1": "v1v",
            "V2": "v2v",
            "V3": "v3v",
            "V4": "v4v",
        },
    }

    err := reflectwalk.Walk(data, w)
    if err != nil {
        fmt.Println(err)
    }
    return
}
```

Здесь мы определяем новую переменную с именем `data`, которая содержит хеш-таблицу, и вызываем функцию `reflectwalk.Walk()`, чтобы ее исследовать.

Последняя часть `walkRef.go` выглядит так:

```
r := w.MapVal
fmt.Println("MapVal:", r)
rType := r.Type()
fmt.Printf("Type of r: %s\n", rType)

for _, key := range r.MapKeys() {
    fmt.Println("key:", key, "value:", r.MapIndex(key))
}
}
```

В последней части `walkRef.go` показано, как использовать рефлексии для вывода на экран содержимого поля `MapVal` структуры `WalkMap`. Метод `MapKeys()` возвращает срез `reflect.Values`, каждое значение которого содержит один из ключей хеш-таблицы. Метод `MapIndex()` позволяет вывести на экран значение ключа. Методы `MapKeys()` и `MapIndex()` работают только с типом `reflect.Map` и позволяют перебирать все значения хеш-таблицы, однако последовательность возвращаемых элементов хеш-таблицы будет случайной.

Прежде чем в первый раз использовать библиотеку `reflectwalk`, необходимо скачать ее, что можно сделать следующим образом:

```
$ go get github.com/mitchellh/reflectwalk
```




Если вы решите использовать Go-модули, то процесс загрузки библиотеки `reflectwalk` будет автоматическим, что гораздо проще.

Выполнение `walkRef.go` приведет к следующим результатам:

```
$ go run walkRef.go
MapVal: map[V1:v1v V2:v2v V3:v3v V4:v4v]
Type of r: map[string]string
key: V2 value: v2v
key: V3 value: v3v
key: V4 value: v4v
key: V1 value: v1v
```

Объектно-ориентированное программирование на Go

Как вы уже знаете, в Go не используется наследование — вместо этого язык поддерживает композицию. Интерфейсы Go обеспечивают своего рода полиморфизм. Таким образом, хотя Go и не является объектно-ориентированным языком, некоторые его возможности позволяют имитировать объектно-ориентированное программирование.



Если вы действительно хотите разрабатывать приложения на основе объектно-ориентированной методологии, то, возможно, Go не лучший вариант для этого. Мне не очень нравится Java, поэтому я бы предложил рассмотреть варианты с C++ или Python.

Прежде всего позвольте показать вам два приема, которые будут применяться в программе Go из этого раздела. В первом из них используются методы, позволяющие ассоциировать функцию с типом. Другими словами, в некотором смысле функция и этот тип образуют объект. Второй прием позволяет встраивать тип в новый структурный тип, чтобы создать своего рода иерархию.

Есть еще третий прием, в котором с помощью интерфейса Go можно создать несколько элементов — *объектов одного и того же класса*. Этот прием не рассмотрен в данном разделе, поскольку он был показан ранее в главе.

Здесь главное, что интерфейс Go позволяет определять общее поведение для нескольких разных элементов, так что все эти элементы имеют общие характеристики объекта. Это позволяет утверждать, что все разные элементы являются объектами одного и того же класса; однако в реальном объектно-ориентированном языке программирования объекты и классы имеют гораздо больше возможностей.

Мы рассмотрим первые два приема на примере программы `ooo.go`, которую разделим на четыре части.

Первый фрагмент `ooo.go` содержит следующий код Go:

```
package main

import (
    "fmt"
)

type a struct {
    XX int
    YY int
}

type b struct {
    AA string
    XX int
}
```

Вторая часть программы выглядит так:

```
type c struct {
    A a
    B b
}
```

Таким образом, композиция позволяет создать структуру из элементов Go, используя несколько типов `struct`. В данном случае в типе данных `C` объединены переменные `a` и `b`.

Третья часть `ooo.go` содержит следующий код Go:

```
func (A a) A() {
    fmt.Println("Function A() for A")
}

func (B b) A() {
    fmt.Println("Function A() for B")
}
```

Здесь определены два метода, которые могут иметь одно и то же имя (`A()`), поскольку у них разные заголовки функций (они определены для разных типов) — первый работает с переменными `a`, второй — с переменными `b`. Этот прием позволяет использовать одно и то же имя функции для нескольких типов.

Последняя часть `ooo.go` выглядит так:

```
func main() {
    var i c
    i.A.A()
    i.B.A()
}
```

Код Go в `ooo.go` очень прост по сравнению с кодом объектно-ориентированного языка программирования, в котором реализованы абстрактные классы и наследование. Однако этого более чем достаточно для создания типов и структурных элементов, а также для использования разных типов данных, имеющих методы с одинаковыми именами.

Выполнение `ooo.go` приведет к следующим результатам:

```
$ go run ooo.go
Function A() for A
Function A() for B
```

Однако, как показывает следующий код, композиция не то же самое, что наследование, и тип `first` ничего не знает об изменениях, добавленных в функцию `shared()` типом `second`:

```
package main

import (
    "fmt"
)

type first struct{}

func (a first) F() {
    a.shared()
}

func (a first) shared() {
    fmt.Println("This is shared() from first!")
}

type second struct {
    first
}

func (a second) shared() {
    fmt.Println("This is shared() from second!")
}

func main() {
    first{}.F()
    second{}.shared()
    i := second{}
    j := i.first
    j.F()
}
```

Обратите внимание, что тип `first` встроен в тип `second`, и эти два типа совместно используют функцию с именем `shared()`.

Если сохранить этот код Go в файле `goCoIn.go` и выполнить его, то получим следующий результат:

```
$ go run goCoIn.go
This is shared() from first!
This is shared() from second!
This is shared() from first!
```

Вызовы `first{}.F()` и `second{}.shared()` приводят к ожидаемым результатам, однако функция `j.F()` по-прежнему вызывает `first.shared()`, а не `second.shared()`. И это несмотря на то, что для типа `second` реализация функции `shared()` была изменена. В объектно-ориентированной терминологии это называется *перепределением метода*.

Обратите внимание, что вызов `j.F()` может быть записан как `(i.first).F()` или как `(second{}.first).F()` без необходимости определять слишком много переменных. Но, чтобы этот код был понятнее, лучше разделить его на три строки.

Основы git и GitHub

GitHub — это веб-сайт и сервис для хранения и сборки программного обеспечения. Для работы с GitHub можно использовать как его графический интерфейс, так и утилиты командной строки. С другой стороны, у нас есть `git(1)` — утилита командной строки, которая позволяет выполнять много задач, включая работу с репозиториями GitHub.



Альтернативой GitHub является GitLab. Большинство, если не все представленные здесь команды и опции `git(1)` будут работать и для связи с GitLab без каких-либо изменений.

В этом разделе представлено краткое описание утилиты `git(1)` и ее основных наиболее часто используемых команд.

Использование git

Обратите внимание, что у `git(1)` огромное количество команд и параметров, но далеко не все из них вам придется использовать ежедневно. В этом подразделе я приведу самые полезные и распространенные команды `git(1)`, исходя из моего опыта и стиля работы.

Для того чтобы перенести существующий репозиторий GitHub на локальный компьютер, нужно использовать команду `git clone`, в которой указать URL-адрес репозитория:

```
$ git clone git@github.com:mactsouk/go-kafka.git
Cloning into 'go-kafka'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (8/8), done.
```

```
remote: Total 13 (delta 4), reused 10 (delta 4), pack-reused 0
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (4/4), done.
```

Команда git status

Команда `git status` показывает состояние рабочего дерева. Если оно полностью синхронизировано, то `git status` возвращает примерно следующее:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

Если же есть несинхронизированные изменения, то результат `git status` будет выглядеть примерно так:

```
On branch master
Your branch is up to date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:
      main.go
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newFile.go
no changes added to commit (use "git add" and/or "git commit -a")
```

Команда git pull

Команда `git pull` используется для получения обновлений из удаленного репозитория. Она особенно полезна, когда с одним репозиторием работают несколько человек или если вы работаете с нескольких компьютеров.

Команда git commit

Команда `git commit` предназначена для записи изменений в репозиторий. После выполнения команды `git commit` обычно нужно выполнить `git push`, чтобы передать изменения в удаленный репозиторий. Типичный способ выполнения команды `git commit` выглядит так:

```
$ git commit -a -m "Commit message"
```

Параметр `-m` указывает на то, что вместе с данными на сервер будет передано сообщение, в параметр `-a` заставляет `git commit` автоматически учесть все измененные файлы. Заметьте, что сюда не входят новые файлы, которые нужно сначала добавить в репозиторий с помощью команды `git add`.

Команда git push

Чтобы перенести локальные изменения в репозиторий GitHub, нужно выполнить команду `git push`. Результат работы выглядит примерно так:

```
$ touch a_file.go
$ git add a_file.go
$ git commit -a -m "Adding a new file"
[master 782c4da] Adding a new file
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 ch07/a_file.go
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 98f8a77..782c4da master -> master
```

Работа с ветками

Ветка — это способ управления рабочим процессом, позволяющий отделить изменения от основной ветки. В каждом репозитории есть главная ветка, используемая по умолчанию, — обычно она называется `master` — и, возможно, несколько других веток.

Чтобы создать на локальном компьютере новую ветку с именем `new_branch` и перейти к ней, нужно использовать следующую команду:

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
```

Если вы хотите подключить эту ветку к GitHub, следует выполнить такую команду:

```
$ git push --set-upstream origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'new_branch' on GitHub by visiting:
remote:
https://github.com/PacktPublishing/Mastering-Go-Second-Edition/pull/new/new_branch
remote:
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 * [new branch]
new_branch -> new_branch
Branch 'new_branch' set up to track remote branch 'new_branch' from 'origin'.
```

Если вы хотите перейти с текущей ветки на основную ветку, то нужно выполнить следующую команду:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

Для того чтобы удалить локальную ветку — в данном случае `new_branch`, — нужно выполнить команду `git branch -D`:

```
$ git --no-pager branch -a
* master
  new_branch
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/new_branch
$ git branch -D new_branch
Deleted branch new_branch (was 98f8a77).
$ git --no-pager branch -a
* master
  remotes/origin/HEAD -> origin/master
  remotes/origin/master
  remotes/origin/new_branch
```

Работа с файлами

Когда вы создаете в репозитории или удаляете оттуда один или несколько файлов, необходимо сообщить `git(1)` об этом. Для того чтобы удалить файл `a_file.go`, нужно сделать следующее:

```
$ rm a_file.go
$ git rm a_file.go
rm 'ch07/a_file.go'
```

Если после этого выполнить `git status`, то получим такой результат:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted:    a_file.go
```

Чтобы изменения вступили в силу, нужно сначала выполнить `git commit`, а затем `git push`:

```
$ git commit -a -m "Deleting a_file.go"
[master 1b06700] Deleting a_file.go
```

```

1 file changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 ch07/a_file.go
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
782c4da..1b06700 master -> master

```

Файл .gitignore

В `.gitignore` содержится список файлов и каталогов, которые следует игнорировать при записи изменений на GitHub. Содержимое файла `.gitignore` может выглядеть, например, так:

```

$ cat .gitignore
public/
.DS_Store
*.swp

```

Обратите внимание: после того как файл `.gitignore` создан, его следует добавить в текущую ветку с помощью команды `git add`.

Команда git diff

Команда `git diff` показывает различия между зафиксированными изменениями и рабочим репозиторием, веткой и т. п.

Следующая команда покажет изменения между вашими файлами и файлами, которые находятся на GitHub (до последней команды `git push`). Именно эти изменения будут добавлены в версию, хранящуюся на GitHub, при выполнении команды `git push`:

```

$ git diff
diff --git a/content/blog/Stats.md b/content/blog/Stats.md
index 0f36b60..af64ec3 100644
--- a/content/blog/Stats.md
+++ b/content/blog/Stats.md
@@ -16,6 +16,8 @@ title: Statistical analysis of random numbers
## Developing a Kafka producer in Go
+Please note that the format of the first record that is written to Kafka
+specifies the format of the subsequent records
### Viewing the data in Lenses

```


Работа с тегами

Тег — это способ пометить определенную версию кода. Тег можно представить как ветку, которая никогда не меняется.

Чтобы создать *легкий тег*, нужно выполнить такую команду:

```
$ git tag c7.0
```

Чтобы получить информацию о конкретном теге, нужно сделать следующее:

```
$ git --no-pager show v1.0.0
commit f415872e62bd71a004b680d50fa089c139359533 (tag: v1.0.0)
Author: Mihalis Tsoukalos <mihalistsoukalos@gmail.com>
Date: Sat Mar 2 20:33:58 2019 +0200
    Initial version 1.0.0
diff --git a/go.mod b/go.mod
new file mode 100644
index 0000000..c4928c5
--- /dev/null
+++ b/go.mod
@@ -0,0 +1,3 @@
+module github.com/mactsouk/myModule
+
+go 1.12
diff --git a/myModule.go b/myModule.go
index e69de29..fa6b0fe 100644
--- a/myModule.go
+++ b/myModule.go
@@ -0,0 +1,9 @@
+package myModule
+
+import (
+    "fmt"
+)
+
+func Version() {
+    fmt.Println("Version 1.0.0")
+}
```

Чтобы получить список всех доступных тегов, можно воспользоваться командой `git tag`:

```
$ git --no-pager tag
c7.0
```

Чтобы внести тег в GitHub, нужно сделать следующее:

```
$ git push origin c7.0
Total 0 (delta 0), reused 0 (delta 0)
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
* [new tag] c7.0 -> c7.0
```

Чтобы удалить существующий тег с локального компьютера, нужно выполнить следующую команду:

```
$ git tag -d c7.0
Deleted tag 'c7.0' (was 1b06700)
```

Можно также удалить тег с сервера GitHub:

```
$ git push origin :refs/tags/c7.0
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
- [deleted]      c7.0
```

Команда git cherry-pick

Команда `git cherry-pick` — это сложная команда, которую нужно использовать осторожно, поскольку она вносит в текущую ветку изменения, зафиксированные некоторыми из последних команд `git commit`. Следующая команда вносит в текущую ветвь фиксацию номер 4226f2c4:

```
$ git cherry-pick 4226f2c4
```

Эта команда вносит в текущую ветку все фиксации с 4226f2c4 по 0d820a87, включая фиксацию номер 4226f2c4:

```
$ git cherry-pick 4226f2c4..0d820a87
```

Данная команда вносит в текущую ветку все фиксации с 4226f2c4 по 0d820a87, включая фиксацию 4226f2c4:

```
$ git cherry-pick 4226f2c4^..0d820a87
```



Представленный список команд и параметров утилиты `git(1)` далеко не полный, однако этого достаточно, чтобы вы смогли работать с `git(1)` и GitHub. Эти знания пригодятся вам при создании модулей Go.

Отладка с помощью Delve

Delve — это текстовый *отладчик* программ Go, также написанный на Go. Чтобы скачать Delve для macOS Mojave, нужно сделать следующее:

```
$ go get -u github.com/go-delve/delve/cmd/dlv
$ ls -l ~/go/bin/dlv
-rwxr-xr-x 1 mtsouk staff 16M Mar 7 09:04 /Users/mtsouk/go/bin/dlv
```

Поскольку Delve зависит от множества модулей и пакетов Go, процесс установки займет некоторое время. Двоичный файл Delve размещается в каталоге `~/go/bin`.

Чтобы узнать информацию о версии отладчика, нужно выполнить команду `dlv version`:

```
$ ~/go/bin/dlv version
Delve Debugger
Version: 1.2.0
Build: $Id: 068e2451004e95d0b042e5257e34f0f08ce01466 $
```

Обратите внимание, что Delve также работает на компьютерах с Linux и Microsoft Windows. Учтите, что Delve — это внешняя программа, то есть для работы Delve не нужно подключать какие-либо пакеты к вашим программам Go.

Поскольку этот раздел представляет собой краткое описание основ Delve, далее мы рассмотрим небольшой пример, который поможет вам начать работу с отладчиком Delve. Приведенная здесь информация раскрывает общие принципы не только Delve, но и почти любого другого отладчика.

Пример отладки

Если путь `~/go/bin` хранится в системной переменной `PATH`, то мы можем вызывать Delve откуда захотим, просто как `dlv`. В противном случае нужно указывать полный путь. В этом разделе я буду использовать полный путь к утилите.

Первая команда Delve, которую следует усвоить, — `debug`. По этой команде Delve компилирует пакет `main` в текущем рабочем каталоге и начинает его отладку. Если в текущем рабочем каталоге нет пакета `main`, то выводится следующее сообщение об ошибке:

```
$ ~/go/bin/dlv debug
go: cannot find main module; see 'go help modules'
exit status 1
```

Перейдем в каталог `./ch07/debug` и отладим настоящую программу, которая хранится в файле `main.go`. Код в `main.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
)

func function(i int) int {
    return i * i
}

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need at least one argument.")
    }
}
```

```

        return
    }

    i := 5
    fmt.Println("Debugging with Delve")
    fmt.Println(i)
    fmt.Println(function(i))

    for arg, _ := range os.Args[1:] {
        fmt.Println(arg)
    }
}

```

Чтобы передать в программу аргументы командной строки, следует запустить Delve так:

```
$ ~/go/bin/dlv debug -- arg1 arg2 arg3
```

Мы выполним Delve следующим образом:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
```

Экранное приглашение Delve, (dlv), открывает строку, в которой можно вводить команды Delve. Если ввести здесь просто `c` (от `continue` — «продолжить»), то программа Go будет выполняться так, как если бы мы ввели команду `go run` из оболочки UNIX:

```
(dlv) c
Debugging with Delve
5
25
0
1
2
Process 57252 has exited with status 0
(dlv)
```

Если снова ввести `c` или `continue`, то получим следующее:

```
(dlv) c
Process 57252 has exited with status 0
```

Так происходит потому, что программа закончила работу и в данный момент ее выполнение не может быть продолжено. Чтобы иметь возможность ее дальнейшей отладки, придется запустить программу заново. Для перезапуска программы нужно ввести `r` или `restart`, что эквивалентно выполнению Delve из оболочки UNIX:

```
(dlv) r
Process restarted with PID 57257
(dlv)
```

Теперь попробуем что-нибудь еще. Мы создадим две точки останова, одну для функции `main()` и вторую — для `function()`:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
(dlv) break main.main
Breakpoint 1 set at 0x10b501b for main.main() ./main.go:12
(dlv) break function
Breakpoint 2 set at 0x10b4fe0 for main.function() ./main.go:8
(dlv)
```

Если теперь ввести `continue`, то отладчик остановит программу на строке `main()` или `function()`, в зависимости от того, какая из них ему встретится раньше. Поскольку это исполняемая программа, то функция `main()` будет первой:

```
(dlv) c
> main.main() ./main.go:12 (hits goroutine(1):1 total:1) (PC: 0x10b501b)
 7:
 8:   func function(i int) int {
 9:       return i * i
10:   }
11:
=> 12:   func main() {
13:       if len(os.Args) == 1 {
14:           fmt.Println("Need at least one argument.")
15:           return
16:       }
17:
(dlv)
```

Двойная стрелка `=>` указывает на строку исходного кода, в которой произошел останов. Если ввести `next`, то отладчик перейдет к следующему оператору Go — команду `next` можно вводить сколько угодно раз, пока мы не достигнем конца программы:

```
(dlv) next
> main.main() ./main.go:13 (PC: 0x10b5032)
 8:   func function(i int) int {
 9:       return i * i
10:   }
11:
12:   func main() {
=> 13:   if len(os.Args) == 1 {
14:       fmt.Println("Need at least one argument.")
15:       return
16:   }
17:
18:   i := 5
(dlv) next
> main.main() ./main.go:18 (PC: 0x10b50d0)
```

```

13:     if len(os.Args) == 1 {
14:         fmt.Println("Need at least one argument.")
15:         return
16:     }
17:
=> 18:     i := 5
19:     fmt.Println("Debugging with Delve")
20:     fmt.Println(i)
21:     fmt.Println(function(i))
22:
23:     for arg, _ := range os.Args[1:] {
(dlv) next
> main.main() ./main.go:19 (PC: 0x10b50db)
14:         fmt.Println("Need at least one argument.")
15:         return
16:     }
17:
18:     i := 5
=> 19:     fmt.Println("Debugging with Delve")
20:     fmt.Println(i)
21:     fmt.Println(function(i))
22:
23:     for arg, _ := range os.Args[1:] {
24:         fmt.Println(arg)
(dlv) print i
5

```

Последняя команда (`print i`) выводит на экран значение переменной `i`. Если после этого ввести `continue`, то мы перейдем к следующей точке останова, если она есть, или к концу программы:

```

(dlv) c
Debugging with Delve
5
> main.function() ./main.go:8 (hits goroutine(1):1 total:1) (PC: 0x10b4fe0)
3:     import (
4:         "fmt"
5:         "os"
6:     )
7:
=> 8:     func function(i int) int {
9:         return i * i
10:    }
11:
12:     func main() {
13:         if len(os.Args) == 1 {

```

В данном случае следующей точкой останова является функция `function()`, как было определено ранее.

Обратите внимание, что для отладки Go-тестов следует использовать команду `dlv test`, и отладчик Delve сам позаботится об остальном.

Дополнительные ресурсы

Следующие ресурсы будут для вас очень полезны:

- ❑ посетите страницу документации стандартного Go-пакета `reflect`, расположенную по адресу <https://golang.org/pkg/reflect/>. У этого пакета гораздо больше возможностей, чем описано в данной главе;
- ❑ GitHub: <https://github.com/>;
- ❑ GitLab: <https://gitlab.com/>;
- ❑ подробнее о Delve читайте здесь: <https://github.com/go-delve/delve>;
- ❑ больше о библиотеке `reflectwalk`, созданной Митчеллом Хашимото (Mitchell Hashimoto), вы узнаете на странице <https://github.com/mitchellh/reflectwalk>. Изучая его код, вы лучше поймете, что такое рефлексия.

Упражнения

- ❑ Напишите собственный интерфейс и используйте его в отдельной программе Go. В чем польза вашего интерфейса?
- ❑ Напишите интерфейс для вычисления объема трехмерных фигур, таких как кубы и сферы.
- ❑ Напишите интерфейс для вычисления длины отрезков и расстояния между двумя точками на плоскости.
- ❑ Исследуйте рефлексия, используя собственный пример.
- ❑ Как работает рефлексия с хеш-таблицами Go?
- ❑ Если вы хорошо разбираетесь в математике, попробуйте написать интерфейс, который бы реализовал четыре основные математические операции для действительных и комплексных чисел. Не используйте стандартные типы Go `complex64` и `complex128`, а определите собственную структуру для поддержки комплексных чисел.

Резюме

В этой главе вы познакомились с процессом отладки, `git(1)`, GitHub и интерфейсами, которые похожи на контракты. Вы также узнали о методах типа, операциях утверждения типа и рефлексии в Go. Несмотря на то что рефлексия — очень мощное свойство Go, она способна замедлить работу ваших программ Go, поскольку вводит новый уровень сложности на этапе выполнения программы. Кроме того, если неосторожно использовать рефлексия, программа Go может прерваться с ошибкой.

Вы также узнали о том, как писать код Go, который следует принципам объектно-ориентированного программирования. Если вы собираетесь запомнить только что-то одно из этой главы, то следует учитывать, что Go не является объектно-ориентированным языком программирования, но может имитировать некоторые функциональные возможности, предлагаемые объектно-ориентированными языками, такими как Java и C++. Другими словами, если вы планируете постоянно разрабатывать программное обеспечение на основе объектно-ориентированной парадигмы, лучше выбрать вместо Go другой язык программирования. Впрочем, объектно-ориентированное программирование не является панацеей и можно создать лучшую, более ясную и надежную структуру программы, выбрав такой язык программирования, как Go!

Возможно, в этой главе было больше теории, чем вы ожидали, зато следующая глава вознаградит ваше терпение; она посвящена системному программированию на Go. В ней рассмотрены вопросы файлового ввода/вывода, работа с системными файлами UNIX, обработка сигналов UNIX и поддержка каналов в UNIX.

В следующей главе также рассказано об использовании пакетов `flag` и `viper` для поддержки инструментов командной строки с несколькими аргументами и параметрами, а также о пакете `cobra`, разрешениях на доступ к файлам UNIX и некоторых расширенных возможностях использования функций, предлагаемых стандартным Go-пакетом `syscall`. Если вы действительно интересуетесь системным программированием на Go, обратитесь к моей книге *Go Systems Programming*, в которой этот вопрос обсуждается более подробно.

8

Как объяснить UNIX-системе, что она должна делать

В предыдущей главе рассмотрены две актуальные, но несколько теоретические темы Go: интерфейсы и рефлексия. Код Go, с которым вы познакомитесь в этой главе, едва ли можно назвать теоретическим!

Тема данной главы — *системное программирование*. Ведь, в конце концов, Go — это полноценный язык системного программирования, появившийся в результате разочарования. Его создатели были недовольны выбором языков программирования для создания системного программного обеспечения, который существовал на тот момент, и поэтому решили создать новый язык программирования.



В этой главе содержится ряд интересных и в некоторой степени узких тем, которые не были включены в книгу *Go Systems Programming* (Packt Publishing, 2017).

В этой главе рассмотрены следующие темы:

- ☐ процессы UNIX;
- ☐ пакет `flag`;
- ☐ пакет `viper`;
- ☐ пакет `cobra`;
- ☐ использование интерфейсов `io.Reader` и `io.Writer`;
- ☐ обработка сигналов UNIX средствами Go с помощью пакета `os/signal`;
- ☐ поддержка каналов UNIX в системных утилитах UNIX;
- ☐ разработка клиентов Go для Docker;
- ☐ чтение текстовых файлов;
- ☐ чтение файлов в формате CSV;

- ❑ запись файлов;
- ❑ пакет `bytes`;
- ❑ расширенные возможности использования пакета `syscall`;
- ❑ полномочия доступа к файлам UNIX.

О процессах в UNIX

Процесс — это, с одной стороны, среда выполнения, в которой содержатся инструкции, пользовательские данные и части системных данных, а также другие типы ресурсов, получаемые во время выполнения. С другой стороны, *программа* — это двоичный файл, в котором содержатся инструкции и данные, используемые для инициализации инструкций и частей пользовательских данных процесса. Каждый действующий процесс UNIX имеет уникальный идентификатор — целое число без знака, которое называется *идентификатором процесса*.

Существует три категории процессов: *пользовательские процессы*, *процессы-демоны* и *процессы ядра*. Пользовательские процессы выполняются в пространстве пользователя и обычно не имеют специальных прав доступа. Процессы-демоны — это программы, которые размещаются в пользовательском пространстве и работают в фоновом режиме, без необходимости использования терминала. Процессы ядра выполняются только в пространстве ядра и имеют полный доступ ко всем структурам данных ядра.



С-стиль создания новых процессов заключается в системном вызове функции `fork()`. Значение, возвращаемое `fork()`, позволяет программисту различать родительский и дочерний процессы. В Go, наоборот, такая функциональность не поддерживается, вместо нее используются горутины.

Пакет flag

Флаги — это строки, отформатированные специальным образом, которые передаются в программу при запуске для управления ее поведением. Работать с флагами напрямую может оказаться очень сложно, особенно если вы хотите поддерживать несколько флагов. Поэтому, если вы разрабатываете утилиты командной строки UNIX, пакет `flag` будет вам очень интересен и полезен.

Пакет `flag` не делает никаких предположений о последовательности аргументов и параметров командной строки; он выводит полезные сообщения в случае возникновения ошибки при выполнении утилиты командной строки.



Главное преимущество пакета `flag` заключается в том, что он является частью стандартной библиотеки Go, следовательно, он тщательно протестирован и отлажен.

Мы рассмотрим две программы Go, в которых используется пакет `flag`: одну проще, а вторую — сложнее. Первая программа называется `simpleFlag.go`. Разделим ее на четыре части. Программа `simpleFlag.go` распознает два аргумента командной строки — логический и целочисленный.

В первой части `simpleFlag.go` содержится следующий код Go:

```
package main

import (
    "flag"
    "fmt"
)
```

Вторая часть кода `simpleFlag.go` выглядит так:

```
func main() {
    minusK := flag.Bool("k", true, "k flag")
    minus0 := flag.Int("0", 1, "0")
    flag.Parse()
```

Оператор `flag.Bool("k", true, "k flag")` определяет логический аргумент командной строки с именем `k`, значение которого по умолчанию равно `true`. Последний параметр оператора — это строка использования, которая будет отображаться вместе с другой информацией об использовании программы. Аналогично функция `flag.Int()` добавляет в программу поддержку целочисленного аргумента командной строки.



После определения аргументов командной строки, которые поддерживает программа, всегда нужно вызывать `flag.Parse()`.

В третьей части программы `simpleFlag.go` содержится следующий код Go:

```
valueK := *minusK
value0 := *minus0
value0++
```

В этом коде Go показано, как можно получить значения аргументов командной строки. К счастью, пакет `flag` автоматически преобразует входные данные, связанные с флагом `flag.Int()`, в целочисленное значение и нам не нужно делать это самостоятельно. Кроме того, пакет `flag` следит, чтобы аргументу было присвоено допустимое целочисленное значение.

Остальной код Go в программе `simpleFlag.go` выглядит так:

```
    fmt.Println("-k:", valueK)
    fmt.Println("-O:", valueO)
}
```

Теперь, получив значения нужных параметров, мы готовы их использовать.

Взаимодействие с `simpleFlag.go` приводит к результатам такого вида:

```
$ go run simpleFlag.go -O 100
-k: true
-O: 101
$ go run simpleFlag.go -O=100
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k false
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k=false
-k: false
-O: 101
```

Если при выполнении файла `simpleFlag.go` возникает ошибка, то на экран выводится сообщение следующего вида от пакета `flag`:

```
$ go run simpleFlag.go -O=notAnInteger
invalid value "notAnInteger" for flag -O: parse error
Usage of /var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build593534621/b001/exe/simpleFlag:
  -O int
      0 (default 1)
  -k   flag (default true)
exit status 2
```

Обратите внимание на удобное сообщение, которое выводится автоматически, если пользователь ошибочно задаст аргументы командной строки, определенные для данной программы.

Теперь пора представить более реалистичную, расширенную программу, использующую пакет `flag`. Эта программа называется `funWithFlag.go`. Рассмотрим ее, разделив на пять частей. Утилита `funWithFlag.go` распознает разные типы аргументов, в том числе такой, который принимает несколько значений, разделенных запятыми. Кроме того, на примере этой программы вы увидите, как можно получить доступ к аргументам командной строки, которые расположены в конце строки запуска исполняемого файла и не соответствуют ни одному из возможных вариантов.

Функция `flag.Var()`, используемая в `funWithFlag.go`, создает флаг любого типа, который соответствует интерфейсу `flag.Value`. Этот интерфейс определяется следующим образом:

```
type Value interface {
    String() string
    Set(string) error
}
```

В первой части `funWithFlag.go` содержится следующий код Go:

```
package main
```

```
import (
    "flag"
    "fmt"
    "strings"
)
```

```
type NamesFlag struct {
    Names []string
}
```

Структуру `NamesFlag` мы вскоре используем для интерфейса `flag.Value`.

Вторая часть `funWithFlag.go` выглядит так:

```
func (s *NamesFlag) GetNames() []string {
    return s.Names
}
```

```
func (s *NamesFlag) String() string {
    return fmt.Sprint(s.Names)
}
```

Третья часть `funWithFlag.go` содержит следующий код:

```
func (s *NamesFlag) Set(v string) error {
    if len(s.Names) > 0 {
        return fmt.Errorf("Cannot use names flag more than once!")
    }

    names := strings.Split(v, ",")
    for _, item := range names {
        s.Names = append(s.Names, item)
    }
    return nil
}
```

Прежде всего здесь метод `Set()` гарантирует, что соответствующему аргументу командной строки еще не присвоено значение. Затем этот метод получает входные

данные и разделяет их на отдельные аргументы с помощью функции `strings.Split()`. После этого аргументы сохраняются в поле `Names` структуры `NamesFlag`.

Четвертый фрагмент `funWithFlag.go` содержит следующий код Go:

```
func main() {
    var manyNames NamesFlag
    minusK := flag.Int("k", 0, "An int")
    minusO := flag.String("o", "Mihalis", "The name")
    flag.Var(&manyNames, "names", "Comma-separated list")

    flag.Parse()
    fmt.Println("-k:", *minusK)
    fmt.Println("-o:", *minusO)
```

Последняя часть утилиты `funWithFlag.go` выглядит так:

```
for i, item := range manyNames.GetNames() {
    fmt.Println(i, item)
}

fmt.Println("Remaining command line arguments:")
for index, val := range flag.Args() {
    fmt.Println(index, ":", val)
}
}
```

Срез `flag.Args()` содержит оставшиеся аргументы командной строки, а переменная `manyNames` — значения аргументов командной строки из `flag.Var()`.

Выполнение кода из `funWithFlag.go` приводит к результатам такого вида:

```
$ go run funWithFlag.go -names=Mihalis,Jim,Athina 1 two Three
-k: 0
-o: Mihalis
0 Mihalis
1 Jim
2 Athina
Remaining command line arguments:
0 : 1
1 : two
2 : Three
$ go run funWithFlag.go -Invalid=Marietta 1 two Three
flag provided but not defined: -Invalid
Usage of funWithFlag:
-k int
    An int
-names value
    Comma-separated list
-o string
    The name (default "Mihalis")
```

```
exit status 2
$ go run funWithFlag.go -names=Marietta -names=Mihalis
invalid value "Mihalis" for flag -names: Cannot use names flag more than
once!
Usage of funWithFlag:
  -k int
      An int
  -names value
      Comma-separated list
  -o string
      The name (default "Mihalis")
exit status 2
```



Вам, скорее всего, не избежать использования пакета Go для обработки аргументов командной строки вашей программы, если только вы не разрабатываете простейшую утилиту командной строки, которая не требует аргументов.

Пакет viper

Пакет `viper` — мощный Go-пакет, который поддерживает большое количество опций. Все проекты `viper` соответствуют единому шаблону. Сначала нужно инициализировать `viper`, а затем определить интересующие вас элементы. После этого вы получаете доступ к этим элементам, можете читать и использовать их значения. Обратите внимание, что пакет `viper` способен полностью заменить пакет `flag`.

Нужные значения можно получить либо напрямую, как при использовании пакета `flag` стандартной библиотеки Go, так и косвенно, с помощью файлов конфигурации. При использовании файлов конфигурации, в которых свойства представлены в формате JSON, YAML, TOML, HCL или Java, `viper` автоматически выполняет весь синтаксический анализ, что избавляет вас от необходимости писать и отлаживать большой объем кода Go. Пакет `viper` также позволяет извлекать значения из структур Go и сохранять их в этих структурах. Однако для этого необходимо, чтобы поля структуры Go соответствовали ключам файла конфигурации.

Домашняя страница пакета `viper` находится на GitHub (<https://github.com/spf13/viper>). Обратите внимание, что вы не обязаны использовать все возможности `viper` в своих утилитах — применяйте только те из них, которые вам действительно нужны. Общее правило заключается в том, чтобы использовать те функции `viper`, которые упрощают код. Иначе говоря, если вашей утилите требуется слишком много аргументов и флагов командной строки, то лучше использовать файл конфигурации.

Простой пример использования viper

Прежде чем перейти к более сложным примерам, мы рассмотрим код Go, в котором используется viper. Программа называется `usingViper.go`. Разделим ее на три части.

Первая часть `useViper.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

Вторая часть `useViper.go` выглядит следующим образом:

```
func main() {
    viper.BindEnv("GOMAXPROCS")
    val := viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
    viper.Set("GOMAXPROCS", 10)
    val = viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
}
```

Последняя часть `useViper.go` содержит такой код Go:

```
viper.BindEnv("NEW_VARIABLE")
val = viper.Get("NEW_VARIABLE")
if val == nil {
    fmt.Println("NEW_VARIABLE not defined.")
    return
}
fmt.Println(val)
}
```

Цель этой программы — показать, как читать и изменять переменные среды, используя viper. Пакет `flag` не предоставляет такой функциональности — это позволяет делать стандартный Go-пакет `os`, однако не так просто, как viper.

Для начала необходимо загрузить пакет viper. Если вы не используете Go-модули, это можно сделать следующим образом:

```
$ go get -u github.com/spf13/viper
```

Если вы используете Go-модули, то Go автоматически загрузит viper при первой попытке запустить программу Go, в которой применяется этот пакет.

Выполнение `useViper.go` приведет к результатам такого вида:

```
$ go run useViper.go
GOMAXPROCS: <nil>
GOMAXPROCS: 10
NEW_VARIABLE not defined.
```


От flag к viper

Возможно, у вас уже есть программа Go, в которой используется пакет `flag` и которую вы хотели бы перевести на пакет `viper`. Например, рассмотрим следующий код Go, где используется пакет `flag`:

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    minusI := flag.Int("i", 100, "i parameter")
    flag.Parse()
    i := *minusI
    fmt.Println(i)
}
```

В новой версии этой программы, которую мы сохраним в файле `flagToViper.go`, будет использоваться пакет `viper`. Мы рассмотрим эту программу, разделив ее на три части. Первая часть `flagToViper.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "github.com/spf13/pflag"
    "github.com/spf13/viper"
)
```

Для того чтобы `viper` мог работать с аргументами командной строки, нам нужно импортировать пакет `pflag`.

Вторая часть `flagToViper.go` содержит следующий код Go:

```
func main() {
    flag.Int("i", 100, "i parameter")
    pflag.CommandLine.AddGoFlagSet(flag.CommandLine)
    pflag.Parse()
```

Как видим, здесь по-прежнему используется `flag.Int()`, чтобы изменить как можно меньше кода, но для синтаксического анализа вызывается `pflag.Parse()`. Однако магия заключается в вызове `pflag.CommandLine.AddGoFlagSet(flag.CommandLine)`: именно этот вызов импортирует данные из пакета `flag` в пакет `pflag`.

Последняя часть `flagToViper.go` выглядит так:

```
    viper.BindPFlags(pflag.CommandLine)
    i := viper.GetInt("i")
    fmt.Println(i)
}
```

Осталось вызвать последнюю функцию — `viper.BindPFlags()`, и теперь можно получить значение целочисленного аргумента командной строки с помощью функции `viper.GetInt()`. Для иных типов данных придется вызывать другие функции `viper`.

Здесь может понадобиться загрузить в программу `flagToViper.go` Go-пакет `pflag`. Это можно сделать так:

```
$ go get -u github.com/spf13/pflag
```

Выполнение `flagToViper.go` приведет к результатам следующего вида:

```
$ go run flagToViper.go
100
$ go build flagToViper.go
$ ./flagToViper -i 0
0
$ ./flagToViper -i abcd
invalid argument "abcd" for "-i, --i" flag: parse error
Usage of ./flagToViper:
  -i, --i int i parameter
invalid argument "abcd" for "-i, --i" flag: parse error
```

Если по какой-либо причине программе `flagToViper.go` был передан неизвестный аргумент командной строки, `viper` выдаст в ответ сообщение об ошибке:

```
$ ./flagToViper -j 200
unknown shorthand flag: 'j' in -j
Usage of ./flagToViper:
  -i, --i int i parameter (default 100)
unknown shorthand flag: 'j' in -j
exit status 2
```

Чтение конфигурационных файлов в формате JSON

В этом подразделе показано, как с помощью пакета `viper` можно читать конфигурационные файлы в формате JSON. Для этого мы рассмотрим утилиту `readJSON.go`, которую разделим на три части. Первая часть `readJSON.go` выглядит так:

```
package main
```

```
import (
    "fmt"
    "github.com/spf13/viper"
)
```

Во второй части `readJSON.go` содержится следующий код:

```
func main() {
    viper.SetConfigType("json")
    viper.SetConfigFile("./myJSONConfig.json")
    fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
    viper.ReadInConfig()
```

Здесь выполняется синтаксический анализ конфигурационного файла. Обратите внимание, что имя конфигурационного JSON-файла в `readJSON.go` жестко закодировано с помощью вызова функции `viper.SetConfigFile("./myJSONConfig.json")`.

Последняя часть `readJSON.go` выглядит так:

```
if viper.IsSet("item1.key1") {
    fmt.Println("item1.key1:", viper.Get("item1.key1"))
} else {
    fmt.Println("item1.key1 not set!")
}

if viper.IsSet("item2.key3") {
    fmt.Println("item2.key3:", viper.Get("item2.key3"))
} else {
    fmt.Println("item2.key3 is not set!")
}

if !viper.IsSet("item3.key1") {
    fmt.Println("item3.key1 is not set!")
}
}
```

Именно здесь программа проверяет значения конфигурационного JSON-файла, чтобы выяснить, есть ли там нужные ключи.

Содержимое файла `myJSONConfig.json` выглядит так:

```
{
  "item1": {
    "key1": "val1",
    "key2": false,
    "key3": "val3",
  },
  "item2": {
    "key1": "val1",
    "key2": true,
    "key3": "val3",
  }
}
```

Выполнение кода из `readJSON.go` приведет к результатам следующего вида:

```
$ go run readJSON.go
Using config: ./myJSONConfig.json
item1.key1: val1
item2.key3: val3
item3.key1 is not set!
```

Если программа не находит `myJSONConfig.json`, то она не сообщит об ошибке, а будет действовать так, как будто прочитала пустой конфигурационный файл:

```
$ mv myJSONConfig.json ..
$ go run readJSON.go
```

```
Using config: ./myJSONConfig.json
item1.key1 not set!
item2.key3 is not set!
item3.key1 is not set!
```

Чтение конфигурационных файлов в формате YAML

YAML — это еще один популярный текстовый формат, который используется для конфигурационных файлов. В этом подразделе показано, как читать конфигурационные файлы формата YAML с помощью пакета `viper`. На этот раз имя конфигурационного YAML-файла будет передаваться утилите в виде аргумента командной строки. Кроме того, в утилите будет использоваться функция `viper.AddConfigPath()` для добавления трех путей поиска, по которым `viper` будет автоматически искать конфигурационные файлы. Утилита, которую мы рассмотрим, называется `readYAML.go`. Разделим ее на четыре части.

Первая часть `readYAML.go` выглядит так:

```
package main
```

```
import (
    "fmt"
    flag "github.com/spf13/pflag"
    "github.com/spf13/viper"
    "os"
)
```

В преамбуле программы определяется псевдоним (`flag`) для Go-пакета `pflag`. Вторая часть `readYAML.go` содержит следующий код Go:

```
func main() {
    var configFile *string = flag.String("c", "myConfig",
        "Setting the configuration file")
    flag.Parse()

    _, err := os.Stat(*configFile)

    if err == nil {
        fmt.Println("Using User Specified Configuration file!")
        viper.SetConfigFile(*configFile)
    } else {
        viper.SetConfigName(*configFile)
        viper.AddConfigPath("/tmp")
        viper.AddConfigPath("$HOME")
        viper.AddConfigPath(".")
    }
}
```

Этот код проверяет, существует ли файл, указанный в значении флага конфигурации (`--c`), используя вызов `os.Stat()`. Если указанный файл существует,

то будет использован предоставленный файл; в противном случае — имя файла конфигурации, предлагаемое по умолчанию (`myConfig`). Обратите внимание, что мы не указываем точно, что хотим использовать конфигурационный файл в формате YAML, — программа будет искать все поддерживаемые форматы файлов при условии, что имя файла без расширения — `myConfig`. Именно так работает `viper`. Для поиска конфигурационных файлов будут использоваться три пути в следующем порядке: `/tmp`, домашний каталог текущего пользователя и текущий рабочий каталог.

Вообще говоря, использовать каталог `/tmp` для хранения конфигурационных файлов не рекомендуется — прежде всего потому, что содержимое каталога `/tmp` автоматически удаляется после каждой перезагрузки системы; здесь этот каталог используется только для простоты.

Использование `viper.ConfigFileUsed()` совершенно оправданно, поскольку конфигурационный файл не задан жестко, следовательно, нам придется определять его самостоятельно.

Третья часть `readYAML.go` выглядит так:

```
err = viper.ReadInConfig()
if err != nil {
    fmt.Printf("%v\n", err)
    return
}
fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
```

Чтение и синтаксический анализ YAML-файла выполняются с помощью вызова функции `viper.ReadInConfig()`.

Последняя часть `readYAML.go` выглядит следующим образом:

```
if viper.IsSet("item1.k1") {
    fmt.Println("item1.val1:", viper.Get("item1.k1"))
} else {
    fmt.Println("item1.k1 not set!")
}

if viper.IsSet("item1.k2") {
    fmt.Println("item1.val2:", viper.Get("item1.k2"))
} else {
    fmt.Println("item1.k2 not set!")
}

if !viper.IsSet("item3.k1") {
    fmt.Println("item3.k1 is not set!")
}
}
```

Именно здесь программа проверяет содержимое проанализированного конфигурационного файла, чтобы выяснить, существуют ли там нужные ключи.

Содержимое `myConfig.yaml` выглядит так:

```
item1:
  k1:
    - true
  k2:
    - myValue
```

Выполнение `readYAML.go` приведет к результатам следующего вида:

```
$ go build readYAML.go
$ ./readYAML
Using config: /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch08/viper/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
$ mv myConfig.yaml /tmp
$ ./readYAML
Using config: /tmp/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
```

Пакет cobra

Пакет `cobra` — очень удобный и популярный Go-пакет, позволяет разрабатывать утилиты командной строки, имеющие команды, подкоманды и псевдонимы. Если вам когда-либо приходилось использовать `hugo`, `docker` или `kubectl`, то вы сразу поймете, что делает `cobra`, поскольку все эти инструменты разработаны с использованием данного пакета.

Как вы скоро увидите, команды в `cobra` могут иметь один или несколько псевдонимов. Это очень удобно, если вы хотите угодить как любителям, так и опытным пользователям. Пакет `cobra` также поддерживает постоянные и локальные флаги — соответственно флаги, доступные для всех команд, и флаги, доступные только для одной команды. Для синтаксического анализа аргументов командной строки в `cobra` по умолчанию используется пакет `viper`.

Все проекты с использованием `cobra` создаются по одному шаблону. Сначала используется инструмент `cobra`, затем создаются команды, а потом вносятся необходимые изменения в сгенерированные файлы исходного кода Go, чтобы реализовать желаемую функциональность. В зависимости от сложности создаваемой утилиты таких изменений может потребоваться много. Несмотря на то что `cobra` экономит много времени, вам все равно придется писать код, который реализует желаемую функциональность.

Если хотите установить двоичную версию утилиты, вы всегда можете выполнить команду `go install` из любого каталога проекта `cobra`. По умолчанию двоичный исполняемый файл будет размещен в каталоге `~/go/bin`.

Домашняя страница пакета `cobra` находится на GitHub (<https://github.com/spf13/cobra>).

Простой пример cobra

В этом разделе реализуем простую утилиту командной строки, использующую `cobra` и инструмент `~/go/bin/cobra`, который поставляется в комплекте с пакетом. Если выполнить команду `~/go/bin/cobra` без аргументов командной строки, то получим следующий результат:

```
$ ~/go/bin/cobra
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.
Usage:
  cobra [command]
Available Commands:
  add      Add a command to a Cobra Application
  help     Help about any command
  init     Initialize a Cobra Application
Flags:
  -a, --author string    author name for copyright attribution
                        (default "YOUR NAME")
      --config string    config file (default is $HOME/.cobra.yaml)
  -h, --help             help for cobra
  -l, --license string   name of license for the project
      --viper            use Viper for configuration (default true)
Use "cobra [command] --help" for more information about a command.
```

Теперь, получив эту информацию, мы создадим новый проект `cobra`:

```
$ ~/go/bin/cobra init cli
Your Cobra application is ready at
/Users/mtsouk/go/src/cli
Give it a try by going there and running `go run main.go`.
Add commands to it by running `cobra add [cmdname]`.
$ cd ~/go/src/cli
$ ls -l
total 32
-rw-r--r--  1 mtsouk  staff   11358 Mar 13 09:51 LICENSE
drwxr-xr-x  3 mtsouk  staff     96 Mar 13 09:51 cmd
-rw-r--r--  1 mtsouk  staff    669 Mar 13 09:51 main.go
$ ~/go/bin/cobra add cmdOne
cmdOne created at /Users/mtsouk/go/src/cli/cmd/cmdOne.go
$ ~/go/bin/cobra add cmdTwo
cmdTwo created at /Users/mtsouk/go/src/cli/cmd/cmdTwo.go
```

По команде `cobra init` в каталоге `~/go/src` создается новый проект `cobra`, название которого соответствует последнему аргументу команды (`cli`). По команде `cobra add` в инструмент командной строки добавляется новая команда и создаются все необходимые файлы и функции Go для выполнения этой команды.

Таким образом, при каждом запуске команды `cobra add` Cobra выполняет за нас большую часть грязной работы. Однако вам все равно придется реализовать функциональность добавленных команд — в данном случае это команды `cmdOne` и `cmdTwo`. Команда `cmdOne` принимает локальный флаг командной строки с именем `number` — чтобы эта функция работала, потребуется написать дополнительный код.

Теперь, если выполнить команду `go run main.go`, то получим следующий результат:

```
A longer description that spans multiple lines and likely contains
examples and usage of using your application. For example:
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.
```

Usage:

```
cli [command]
```

Available Commands:

```
cmdOne    A brief description of your command
cmdTwo    A brief description of your command
help      Help about any command
```

Flags:

```
--config string  config file (default is $HOME/.cli.yaml)
-h, --help      help for cli
-t, --toggle    Help message for toggle
```

Use "cli [command] --help" for more information about a command.

Код Go для реализации команды `cmdOne` находится в файле `./cmd/cmdOne.go`, а для команды `cmdTwo` — в файле `./cmd/cmdTwo.go`.

Окончательная версия `./cmd/cmdOne.go` выглядит так:

```
package cmd
```

```
import (
    "fmt"
    "github.com/spf13/cobra"
)
```

```
// cmdOneCmd represents the cmdOne command
var cmdOneCmd = &cobra.Command{
    Use: "cmdOne",
    Short: "A brief description of your command",
    Long: `A longer description that spans multiple lines and likely
contains examples
and usage of using your command. For example:
```


Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.`

```
Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("cmdOne called!")
    number, _ := cmd.Flags().GetInt("number")
    fmt.Println("Going to use number", number)
    fmt.Printf("Square: %d\n", number*number)
},
}

func init() {
    rootCmd.AddCommand(cmdOneCmd)
    cmdOneCmd.Flags().Int("number", 0, "A help for number")
}
```

Данный код Go не содержит комментариев, которые автоматически генерируются пакетом `cobra`. В следующей строке кода из функции `init()` определяется новый флаг командной строки:

```
cmdOneCmd.Flags().Int("number", 0, "A help for number")
```

Этот флаг называется `number` и используется в блоке `cobra.Command` следующим образом:

```
number, _ := cmd.Flags().GetInt("number")
```

Теперь мы можем использовать переменную `number` любым удобным способом.

Окончательная версия кода Go из файла `./cmd/cmdTwo.go`, которая включает в себя комментарии и информацию о лицензии, выглядит так:

```
// Copyright © 2019 NAME HERE <EMAIL ADDRESS>
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
// http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the license is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the license.
```

```
package cmd
```

```
import (
    "fmt"
    "github.com/spf13/cobra"
)
```

```
// cmdTwoCmd represents the cmdTwo command
var cmdTwoCmd = &cobra.Command{
    Use: "cmdTwo",
    Short: "A brief description of your command",
    Long: `A longer description that spans multiple lines and likely
contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.` ,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("cmdTwo called!")
    },
}

func init() {
    rootCmd.AddCommand(cmdTwoCmd)

    // Here you will define your flags and configuration settings.

    // Cobra supports Persistent Flags which will work for this command
    // and all subcommands, e.g.:
    // cmdTwoCmd.PersistentFlags().String("foo", "", "A help for foo")

    // Cobra supports local flags which will only run when this command
    // is called directly, e.g.:
    // cmdTwoCmd.Flags().BoolP("toggle", "t", false, "Help message for toggle")
}
```

Это реализация по умолчанию команды `cmdTwo`, сгенерированной пакетом `cobra`.

При выполнении команды `cli` получим результат следующего вида:

```
$ go run main.go cmdOne
cmdOne called!
Going to use number 0
Square: 0
$ go run main.go cmdOne --number -20
cmdOne called!
Going to use number -20
Square: 400
$ go run main.go cmdTwo
cmdTwo called!
```

Если предоставить инструменту неверные входные данные, то он выдаст такие сообщения об ошибках:

```
$ go run main.go cmdThree
Error: unknown command "cmdThree" for "cli"
Run 'cli --help' for usage.
```

```

unknown command "cmdThree" for "cli"
exit status 1
$ go run main.go cmdOne --n -20
Error: unknown flag: --n
Usage:
  cli cmdOne [flags]
Flags:
  -h, --help            help for cmdOne
      --number int       A help for number
Global Flags:
      --config string    config file (default is $HOME/.cli.yaml)
unknown flag: --n
exit status 1

```

Автоматически генерируемое справочное сообщение для команды `cmdOne` выглядит так:

```

$ go run main.go cmdOne --help
A longer description that spans multiple lines and likely contains examples
and usage of using your command. For example:
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.
Usage:
  cli cmdOne [flags]
Flags:
  -h, --help            help for cmdOne
      --number int       A help for number
Global Flags:
      --config string    config file (default is $HOME/.cli.yaml)

```

Структуру каталогов и файлов окончательной версии утилиты можно представить следующим образом с помощью команды `tree(1)`:

```

$ tree
.
├── LICENSE
├── cmd
│   ├── cmdOne.go
│   ├── cmdTwo.go
│   └── root.go
└── main.go
1 directory, 5 files

```

Создание псевдонимов команд

В этом подразделе показано, как с помощью `cobra` создавать псевдонимы для уже существующих команд.

Как и прежде, сначала нам нужно создать новый проект `cobra`, который в данном случае будет называться `aliases`, а также желаемые команды. Это можно сделать так:

```
$ ~/go/bin/cobra init aliases
$ cd ~/go/src/aliases
$ ~/go/bin/cobra add initialization
initialization created at
/Users/mtsouk/go/src/aliases/cmd/initialization.go
$ ~/go/bin/cobra add preferences
preferences created at /Users/mtsouk/go/src/aliases/cmd/preferences.go
```

Итак, теперь у нас есть утилита командной строки, которая поддерживает две команды: `initialization` и `preferences`.

Все псевдонимы существующих команд `cobra` должны быть четко указаны в коде Go. Для команды `initialization` нам понадобится следующая строка кода из файла `./cmd/initialization.go`:

```
Aliases: []string{"initialize", "init"},
```

Этот оператор создает два псевдонима для команды `initialization`: `initialize` и `init`. Содержимое окончательной версии `./cmd/initialization.go` без учета комментариев будет следующим:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var initializationCmd = &cobra.Command{
    Use:     "initialization",
    Aliases: []string{"initialize", "init"},
    Short:   "A brief description of your command",
    Long:    `A longer description of your command`,
    Run:     func(cmd *cobra.Command, args []string) {
        fmt.Println("initialization called")
    },
}

func init() {
    rootCmd.AddCommand(initializationCmd)
}
```

Аналогично для команды `preferences` нужно добавить в файл `./cmd/preferences.go` следующую строку кода Go:

```
Aliases: []string{"prefer", "pref", "prf"},
```

Это утверждение создает три псевдонима для команды `preferences`: `prefer`, `pref` и `prf`.

В итоге окончательная версия `./cmd/preferences.go` без учета комментариев выглядит так:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var preferencesCmd = &cobra.Command{
    Use:      "preferences",
    Aliases:  []string{"prefer", "pref", "prf"},
    Short:    "A brief description of your command",
    Long:     `A longer description of your command`,
    Run:      func(cmd *cobra.Command, args []string) {
        fmt.Println("preferences called")
    },
}

func init() {
    rootCmd.AddCommand(preferencesCmd)
}
```

Выполнение утилиты командной строки `aliases` приведет к результатам такого вида:

```
$ go run main.go prefer
preferences called
$ go run main.go prf
preferences called
$ go run main.go init
initialization called
```

Если вы дадите псевдониму ошибочную команду, то он выдаст сообщение об ошибке следующего типа:

```
$ go run main.go inits
Error: unknown command "inits" for "aliases"
Run 'aliases --help' for usage.
unknown command "inits" for "aliases"
exit status 1
$ go run main.go prefer
Error: unknown command "prefer" for "aliases"
Did you mean this?
    preferences
Run 'aliases --help' for usage.
unknown command "prefer" for "aliases"
Did you mean this?
    preferences
exit status 1
```

Утилита `tree(1)` не входит в стандартную комплектацию большинства систем UNIX, поэтому ее приходится устанавливать отдельно. Она поможет нам получить представление о структуре каталогов и файлов созданной утилиты `cobra`:

```
$ tree
```

```

.
├── LICENSE
├── cmd
│   ├── initialization.go
│   ├── preferences.go
│   └── root.go
└── main.go
1 directory, 5 files
```



В этой главе представлены далеко не все возможности `viper` и `cobra`.

Интерфейсы `io.Reader` и `io.Writer`

Как отмечено в предыдущей главе, для соответствия интерфейсу `io.Reader` тип данных требует реализации метода `Read()`, а для соответствия интерфейсу `io.Writer` — метода `Write()`. Оба эти интерфейса очень популярны в Go, и мы скоро их применим.

Буферизованный и небуферизованный ввод и вывод в файл

Буферизованный ввод и вывод в файл происходит в том случае, когда существует буфер для временного хранения данных перед чтением данных или записью. Таким образом, вместо того чтобы читать файл побайтно, мы одновременно считываем несколько байтов. Мы помещаем эти данные в буфер и ждем, пока кто-нибудь их прочтет желаемым способом. Небуферизованный ввод и вывод в файл происходит тогда, когда у нас нет буфера для временного хранения данных перед их фактическим чтением или записью.

Возможно, вы спросите: как решить, когда использовать буферизованный, а когда небуферизованный ввод и вывод в файл? При работе с критически важными данными, как правило, лучше выбрать небуферизованный ввод и вывод, поскольку буферизованное чтение данных может привести к устареванию данных, а буферизованная запись — к потере данных в случае отключения питания компьютера. Однако в большинстве случаев на данный вопрос нет однозначного ответа. Это означает, что мы можем использовать любой вариант, если он облегчает выполнение нашей задачи.

Пакет bufio

Как следует из названия, пакет `bufio` предназначен для буферизованного ввода и вывода данных (buffered input and output). Однако внутри пакета `bufio` все равно используются объекты типа `io.Reader` и `io.Writer`, для которых в пакете создается обертка, так что получаются объекты типа `bufio.Reader` и `bufio.Writer` соответственно. Как станет ясно из следующих подразделов, пакет `bufio` очень часто применяется для чтения текстовых файлов.

Чтение текстовых файлов

Текстовый файл — самый распространенный тип файлов, встречающихся в системах UNIX. В этом разделе вы познакомитесь с тремя способами чтения текстовых файлов: построчно, по словам и посимвольно. Вы увидите, что самым простым способом доступа к текстовому файлу является построчное чтение, в то время как чтение текстового файла по словам является самым сложным из всех методов.

Если вы внимательно посмотрите на программы `byLine.go`, `byWord.go` и `byCharacter.go`, то увидите в их коде Go много общего. Во-первых, все три утилиты читают входной файл построчно. Во-вторых, эти утилиты имеют одинаковую функцию `main()`, различающуюся только функцией, которая вызывается в цикле `for`. Наконец, три функции, которые обрабатывают входные текстовые файлы, практически идентичны, за исключением части, которая реализует фактическую функциональность.

Построчное чтение текстового файла

Проходить файл строка за строкой — самый распространенный метод чтения текстового файла. Именно поэтому мы рассмотрим его в первую очередь. Понять этот способ вам поможет код Go программы `byLine.go`, который мы разделим на три части.

Первый фрагмент `byLine.go` содержит следующий код Go:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

Как видим, здесь включен пакет `bufio`, следовательно, мы будем использовать буферизованный ввод.

Вторая часть `byLine.go` содержит следующий код Go:

```
func lineByLine(file string) error {
    var err error
    f, err := os.Open(file)

    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            break
        }
        fmt.Print(line)
    }
    return nil
}
```

Главная магия происходит в функции `lineByLine()`. Убедившись, что файл с таким именем открывается для чтения, мы создаем нового читателя, используя функцию `bufio.NewReader()`. Затем мы используем этого читателя для построчного чтения входного файла с помощью функции `bufio.ReadString()`. Этот прием выполняется с помощью аргумента функции `bufio.ReadString()`. Данный аргумент представляет собой символ, который дает `bufio.ReadString()` команду продолжать чтение, пока не будет найден заданный символ. Если постоянно вызывать `bufio.ReadString()`, задавая в качестве этого аргумента символ новой строки, то получим построчное чтение входного файла. Используя функцию `fmt.Print()` вместо `fmt.Println()` для вывода на экран считанной строки, мы можем увидеть, что в каждую считанную строку включен символ новой строки.

Третья часть `byLine.go` выглядит так:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byLine <file1> [<file2> ...]\n")
        return
    }
}
```



```

    for _, file := range flag.Args() {
        err := lineByLine(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}

```

Выполнение программы `byLine.go` и обработка ее результатов с помощью `wc(1)` приведет к результатам следующего вида:

```

$ go run byLine.go /tmp/swtag.log /tmp/adobegc.log | wc
4761      88521      568402

```

Следующая команда позволяет убедиться в точности предыдущих результатов:

```

$ wc /tmp/swtag.log /tmp/adobegc.log
  131      693      8440 /tmp/swtag.log
4630     87828    559962 /tmp/adobegc.log
4761     88521    568402 total

```

Чтение текстового файла по словам

Представленный в этом подразделе способ продемонстрирован на примере программы `byWord.go`, которую мы разделим на четыре части. Как вы увидите в коде Go, разделение строки на слова — иногда сложная задача.

Первая часть этой утилиты выглядит так:

```

package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
    "regexp"
)

```

Вторая часть `byWord.go` содержит следующий код Go:

```

func wordByWord(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)

```

```

for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        return err
    }
}

```

Эта часть функции `wordByWord()` аналогична функции `lineByLine()` утилиты `byLine.go`.

Третья часть `byWord.go`:

```

    r := regexp.MustCompile("^[\\s]+")
    words := r.FindAllString(line, -1)
    for i := 0; i < len(words); i++ {
        fmt.Println(words[i])
    }
}
return nil
}

```

Последняя часть кода функции `wordByWord()` выглядит совершенно незнакомо. Здесь с помощью регулярных выражений каждая строка входного файла разделяется на отдельные слова. Регулярное выражение, определенное в операторе `regexp.MustCompile("^[\\s]+")`, предполагает, что слова отделяются одно от другого посредством пробельных символов.

Последний фрагмент кода `byWord.go` выглядит так:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byWord <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := wordByWord(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}

```

Выполнение `byWord.go` приведет к результатам следующего вида:

```

$ go run byWord.go /tmp/adobegc.log
01/08/18
20:25:09:669
|
[INFO]

```

Чтобы проверить правильность работы `byWord.go`, можно воспользоваться утилитой `wc(1)`:

```
$ go run byWord.go /tmp/adobegc.log | wc
91591    91591    559005
$ wc /tmp/adobegc.log
4831     91591    583454 /tmp/adobegc.log
```

Как видим, количество слов, вычисленных с помощью `wc(1)`, равно количеству строк и слов, которые мы получили, выполнив `byWord.go`.

Посимвольное чтение текстового файла

В этом разделе показано, как прочитать текстовый файл посимвольно, что на практике приходится делать довольно редко, если только вы не разрабатываете текстовый редактор. Соответствующий код Go находится в файле `byCharacter.go`. Разделим его на четыре части.

Первая часть `byCharacter.go` содержит следующий код Go:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

Как видим, для этой задачи нам не придется использовать регулярные выражения.

Второй фрагмент кода `byCharacter.go` выглядит так:

```
func charByChar(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
    }
}
```

В третьей части `byCharacter.go` заключается логика программы:

```
    for _, x := range line {
        fmt.Println(string(x))
    }
}
return nil
}
```

Здесь мы перебираем все прочитанные строки и разделяем каждую строку, используя `range`, который возвращает два значения. Первое из них, которое является местоположением текущего символа в строковой переменной, мы отбрасываем и используем второе. Однако это значение не является символом, поэтому нам необходимо преобразовать его в символ с помощью функции `string()`.

Обратите внимание, что оператор `fmt.Println(string(x))` выводит каждый символ в отдельной строке и поэтому вывод программы будет длинным. Для того чтобы сделать его более компактным, нужно использовать функцию `fmt.Print()`.

Последняя часть `byCharacter.go` содержит следующий код Go:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byChar <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := charByChar(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Выполнение `byCharacter.go` приведет к результатам следующего вида:

```
$ go run byCharacter.go /tmp/adobegc.log
0
1
/
0
8
/
1
8
```

Обратите внимание, что представленный здесь код Go можно использовать для подсчета количества символов, найденных во входном файле, что можно использовать для реализации Go-версии удобной утилиты командной строки `wc(1)`.

Чтение из /dev/random

В этом подразделе показано, как читать данные с устройства /dev/random. Назначение системного устройства /dev/random — генерировать случайные данные, которые можно использовать для тестирования программ или, как в данной ситуации, в качестве начального числа для генератора случайных чисел. Получить данные из /dev/random иногда немного сложнее, и именно поэтому данная тема выделена в отдельный раздел.

На компьютере с macOS Mojave файл /dev/random имеет следующие полномочия доступа:

```
$ ls -l /dev/random
crw-rw-rw- 1 root wheel 14, 0 Mar 12 20:24 /dev/random
```

Аналогично на компьютере с Debian Linux для системного устройства /dev/random установлены следующие полномочия доступа:

```
$ ls -l /dev/random
crw-rw-rw- 1 root root 1, 8 Oct 13 12:19 /dev/random
```

Это означает, что в обоих вариантах UNIX файл /dev/random имеет примерно одинаковые полномочия доступа. Единственное различие между этими вариантами UNIX — это группа UNIX, к которой принадлежит файл: wheel в macOS и root в Debian Linux.

Программа, которая рассмотрена далее, называется devRandom.go. Разделим ее на три части. Первая часть программы выглядит так:

```
package main

import (
    "encoding/binary"
    "fmt"
    "os"
)
```

Для чтения данных из устройства /dev/random нужно импортировать стандартный Go-пакет encoding/binary, поскольку /dev/random предоставляет двоичные данные, которые необходимо декодировать. Вторая часть кода devRandom.go будет такой:

```
func main() {
    f, err := os.Open("/dev/random")
    defer f.Close()

    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Мы открываем `/dev/random` как обычный файл, поскольку все в UNIX — это файлы.

Последний фрагмент `devRandom.go` содержит следующий код Go:

```
var seed int64
binary.Read(f, binary.LittleEndian, &seed)
fmt.Println("Seed:", seed)
}
```

Для чтения данных с системного устройства `/dev/random` нам нужна функция `binary.Read()`, которая принимает три аргумента. Значение второго из них (`binary.LittleEndian`) указывает, что мы хотим использовать *прямой порядок* байтов. Второй из возможных вариантов, `binary.BigEndian`, выбирается в тех случаях, когда на компьютере используется *обратный порядок* байтов.

Выполнение `devRandom.go` приведет к результатам следующего вида:

```
$ go run devRandom.go
Seed: -2044736418491485077
$ go run devRandom.go
Seed: -5174854372517490328
$ go run devRandom.go
Seed: 7702177874251412774
```

Чтение заданного количества данных

В этом разделе показано, как прочесть именно столько данных, сколько нам нужно. Такой метод особенно полезен при чтении двоичных файлов, когда требуется декодировать считанные данные определенным образом. Однако это работает и для текстовых файлов.

Логика метода заключается в следующем: мы создаем *байтовый срез* нужного размера и используем его для чтения данных. Чтобы было еще интереснее, мы реализуем этот функционал как функцию с двумя параметрами: один параметр будет указывать количество данных, которые мы хотим прочесть, а второй (с типом `*os.File`) будет использоваться для доступа к нужному файлу. Функция будет возвращать прочитанные данные.

Программа Go, которую мы рассмотрим, называется `readSize.go`. Разделим ее на четыре части. Эта утилита принимает единственный аргумент — размер байтового среза.



Программа, реализующая метод, описанный в этом разделе, поможет вам скопировать любой файл, используя соответствующий размер буфера.

Первая часть `readSize.go` содержит ожидаемую преамбулу:

```
package main

import (
    "fmt"
    "io"
    "os"
    "strconv"
)
```

Вторая часть `readSize.go` содержит следующий код Go:

```
func readSize(f *os.File, size int) []byte {
    buffer := make([]byte, size)

    n, err := f.Read(buffer)
    if err == io.EOF {
        return nil
    }

    if err != nil {
        fmt.Println(err)
        return nil
    }

    return buffer[0:n]
}
```

Эта функция нам уже встречалась. Ее код прост, однако есть один момент, который необходимо пояснить. Метод `io.Reader.Read()` возвращает два значения: количество прочитанных байтов и переменную `error`. Функция `readSize()`, как и раньше, возвращает то же значение, которое возвращает `io.Read()`, чтобы вернуть байтовый срез заданного размера. Эта информация важна только тогда, когда достигнут конец файла. Однако она гарантирует, что данные, выводимые утилитой, будут такими же, как и входные данные, и в них не будет содержаться никаких дополнительных символов.

Наконец, остается код, который проверяет наличие `io.EOF`, что является ошибкой и означает, что мы достигли конца файла. Когда возникает такая ошибка, функция завершает работу. Дейв Чейни (Dave Cheney), участник проекта Go и разработчик программного обеспечения с открытым кодом, называет такие ошибки «ошибками дозорного», поскольку они означают, что на самом деле ошибки не произошло.

Третья часть кода этой утилиты выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("<buffer size> <filename>")
    }
}
```

```

    return
}

bufferSize, err := strconv.Atoi(os.Args[1])
if err != nil {
    fmt.Println(err)
    return
}

file := os.Args[2]
f, err := os.Open(file)
if err != nil {
    fmt.Println(err)
    return
}
defer f.Close()

```

Последний фрагмент кода `readSize.go`:

```

for {
    readData := readSize(f, bufferSize)
    if readData != nil {
        fmt.Print(string(readData))
    } else {
        break
    }
}
}

```

Здесь мы продолжаем чтение входного файла, пока функция `readSize()` не вернет ошибку или `nil`.

Выполнение `readSize.go` с аргументом, указывающим на обработку двоичного файла, и последующая обработка результата с помощью `wc(1)` позволяют проверить правильность работы программы:

```

$ go run readSize.go 1000 /bin/ls | wc
   80   1032   38688
$ wc /bin/ls
   80   1032   38688 /bin/ls

```

Преимущества двоичных форматов

В предыдущем разделе на примере утилиты `readSize.go` показано, как читать файл побайтно — такой способ обычно применяется к двоичным файлам. Возможно, вы спросите: зачем читать данные в двоичном формате, когда текстовые форматы намного понятнее? Основная причина — экономия места. Представьте, что вы хотите сохранить в файле число 20 в виде строки. Нетрудно понять, что для хранения числа 20 в формате символов ASCII потребуется два байта: один для хранения символа 2, а другой — для хранения символа 0.

Для хранения числа 20 в двоичном формате достаточно всего одного байта, поскольку 20 можно представить как 00010100 в двоичном формате или как 0x14 в шестнадцатеричном формате.

Это различие может показаться незначительным, пока речь идет о небольших объемах данных, но оно существенно, когда речь заходит о данных, используемых в таких приложениях, как серверы баз данных.

Чтение CSV-файлов

CSV-файлы — это простые форматированные текстовые файлы. В этом разделе показано, как прочитать текстовый файл, в котором находятся координаты точек на плоскости, — другими словами, в каждой строке содержится пара координат. Кроме того, мы также воспользуемся внешней библиотекой Go, которая называется *Glut*, — она поможет нам построить график по точкам, координаты которых будут прочитаны из CSV-файла. Обратите внимание, что в *Glut* используется *Gnuplot*, поэтому, чтобы использовать *Glut*, придется установить библиотеку *Gnuplot* на ваш компьютер UNIX.

Файл с программой, которую мы рассмотрим в этом разделе, называется `CSVplot.go`. Разделим его на пять частей. Первая из них выглядит так:

```
package main

import (
    "encoding/csv"
    "fmt"
    "github.com/Arafatk/glut"
    "os"
    "strconv"
)
```

Вторая часть `CSVplot.go` содержит следующий код Go:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a data file!")
        return
    }

    file := os.Args[1]
    _, err := os.Stat(file)
    if err != nil {
        fmt.Println("Cannot stat", file)
        return
    }
}
```

В этом коде показано, как с помощью мощной функции `os.Stat()` проверить, существует ли данный файл.

Третья часть CSVplot.go выглядит так:

```
f, err := os.Open(file)
if err != nil {
    fmt.Println("Cannot open", file)
    fmt.Println(err)
    return
}

defer f.Close()
reader := csv.NewReader(f)
reader.FieldsPerRecord = -1
allRecords, err := reader.ReadAll()
if err != nil {
    fmt.Println(err)
    return
}
```

Четвертый фрагмент CSVplot.go содержит следующий код Go:

```
xP := []float64{}
yP := []float64{}
for _, rec := range allRecords {
    x, _ := strconv.ParseFloat(rec[0], 64)
    y, _ := strconv.ParseFloat(rec[1], 64)
    xP = append(xP, x)
    yP = append(yP, y)
}

points := [][]float64{}
points = append(points, xP)
points = append(points, yP)
fmt.Println(points)
```

Здесь мы преобразуем прочитанные строковые значения в числа, которые затем помещаем в двумерный срез с именем points.

Последняя часть CSVplot.go содержит следующий код Go:

```
dimensions := 2
persist := true
debug := false
plot, _ := glot.NewPlot(dimensions, persist, debug)

plot.SetTitle("Using Glot with CSV data")
plot.SetXLabel("X-Axis")
plot.SetYLabel("Y-Axis")
style := "circle"
plot.AddPointGroup("Circle:", style, points)
plot.SavePlot("output.png")
}
```

В этом коде Go показано, как создать PNG-файл с помощью библиотеки Glot и ее функции glot.SavePlot().

Нетрудно догадаться, что, прежде чем вы сможете скомпилировать и выполнить код `CSVplot.go`, вам нужно загрузить библиотеку `Glut`, для чего необходимо выполнить из оболочки UNIX следующую команду:

```
$ go get github.com/Arafatk/glut
```

Исходный CSV-файл, где содержатся координаты точек, по которым будет построен график, имеет следующий вид:

```
$ cat /tmp/dataFile
1,2
2,3
3,3
4,4
5,8
6,5
-1,12
-2,10
-3,10
-4,10
```

Выполнение `CSVplot.go` приведет к следующим результатам:

```
$ go run CSVplot.go /tmp/doesNotExist
Cannot stat /tmp/doesNotExist
$ go run CSVplot.go /tmp/dataFile
[[1 2 3 4 5 6 -1 -2 -3 -4] [2 3 3 4 8 5 12 10 10 10]]
```

Результаты работы `CSVplot.go` лучше видны на рис. 8.1.

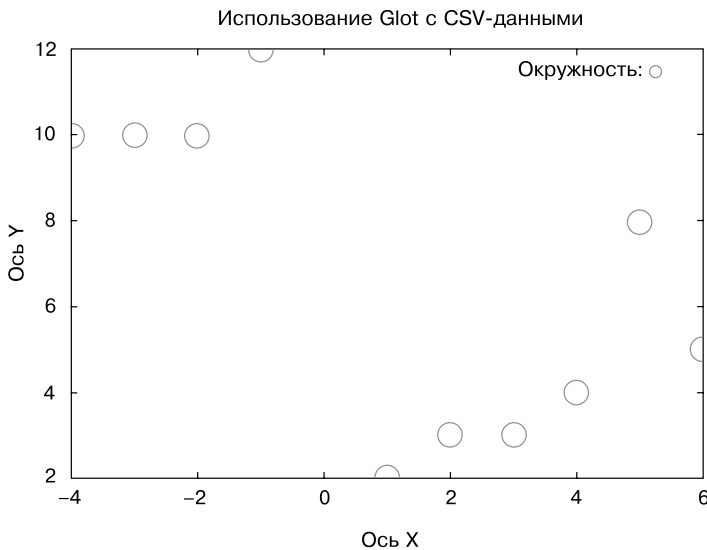


Рис. 8.1. График, который можно получить с помощью `Glut`

Запись в файл

Для записи данных в файлы на диске можно использовать функциональность интерфейса `io.Writer`. Тем не менее, на примере кода Go из программы `save.go` рассмотрим пять разных способов записи данных в файл. Для этого мы разделим программу `save.go` на шесть частей.

Первая часть `save.go` выглядит так:

```
package main
```

```
import (
    "bufio"
    "fmt"
    "io"
    "io/ioutil"
    "os"
)
```

Вторая часть `save.go` содержит следующий код Go:

```
func main() {
    s := []byte("Data to write\n")

    f1, err := os.Create("f1.txt")
    if err != nil {
        fmt.Println("Cannot create file", err)
        return
    }
    defer f1.Close()
    fmt.Fprintf(f1, string(s))
}
```

Обратите внимание, что мы будем использовать байтовый срез `s` в каждой строке этой программы Go, где производится запись. Кроме того, применяемая здесь функция `fmt.Fprintf()` позволяет записывать данные в ваши собственные файлы журнала, используя нужный формат. В нашем случае `fmt.Fprintf()` записывает данные в файл, обозначенный как `f1`.

В третьей части `save.go` содержится следующий код Go:

```
f2, err := os.Create("f2.txt")
if err != nil {
    fmt.Println("Cannot create file", err)
    return
}
defer f2.Close()
n, err := f2.WriteString(string(s))
fmt.Printf("wrote %d bytes\n", n)
```

В данном случае для записи данных в файл используется функция `f2.WriteString()`.

Четвертый фрагмент кода `save.go` выглядит так:

```
f3, err := os.Create("f3.txt")
if err != nil {
    fmt.Println(err)
    return
}
w := bufio.NewWriter(f3)
n, err = w.WriteString(string(s))
fmt.Printf("wrote %d bytes\n", n)
w.Flush()
```

В данном случае функция `bufio.NewWriter()` открывает файл для записи, а `bufio.WriteString()` записывает данные.

В пятой части `save.go` показан другой метод записи в файл:

```
f4 := "f4.txt"
err = ioutil.WriteFile(f4, s, 0644)
if err != nil {
    fmt.Println(err)
    return
}
```

Этот метод требует только одного вызова функции `ioutil.WriteFile()` для записи данных, и не требует использования `os.Create()`.

Последний фрагмент кода `save.go` выглядит так:

```
f5, err := os.Create("f5.txt")
if err != nil {
    fmt.Println(err)
    return
}
n, err = io.WriteString(f5, string(s))
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("wrote %d bytes\n", n)
}
```

В этом методе для записи данных в файл использована функция `io.WriteString()`.

Выполнение `save.go` приведет к следующим результатам:

```
$ go run save.go
wrote 14 bytes
wrote 14 bytes
wrote 14 bytes
$ ls -l f?.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f1.txt
```

```

-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f2.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f3.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f4.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f5.txt
$ cat f?.txt
Data to write
Data to write
Data to write
Data to write
Data to write

```

В следующем разделе показано, как сохранить данные в файле с помощью специализированной функции пакета, который входит в состав стандартной библиотеки Go.

Загрузка и сохранение данных на диске

Помните ли вы приложение `keyValue.go` из главы 4? Оно, правда, было далеко от совершенства. В этом разделе вы узнаете, как записать данные из *хранилища «ключ — значение»* на диск и как загрузить их обратно в память при следующем запуске приложения.

Мы создадим две новые функции: одну, с именем `save()`, для сохранения данных на диск, и вторую, с именем `load()`, для получения данных с диска. Таким образом, мы представим здесь только различия между кодом `keyValue.go` и `kvSaveLoad.go` с помощью UNIX-утилиты командной строки `diff(1)`.



UNIX-утилита командной строки `diff(1)` бывает очень удобной, если нужно найти различия между двумя текстовыми файлами. Чтобы узнать о ней больше, введите `man 1 diff` в командной строке оболочки UNIX.

Если вы зададитесь вопросом, как реализовать эту задачу, то поймете, что для ее решения вам, в сущности, нужен простой способ сохранить содержимое отображения Go на диске, а также способ загрузить данные из файла и поместить их в хеш-таблицу Go.

Процесс преобразования данных в поток байтов называется *сериализацией*. Процесс чтения файла данных и преобразования этих данных в объект называется *десериализацией*. Для сериализации и десериализации в программе `kvSaveLoad.go` используется стандартный Go-пакет `encoding/gob`. Пакет `encoding/gob` сохраняет данные в формате `gob`. Официально такие форматы называются *поточковым кодированием*. Преимущество формата `gob` заключается в том, что Go выполняет всю грязную работу, и вам не приходится беспокоиться об этапах кодирования и декодирования.

В Go для сериализации и десериализации данных также используются такие пакеты, как `encoding/xml`, который преобразует данные в *формат XML*, и `encoding/json`, который сохраняет данные в *формате JSON*.

Далее показаны различия в коде `kvSaveLoad.go` и `keyValue.go`, без учета реализаций функций `save()` и `load()`, которые мы внимательно рассмотрим позже:

```
$ diff keyValue.go kvSaveLoad.go
4a5
> "encoding/gob"
16a18,55
> var DATAFILE = "/tmp/dataFile.gob"
> func save() error {
>
>     return nil
> }
>
> func load() error {
>
> }
59a99,104
>
>     err := load()
>     if err != nil {
>         fmt.Println(err)
>     }
>
88a134,137
>     err = save()
>     if err != nil {
>         fmt.Println(err)
>     }
```

Важной частью результатов, полученных благодаря `diff(1)`, является определение глобальной переменной `DATAFILE`. Эта переменная содержит путь к файлу, который используется для хранилища данных «ключ — значение». Помимо этого, здесь видно, где вызывается функция `load()`, а также в какой точке вызывается `save()`. Сначала в функции `main()` используется `load()`, а затем, когда пользователь запускает команду `STOP`, выполняется `save()`.

При каждом запуске `kvSaveLoad.go` программа проверяет, есть ли данные для чтения, пытаясь прочитать файл данных, предлагаемый по умолчанию. Если файла данных для чтения нет, то программа начнет работу с пустого хранилища «ключ — значение». Перед завершением работы программа записывает все данные на диск с помощью функции `save()`.

Реализация `save()` выглядит так:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
```

```

err := os.Remove(DATAFILE)
if err != nil {
    fmt.Println(err)
}

saveTo, err := os.Create(DATAFILE)
if err != nil {
    fmt.Println("Cannot create", DATAFILE)
    return err
}
defer saveTo.Close()

encoder := gob.NewEncoder(saveTo)
err = encoder.Encode(DATA)
if err != nil {
    fmt.Println("Cannot save to", DATAFILE)
    return err
}
return nil
}

```

Обратите внимание: функция `save()` прежде всего удаляет существующий файл данных с помощью функции `os.Remove()`, чтобы позже создать новый.

Одна из самых важных задач, которые выполняет функция `save()`, — убедиться, что мы действительно можем создать и записать нужный файл. Есть много способов сделать это, и в `save()` используется самый простой из них: проверка значения `error`, возвращаемого функцией `os.Create()`. Если это значение не равно `nil`, это указывает на проблему и функция `save()` завершается без сохранения каких-либо данных.

Функция `load()` реализована следующим образом:

```

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

```

Одна из задач функции `load()` — убедиться, что файл, который мы пытаемся прочитать, действительно существует и его без проблем можно прочитать.

В функции `load()` также используется самый простой подход, который заключается в просмотре значения, возвращаемого функцией `os.Open()`. Если возвращаемое значение `error` равно `nil`, то все в порядке.

Важно также закрыть файл после чтения данных, поскольку впоследствии он будет перезаписан функцией `save()`. Освобождение файла выполняется с помощью инструкции `defer loadFrom.Close()`.

При выполнении программы `kvSaveLoad.go` будут получены результаты такого вида:

```
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
ADD 1 2 3
ADD 4 5 6
STOP
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 1 value: {2 3 }
key: 4 value: {5 6 }
DELETE 1
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
rMacBook:code mtsouk$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
$ ls -l /tmp/dataFile.gob
-rw-r--r-- 1 mtsouk wheel 80 Jan 22 11:22 /tmp/dataFile.gob
$ file /tmp/dataFile.gob
/tmp/dataFile.gob: data
```

В главе 13 представлена окончательная версия хранилища «ключ — значение», которое способно работать через соединение TCP/IP и обслуживать несколько сетевых клиентов посредством *горутин*.

И снова пакет `strings`

Удобный пакет `strings` вам впервые встретился в главе 4. В этом разделе мы рассмотрим функции пакета `strings`, связанные с вводом и выводом данных в файл.

Первая часть программы `str.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)
```

Второй фрагмент кода `str.go` выглядит так:

```
func main() {
    r := strings.NewReader("test")
    fmt.Println("r length:", r.Len())
}
```

Функция `strings.NewReader()` создает на основе строки объект `Reader`, предназначенный только для чтения. Объект `strings.Reader` реализует интерфейсы `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.WriterTo`, `io.ByteScanner` и `io.RuneScanner`.

Третья часть `str.go` выглядит таким образом:

```
b := make([]byte, 1)
for {
    n, err := r.Read(b)
    if err == io.EOF {
        break
    }

    if err != nil {
        fmt.Println(err)
        continue
    }
    fmt.Printf("Read %s Bytes: %d\n", b, n)
}
```

Здесь показано, как можно использовать `strings.Reader` в качестве `io.Reader` для побайтного чтения строки с помощью функции `Read()`.

Последняя часть `str.go` содержит следующий код Go:

```
s := strings.NewReader("This is an error!\n")
fmt.Println("r length:", s.Len())
n, err := s.WriteTo(os.Stderr)

if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Wrote %d bytes to os.Stderr\n", n)
}
```

В этом фрагменте кода показано, как записать сообщение в стандартный канал вывода ошибки с помощью пакета `strings`.

Выполнение `str.go` приведет к следующему результату:

```
$ go run str.go
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
This is an error!
Wrote 18 bytes to os.Stderr
$ go run str.go 2>/dev/null
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
Wrote 18 bytes to os.Stderr
```

Пакет bytes

Подобно тому как стандартный Go-пакет `strings` помогает работать со *строками*, стандартный Go-пакет `bytes` содержит функции для работы с *байтовыми срезами*. В этом разделе мы рассмотрим программу `bytes.go`, код которой разделим на три части.

Первая часть `bytes.go` выглядит так:

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "os"
)
```

Вторая часть `bytes.go` содержит следующий код Go:

```
func main() {
    var buffer bytes.Buffer
    buffer.Write([]byte("This is"))
    fmt.Fprintf(&buffer, " a string!\n")
    buffer.WriteTo(os.Stdout)
    buffer.WriteTo(os.Stdout)
}
```

Сначала мы создаем переменную `bytes.Buffer` и помещаем в нее данные с помощью функций `buffer.Write()` и `fmt.Fprintf()`, а затем дважды вызываем функцию `buffer.WriteTo()`.

При первом вызове `buffer.WriteTo()` будет выведено содержимое переменной `buffer`. Однако при втором вызове `buffer.WriteTo()` вывести нечего, поскольку после первого вызова `buffer.WriteTo()` переменная `buffer` станет пустой.

Последняя часть `bytes.go` выглядит так:

```
buffer.Reset()
buffer.Write([]byte("Mastering Go!"))
r := bytes.NewReader([]byte(buffer.String()))
fmt.Println(buffer.String())
for {
    b := make([]byte, 3)
    n, err := r.Read(b)
    if err == io.EOF {
        break
    }

    if err != nil {
        fmt.Println(err)
        continue
    }
    fmt.Printf("Read %s Bytes: %d\n", b, n)
}
}
```

Метод `Reset()` обнуляет переменную `buffer`, а метод `Write()` снова помещает в нее данные. Затем мы создаем новый объект `Reader` с помощью `bytes.NewReader()`, а после этого используем метод `Read()` интерфейса `io.Reader` для чтения данных из переменной `buffer`.

Выполнение `bytes.go` приведет к следующим результатам:

```
$ go run bytes.go
This is a string!
Mastering Go!
Read Mas Bytes: 3
Read ter Bytes: 3
Read ing Bytes: 3
Read Go Bytes: 3
Read ! Bytes: 1
```

Полномочия доступа к файлам

Популярная тема в программировании систем UNIX — полномочия доступа к файлам UNIX. В этом разделе показано, как вывести на экран полномочия для любого файла при условии, что вы обладаете необходимыми полномочиями в среде UNIX. Программа, которую мы рассмотрим, называется `permissions.go`. Разделим ее на три части.

Первая часть `permissions.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "os"
)
```

Второй фрагмент `permissions.go` содержит такой код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("usage: permissions filename\n")
        return
    }
}
```

Последняя часть этой утилиты выглядит так:

```
filename := arguments[1]
info, _ := os.Stat(filename)
mode := info.Mode()
fmt.Println(filename, "mode is", mode.String()[1:10])
}
```

Вызов `os.Stat(filename)` возвращает структуру с большим количеством данных. Поскольку нас интересуют только права доступа к файлу, мы вызываем метод `Mode()` и выводим на экран его результаты. В сущности, мы выводим только их часть, обозначенную как `mode.String()[1:10]`, поскольку именно здесь находятся интересующие нас данные.

Выполнение `permissions.go` приведет к результатам такого вида:

```
$ go run permissions.go /tmp/adobegc.log
/tmp/adobegc.log mode is rw-rw-rw-
$ go run permissions.go /dev/random
/dev/random mode is crw-rw-rw
```

Запустив утилиту `ls(1)`, мы можем проверить правильность работы `permissions.go`:

```
$ ls -l /dev/random /tmp/adobegc.log
crw-rw-rw- 1 root wheel 14, 0 Jan 8 20:24 /dev/random
-rw-rw-rw- 1 root wheel 583454 Jan 16 19:12 /tmp/adobegc.log
```

Обработка сигналов в UNIX

Чтобы помочь разработчикам в обработке сигналов, Go предоставляет им пакет `os/signal`. В этом разделе показано, как использовать этот пакет для обработки сигналов UNIX.

Прежде всего позвольте сообщить вам некоторую полезную информацию о сигналах UNIX. Вы когда-нибудь нажимали `Ctrl+C`, чтобы остановить работающую программу? Если да, то вы уже знакомы с сигналами UNIX, поскольку нажатие `Ctrl+C` отправляет в программу сигнал *SIGINT*. Строго говоря, *сигналы* UNIX — это программные прерывания, к которым можно обращаться по имени или по номеру и которые являются способом обработки асинхронных событий в системе UNIX. Вообще, безопаснее называть сигнал по имени, потому что тогда меньше вероятность случайно отправить неправильный сигнал.

Программа не способна обработать все существующие сигналы. Некоторые из них не перехватываются, однако и проигнорировать их также нельзя. Сигналы *SIGKILL* и *SIGSTOP* нельзя перехватить, заблокировать или проигнорировать, так как они предоставляют ядру и пользователю `root` возможность остановить любой процесс. Сигнал *SIGKILL*, который имеет номер 9, обычно вызывается в экстремальных ситуациях, когда нужно действовать быстро. Это единственный сигнал, который обычно вызывается по номеру — просто потому, что так быстрее.



Сигнал `signal.SIGINFO` на компьютерах с Linux недоступен. Поэтому, если вы обнаружите его в программе Go, которую хотите запустить на компьютере с Linux, вам придется заменить его на другой сигнал, иначе программа Go не будет скомпилирована и выполнена.

Самый распространенный способ передачи сигнала процессу — использование утилиты `kill(1)`. По умолчанию `kill(1)` передает сигнал *SIGTERM*. Если вы хотите узнать, какие сигналы поддерживаются на вашем компьютере с UNIX, используйте команду `kill -l`.

Если вы попытаетесь отправить сигнал процессу, не имея на то необходимых полномочий, `kill(1)` не выполнит эту операцию и вы получите такое сообщение об ошибке:

```
$ kill 1210
-bash: kill: (1210) - Operation not permitted
```

Обработка двух сигналов

В этом подразделе показано, как обрабатывать два сигнала в программе Go. Для этого мы рассмотрим код, который находится в файле `handleTwo.go`. Разделим его на четыре части. Программа `handleTwo.go` обрабатывает сигналы *SIGINFO* и *SIGINT*, которые в Go называются `syscall.SIGINFO` и `os.Interrupt` соответственно.



Если вы посмотрите в документацию пакета `os`, то увидите, что есть только два сигнала, которые гарантированно присутствуют во всех системах. Это `syscall.SIGKILL` и `syscall.SIGINT`, которые в Go также определены как `os.Kill` и `os.Interrupt` соответственно.

Первая часть `handleTwo.go` содержит следующий код Go:

```
package main
```

```
import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)
```

Вторая часть программы `handleTwo.go` выглядит так:

```
func handleSignal(signal os.Signal) {
    fmt.Println("handleSignal() Caught:", signal)
}
```

Функция `handleSignal()` используется для обработки сигнала `syscall.SIGINFO`, а сигнал `os.Interrupt` будет обрабатываться непосредственно.

Третий фрагмент `handleTwo.go` содержит следующий код Go:

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, os.Interrupt, syscall.SIGINFO)
    go func() {
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                fmt.Println("Caught:", sig)
            case syscall.SIGINFO:
                handleSignal(sig)
                return
            }
        }
    }()
}
```

Этот метод работает так: сначала мы определяем канал с именем `sigs`, по которому будут передаваться данные. Затем мы вызываем функцию `signal.Notify()`, чтобы указать на интересующие нас сигналы. После этого мы реализуем *анонимную функцию*, которая выполняется как горутина и включается тогда, когда поступает любой из интересующих нас сигналов. В главе 9 вы узнаете больше о горутинах и каналах.

Последняя часть программы `handleTwo.go` выглядит так:

```
for {
    fmt.Printf(".")
    time.Sleep(20 * time.Second)
}
}
```

Вызов `time.Sleep()` не дает программе завершиться, поскольку она не выполняет никакой полезной функции. В реальном приложении не было бы необходимости использовать подобный код.

Поскольку для передачи программе сигналов с помощью утилиты `kill(1)` нам нужен идентификатор процесса программы, сначала мы скомпилируем `handleTwo.go` и вместо выполнения команды `go runhandleTwo.go` запустим исполняемый файл. Результатом работы `handleTwo` будет примерно следующее:

```
$ go build handleTwo.go
$ ls -l handleTwo
-rwxr-xr-x 1 mtsouk staff 2005200 Jan 18 07:49 handleTwo
$ ./handleTwo
.^Ccaught: interrupt
.Caught: interrupt
handleSignal() Caught: information request
.Killed: 9
```

Обратите внимание, что для взаимодействия с `handleTwo.go` и получения показанных выше результатов вам потребуется дополнительное окно терминала, в котором нужно выполнить следующие команды:

```
$ ps ax | grep ./handleTwo | grep -v grep
47988 s003 S+ 0:00.00 ./handleTwo
$ kill -s INT 47988
$ kill -s INFO 47988
$ kill -s USR1 47988
$ kill -9 47988
```

Первая команда позволяет получить идентификатор процесса исполняемого файла `handleTwo`, а остальные команды используются для передачи этому процессу требуемых сигналов. Сигнал `SIGUSR1` игнорируется и не появляется на выходе.

У `handleTwo.go` есть проблема: если программа получает сигнал, который не запрограммирован для обработки, то она его игнорирует. Поэтому в следующем подразделе представлен другой подход для более эффективной обработки сигналов.

Обработка всех сигналов

В этом подразделе показано, как обрабатывать все сигналы, но реагировать только на те, которые действительно вас интересуют. Это гораздо лучший и более безопасный метод, чем тот, что был описан в предыдущем разделе. Этот метод будет продемонстрирован на примере Go-кода `handleAll.go`, который мы разделим на четыре части.

Первая часть `handleAll.go` содержит следующий код Go:

```
package main

import (
```



```

    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func handle(signal os.Signal) {
    fmt.Println("Received:", signal)
}

```

Второй фрагмент кода `handleAll.go` выглядит так:

```

func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs)
}

```

Таким образом, главная магия выполняется оператором `signal.Notify(sigs)`. Поскольку никакие конкретные сигналы не указаны, будут обработаны все входящие сигналы.



Мы можем вызывать `signal.Notify()` несколько раз в одной и той же программе для одних и тех же сигналов, используя разные каналы. В таком случае каждый канал получит копию сигналов, на обработку которых он был запрограммирован.

Третья часть кода утилиты `handleAll.go` выглядит следующим образом:

```

go func() {
    for {
        sig := <-sigs
        switch sig {
            case os.Interrupt:
                handle(sig)
            case syscall.SIGTERM:
                handle(sig)
                os.Exit(0)
            case syscall.SIGUSR2:
                fmt.Println("Handling syscall.SIGUSR2!")
            default:
                fmt.Println("Ignoring:", sig)
        }
    }
}()

```

Очень удобно использовать один из сигналов для выхода из программы. Это дает возможность при необходимости убирать в программе лишнее. В данном

случае для этой цели используется сигнал `syscall.SIGTERM`, что не мешает нам использовать `SIGKILL` для остановки программы.

Остальной код Go в `handleAll.go` выглядит так:

```
for {
    fmt.Printf(".")
    time.Sleep(30 * time.Second)
}
}
```

Нам по-прежнему нужно вызывать `time.Sleep()`, чтобы предотвратить немедленное завершение программы.

В этом случае, как и в прошлый раз, лучше создать исполняемый файл для `handleAll.go`, используя инструмент `go build`. Запуск программы `handleAll` и взаимодействие с ней из другого окна терминала приведет к следующим результатам:

```
$ go build handleAll.go
$ ls -l handleAll
-rwxr-xr-x 1 mtsouk staff 2005216 Jan 18 08:25 handleAll
$ ./handleAll
.Ignoring: hangup
Handling syscall.SIGUSR2!
Ignoring: user defined signal 1
Received: interrupt
^CReceived: interrupt
Received: terminated
```

Во втором окне терминала нужно ввести следующие команды:

```
$ ps ax | grep ./handleAll | grep -v grep
49902 s003 S+    0:00.00 ./handleAll
$ kill -s HUP 49902
$ kill -s USR2 49902
$ kill -s USR1 49902
$ kill -s INT 49902
$ kill -s TERM 49902
```

Программирование UNIX-каналов на Go

Согласно философии UNIX, каждая утилита командной строки UNIX должна выполнять только одну задачу, и должна делать это хорошо. На практике это означает, что вместо огромных утилит, выполняющих множество задач, следует разрабатывать небольшие программы, которые в совместном применении будут выполнять желаемую работу. Самый распространенный способ взаимодействия нескольких утилит командной строки UNIX — использование каналов. В *канале UNIX* результаты работы одной утилиты командной строки становятся входными

данными для другой утилиты. Этот процесс может включать в себя более двух программ. Для обозначения каналов UNIX используется символ |.

У каналов есть два серьезных ограничения: во-первых, они обычно работают только в одном направлении, а во-вторых, их можно использовать только для процессов, имеющих общего предка. Идея, лежащая в основе реализации каналов UNIX, заключается в том, что если у вас нет файла для обработки, то следует подождать поступления данных из стандартного потока ввода.

Аналогично если нет указания сохранять результаты в файле, то их следует вывести в стандартный поток вывода — либо для того, чтобы их увидел пользователь, либо для другой программы, которая будет их обрабатывать.

В главе 1 вы узнали, как читать данные из стандартного потока ввода и как записывать их в стандартный поток вывода. Если у вас остаются сомнения по поводу этих двух операций, самое время вернуться к Go-коду `stdOUT.go` и `stdIN.go`.

Реализация утилиты `cat(1)` на Go

В этом подразделе продемонстрирована Go-версия утилиты `cat(1)`. Скорее всего, вас удивит размер этой программы. Мы рассмотрим код `cat.go`, разделив его на три части. Первая часть `cat.go` выглядит так:

```
package main
```

```
import (
    "bufio"
    "fmt"
    "io"
    "os"
)
```

Второй фрагмент `cat.go` содержит следующий код Go:

```
func printFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        io.WriteString(os.Stdout, scanner.Text())
        io.WriteString(os.Stdout, "\n")
    }
    return nil
}
```

В этой части программы реализована функция, цель которой — вывод содержимого файла в стандартный поток вывода.

Последняя часть `cat.go` выглядит так:

```
func main() {
    filename := ""
    arguments := os.Args
    if len(arguments) == 1 {
        io.Copy(os.Stdout, os.Stdin)
        return
    }

    for i := 1; i < len(arguments); i++ {
        filename = arguments[i]
        err := printFile(filename)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

В данном коде содержится вся магия `cat.go`: именно здесь определяется, как будет вести себя программа. Прежде всего, если выполнить `cat.go` без аргументов командной строки, то программа просто скопирует стандартный поток ввода в стандартный поток вывода, что реализовано с помощью оператора `io.Copy(os.Stdout, os.Stdin)`. Но если передать программе аргументы командной строки, то программа обработает их в том порядке, в котором они были заданы.

Выполнение `cat.go` приведет к следующим результатам:

```
$ go run cat.go
Mastering Go!
Mastering Go!
1 2 3 4
1 2 3 4
```

Но если выполнить `cat.go`, задействовав каналы UNIX, все становится по-настоящему интересным:

```
$ go run cat.go /tmp/1.log /tmp/2.log | wc
 2367      44464      279292
$ go run cat.go /tmp/1.log /tmp/2.log | go run cat.go | wc
 2367      44464      279292
```

С помощью `cat.go` также можно вывести на экран содержимое нескольких файлов:

```
$ go run cat.go 1.txt 1 1.txt
 2367      44464      279292
 2367      44464      279292
open 1: no such file or directory
 2367      44464      279292
 2367      44464      279292
```

Обратите внимание, что если попытаться выполнить `cat.go` как `go run cat.go`, то вместо ожидаемого вывода на экран содержимого файла `cat.go` процесс завершится ошибкой и вы получите следующее сообщение:

```
package main: case-insensitive file name collision: "cat.go" and "cat.go"
```

Это происходит из-за того, что Go не понимает, что второй файл `cat.go` следует использовать как аргумент командной строки для команды `go run cat.go`. Вместо этого `go run` пытается дважды скомпилировать `cat.go`, что вызывает сообщение об ошибке. Решение этой проблемы состоит в том, чтобы сначала выполнить `go build cat.go`, а затем использовать `cat.go` или любой другой исходный файл Go в качестве аргумента для сгенерированного двоичного исполняемого файла.

Структура `syscall.PtraceRegs`

Если вы решили, что мы закончили изучение стандартного Go-пакета `syscall`, то вы ошибаетесь! В этом разделе мы поработаем со структурой `syscall.PtraceRegs`, в которой содержится информация о состоянии реестров.

Вы научитесь выводить на экран значения следующих реестров, используя Go-код из файла `ptraceRegs.go`, который мы разделим на четыре части. Звездой утилиты `ptraceRegs.go` является функция `syscall.PtraceGetRegs()`. Для работы с реестрами существуют также функции `syscall.PtraceSetRegs()`, `syscall.PtraceAttach()`, `syscall.PtracePeekData()` и `syscall.PtracePokeData()`, однако в `ptraceRegs.go` эти функции использоваться не будут.

Первая часть утилиты `ptraceRegs.go` выглядит так:

```
package main
```

```
import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
    "time"
)
```

Во второй части `ptraceRegs.go` содержится следующий код Go:

```
func main() {
    var r syscall.PtraceRegs
    cmd := exec.Command(os.Args[1], os.Args[2:]...)

    cmd.Stdout = os.Stdout
    cmd.Stderr = os.Stderr
```

Последние два оператора перенаправляют стандартный поток вывода и стандартный поток ошибок выполняемой команды в стандартный поток вывода UNIX и стандартный поток ошибок соответственно.

В третьей части `ptraceRegs.go` содержится следующий код Go:

```
cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
err := cmd.Start()
if err != nil {
    fmt.Println("Start:", err)
    return
}

err = cmd.Wait()
fmt.Printf("State: %v\n", err)
wpid := cmd.Process.Pid
```

В этом коде Go мы вызываем внешнюю команду, которая указана в аргументах командной строки программы, и находим идентификатор процесса, который будет использоваться в вызове `syscall.PtraceGetRegs()`. Оператор `&syscall.SysProcAttr{Ptrace: true}` указывает на то, что мы хотим использовать `ptrace` в дочернем процессе.

Последний фрагмент кода `ptraceRegs.go` выглядит так:

```
err = syscall.PtraceGetRegs(wpid, &r)
if err != nil {
    fmt.Println("PtraceGetRegs:", err)
    return
}
fmt.Printf("Registers: %#v\n", r)
fmt.Printf("R15=%d, Gs=%d\n", r.R15, r.Gs)

time.Sleep(2 * time.Second)
}
```

Здесь мы вызываем `syscall.PtraceGetRegs()` и выводим на экран результаты, хранящиеся в переменной `r`, которая должна быть передана как указатель.

Выполнение `ptraceRegs.go` на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run ptraceRegs.go
# command-line-arguments
./ptraceRegs.go:11:8: undefined: syscall.PtraceRegs
./ptraceRegs.go:14:9: undefined: syscall.PtraceGetRegs
```

Это значит, что данная программа не работает на компьютерах под управлением macOS и Mac OS X.

Выполнение `ptraceRegs.go` на компьютере с Debian Linux даст следующий результат:

```
$ go version
go version go1.7.4 linux/amd64
$ go run ptraceRegs.go echo "Mastering Go!"
State: stop signal: trace/breakpoint trap
Registers: syscall.PtraceRegs{R15:0x0, R14:0x0, R13:0x0, R12:0x0, Rbp:0x0,
Rbx:0x0, R11:0x0, R10:0x0, R9:0x0, R8:0x0, Rax:0x0, Rcx:0x0, Rdx:0x0,
Rsi:0x0, Rdi:0x0, Orig_rax:0x3b, Rip:0x7f4045f81c20, Cs:0x33, Eflags:0x200,
Rsp:0x7ffe1905b070, Ss:0x2b, Fs_base:0x0, Gs_base:0x0, Ds:0x0, Es:0x0,
Fs:0x0, Gs:0x0}
R15=0, Gs=0
Mastering Go!
```

Вы также найдете полный список реестров на странице документации пакета `syscall`.

Отслеживание системных вызовов

В этом разделе представлена усовершенствованная технология, использующая пакет `syscall` и позволяющая отслеживать системные вызовы, выполняемые в программе Go.

Утилита Go, которую мы рассмотрим, называется `traceSyscall.go`. Разделим ее на пять фрагментов кода. Первая часть `traceSyscall.go` выглядит так:

```
package main
```

```
import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
    "strings"
    "syscall"
)
```

```
var maxSyscalls = 0
```

```
const SYSCALLFILE = "SYSCALLS"
```

О назначении переменной `SYSCALLFILE` мы поговорим чуть позже.

Второй фрагмент кода `traceSyscall.go` выглядит так:

```
func main() {
    var SYSTEMCALLS []string
    f, err := os.Open(SYSCALLFILE)
    defer f.Close()
```

```

if err != nil {
    fmt.Println(err)
    return
}

scanner := bufio.NewScanner(f)
for scanner.Scan() {
    line := scanner.Text()
    line = strings.Replace(line, " ", "", -1)
    line = strings.Replace(line, "SYS_", "", -1)
    temp := strings.ToLower(strings.Split(line, "=")[0])
    SYSTEMCALLS = append(SYSTEMCALLS, temp)
    maxSyscalls++
}

```

Обратите внимание, что содержимое текстового файла `SYSTEMCALLS` взято из документации пакета `syscall`. В этом файле каждому системному вызову поставлен в соответствие номер, который является внутренним представлением Go для данного системного вызова. Этот файл нужен главным образом для вывода на экран имен системных вызовов, которые используются в отслеживаемой программе.

Содержимое текстового файла `SYSTEMCALLS` имеет следующий формат:

```

SYS_READ = 0
SYS_WRITE = 1
SYS_OPEN = 2
SYS_CLOSE = 3
SYS_STAT = 4

```

Прочитав текстовый файл, программа создает для хранения этой информации срез `SYSTEMCALLS`.

Третья часть `traceSyscall.go` выглядит так:

```

COUNTER := make([]int, maxSyscalls)
var regs syscall.PtraceRegs
cmd := exec.Command(os.Args[1], os.Args[2:]...)

cmd.Stdin = os.Stdin
cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}

err = cmd.Start()
err = cmd.Wait()
if err != nil {
    fmt.Println("Wait:", err)
}

pid := cmd.Process.Pid
fmt.Println("Process ID:", pid)

```


В срезе COUNTER хранятся данные о том, сколько раз каждый системный вызов был обнаружен в отслеживаемой программе.

Четвертый фрагмент `traceSyscall.go` содержит следующий код Go:

```
before := true
forCount := 0
for {
    if before {
        err := syscall.PtraceGetRegs(pid, &regs)
        if err != nil {
            break
        }
        if regs.Orig_rax > uint64(maxSyscalls) {
            fmt.Println("Unknown:", regs.Orig_rax)
            return
        }

        COUNTER[regs.Orig_rax]++
        forCount++
    }

    err = syscall.PtraceSyscall(pid, 0)
    if err != nil {
        fmt.Println("PtraceSyscall:", err)
        return
    }

    _, err = syscall.Wait4(pid, nil, 0, nil)
    if err != nil {
        fmt.Println("Wait4:", err)
        return
    }
    before = !before
}
```

Функция `syscall.PtraceSyscall()` заставляет Go продолжать выполнение отслеживаемой программы, но останавливаться, когда эта программа входит или выходит из системного вызова, — именно это нам и нужно! Каждый системный вызов отслеживается перед обработкой, сразу после того, как он завершил свою работу. Поэтому мы используем переменную `before`, чтобы учитывать каждый системный вызов только один раз.

Последняя часть `traceSyscall.go` выглядит так:

```
for i, x := range COUNTER {
    if x != 0 {
        fmt.Println(SYSTEMCALLS[i], "->", x)
    }
}
```

```

    }
    fmt.Println("Total System Calls:", forCount)
}

```

Здесь мы выводим на экран содержимое среза COUNTER. Срез SYSTEMCALLS в данном случае используется для определения имени системного вызова по известному числовому представлению этого вызова в Go.

Выполнение `traceSyscall.go` на компьютере с macOS Mojave приведет к следующим результатам:

```

$ go run traceSyscall.go
# command-line-arguments
./traceSyscall.go:36:11: undefined: syscall.PtraceRegs
./traceSyscall.go:57:11: undefined: syscall.PtraceGetRegs
./traceSyscall.go:70:9: undefined: syscall.PtraceSyscall

```

Как видим, утилита `traceSyscall.go` тоже не работает на macOS и Mac OS X.

Выполнение этой же программы на компьютере с Debian Linux приведет к следующему результату:

```

$ go run traceSyscall.go ls /tmp/
Wait: stop signal: trace/breakpoint trap
Process ID: 5657
go-build084836422 test.go upload_progress_cache
read -> 11
write -> 1
open -> 37
close -> 27
stat -> 1
fstat -> 25
mmap -> 39
mprotect -> 16
munmap -> 4
brk -> 3
rt_sigaction -> 2
rt_sigprocmask -> 1
ioctl -> 2
access -> 9
execve -> 1
getdents -> 2
getrlimit -> 1
statfs -> 2
arch_prctl -> 1
futex -> 1
set_tid_address -> 1
openat -> 1
set_robust_list -> 1
Total System Calls: 189

```

В завершение работы программа `traceSyscall.go` выводит количество появлений каждого системного вызова в программе. Правильность работы `traceSyscall.go` подтверждается результатами утилиты `strace -c`:

```
$ strace -c ls /tmp
test.go upload_progress_cache
```

% time	seconds	usecs/call	calls	errors	syscall
0.00	0.000000	0	11		read
0.00	0.000000	0	1		write
0.00	0.000000	0	37	13	open
0.00	0.000000	0	27		close
0.00	0.000000	0	1		stat
0.00	0.000000	0	25		fstat
0.00	0.000000	0	39		mmap
0.00	0.000000	0	16		mprotect
0.00	0.000000	0	4		munmap
0.00	0.000000	0	3		brk
0.00	0.000000	0	2		rt_sigaction
0.00	0.000000	0	1		rt_sigprocmask
0.00	0.000000	0	2		ioctl
0.00	0.000000	0	9		access
0.00	0.000000	0	1		execve
0.00	0.000000	0	2		getdents
0.00	0.000000	0	1		getrlimit
0.00	0.000000	0	2	2	statfs
0.00	0.000000	0	1		arch_prctl
0.00	0.000000	0	1		futex
0.00	0.000000	0	1		set_tid_address
0.00	0.000000	0	1		openat
0.00	0.000000	0	1		set_robust_list
100.00	0.000000		189	24	total

Идентификаторы пользователя и группы

В этом разделе вы узнаете, как найти идентификатор текущего пользователя, а также идентификаторы групп, к которым принадлежит этот пользователь. Идентификаторы пользователя и группы являются положительными целыми числами, которые хранятся в системных файлах UNIX.

Утилита, которую мы рассмотрим, называется `ids.go`. Разделим ее на две части. Первая часть этой утилиты выглядит так:

```
package main
```

```
import (
    "fmt"
```

```

    "os"
    "os/user"
)

func main() {
    fmt.Println("User id:", os.Getuid())

```

Для того чтобы узнать идентификатор текущего пользователя, нужно просто вызвать функцию `os.Getuid()`.

Вторая часть `ids.go` выглядит так:

```

var u *user.User
u, _ = user.Current()
fmt.Print("Group ids: ")
groupIDs, _ := u.GroupIds()
for _, i := range groupIDs {
    fmt.Print(i, " ")
}
fmt.Println()
}

```

В то же время поиск идентификаторов группы, к которой принадлежит данный пользователь, является более сложной задачей.

Выполнение `ids.go` приведет к результатам следующего вида:

```

$ go run ids.go
User id: 501
Group ids: 20 701 12 61 79 80 81 98 33 100 204 250 395 398 399

```

Docker API и Go

Если вы работаете с Docker, то этот раздел будет для вас особенно полезен, так как вы научитесь общаться с Docker с помощью Go и Docker API.

Мы рассмотрим утилиту `dockerAPI.go`, которую разделим на четыре части. Эта утилита реализует команды `docker ps` и `docker image ls`. Первая команда перечисляет все запущенные контейнеры, а вторая — все доступные образы на локальном компьютере.

Первая часть `dockerAPI.go` выглядит так:

```

package main

import (
    "fmt"
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
    "golang.org/x/net/context"
)

```

Поскольку `dockerAPI.go` требует много внешних пакетов, было бы неплохо выполнить эту программу с использованием Go-модулей; поэтому перед первым запуском `dockerAPI.go` выполните команду `export G0111MODULE = on`. Это также избавит вас от необходимости загружать все необходимые пакеты вручную.

Вторая часть `dockerAPI.go` содержит следующий код Go:

```
func listContainers() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    containers, err := cli.ContainerList(context.Background(),
        types.ContainerListOptions{})
    if err != nil {
        return (err)
    }

    for _, container := range containers {
        fmt.Println("Images:", container.Image, "with ID:", container.ID)
    }
    return nil
}
```

Определение структуры `types.Container` (<https://godoc.org/github.com/docker/docker/api/types#Container>), возвращаемой функцией `ContainerList()`, выглядит так:

```
type Container struct {
    ID          string `json:"Id"`
    Names       []string
    Image       string
    ImageID     string
    Command     string
    Created     int64
    Ports       []Port
    SizeRw      int64 `json:",omitempty"`
    SizeRootFs  int64 `json:",omitempty"`
    Labels      map[string]string
    State       string
    Status      string
    HostConfig  struct {
        NetworkMode string `json:",omitempty"`
    }
    NetworkSettings *SummaryNetworkSettings
    Mounts          []MountPoint
}
```

Чтобы найти любую другую информацию о списке запущенных образов (контейнеров) Docker, следует просто использовать другие поля структуры `types.Container`.

Третья часть `dockerAPI.go` выглядит так:

```
func listImages() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    images, err := cli.ImageList(context.Background(), types.ImageListOptions{})
    if err != nil {
        return (err)
    }

    for _, image := range images {
        fmt.Printf("Images %s with size %d\n", image.RepoTags, image.Size)
    }
    return nil
}
```

Определение структуры `types.ImageSummary` (<https://godoc.org/github.com/docker/docker/api/types#ImageSummary>), которая является срезом, возвращаемым `ImageList()`, имеет следующий вид:

```
type ImageSummary struct {
    Containers int64 `json:"Containers"`
    Created int64 `json:"Created"`
    ID string `json:"Id"`
    Labels map[string]string `json:"Labels"`
    ParentID string `json:"ParentId"`
    RepoDigests []string `json:"RepoDigests"`
    RepoTags []string `json:"RepoTags"`
    SharedSize int64 `json:"SharedSize"`
    Size int64 `json:"Size"`
    VirtualSize int64 `json:"VirtualSize"`
}
```

Последняя часть `dockerAPI.go` выглядит так:

```
func main() {
    fmt.Println("The available images are:")
    err := listImages()
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("The running Containers are:")
    err = listContainers()
    if err != nil {
        fmt.Println(err)
    }
}
```

Выполнение `dockerAPI.go` на моей машине с macOS Mojave привело к следующим результатам:

```
$ go run dockerAPI.go
The available images are:
Images [golang:1.12] with size 772436547
Images [landoop/kafka-lenses-dev:latest] with size 1379088343
Images [confluentinc/cp-kafka:latest] with size 568503378
Images [landoop/fast-data-dev:latest] with size 1052076831
The running Containers are:
Images: landoop/kafka-lenses-dev with ID:
90e1caaab43297810341290137186425878ef5891c787f6707c03be617862db5
```

Если Docker недоступен или не работает, то вы получите следующее сообщение об ошибке:

```
$ go run dockerAPI.go
The available images are:
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
docker daemon running?
The running Containers are:
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the
docker daemon running?
```

Дополнительные ресурсы

Следующие веб-ссылки будут вам очень полезны:

- ❑ почитайте страницу документации пакета `io`, которую вы найдете по адресу <https://golang.org/pkg/io/>;
- ❑ узнайте у Дейва Чейни, как обрабатывать ошибки: <https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-graceful>;
- ❑ чтобы больше узнать о библиотеке печати Glot, посетите официальную веб-страницу по адресу <https://github.com/Arafatk/glot>;
- ❑ другие примеры использования Docker API в Go, Python и HTTP вы найдете по адресу <https://docs.docker.com/develop/sdk/examples/>;
- ❑ вы узнаете больше о стандартном пакете `encoding/binary`, если посетите страницу <https://golang.org/pkg/encoding/binary/>;
- ❑ загляните на страницу документации пакета `encoding/gob`, которую вы найдете по адресу <https://golang.org/pkg/encoding/gob/>;
- ❑ советую также посмотреть видео <https://www.youtube.com/watch?v=JRFNIKUROPE> и <https://www.youtube.com/watch?v=w8nFRoFJ6EQ>;

- ❑ об *Endianness* пишут на многих ресурсах, в том числе здесь: <https://en.wikipedia.org/wiki/Endianness>;
- ❑ посетите страницу документации пакета `flag`, расположенную по адресу <https://golang.org/pkg/flag/>.

Упражнения

- ❑ Напишите программу Go, которая принимает три аргумента: имя текстового файла и две строки. Затем эта утилита должна заменить каждое вхождение первой строки в файле второй строкой. В целях безопасности окончательный результат выведите на экран, чтобы исходный текстовый файл остался без изменений.
- ❑ Используя пакет `encoding/gob`, напишите программу для сериализации и десериализации хеш-таблицы Go, а также среза структур.
- ❑ Создайте программу Go, которая бы обрабатывала три любых выбранных вами сигнала.
- ❑ Напишите на Go утилиту, которая бы заменяла все символы табуляции, найденные в текстовом файле, заданным количеством пробелов, указанным в качестве аргумента командной строки программы. Результат, как и в прошлый раз, выводите на экран.
- ❑ Разработайте утилиту, которая бы построчно читала текстовый файл и удаляла пробелы из каждой строки, используя функцию `strings.TrimSpace()`.
- ❑ Измените программу `kvSaveLoad.go` так, чтобы она поддерживала аргумент командной строки — имя файла, которое будет использоваться для загрузки и сохранения данных.
- ❑ Можете ли вы создать Go-версию утилиты `wc(1)`? Чтобы узнать, какие аргументы командной строки она поддерживает, загляните на справочную страницу `wc(1)`.
- ❑ Можете ли вы написать программу, которая бы использовала *Glut* для построения функции?
- ❑ Измените программу `traceSyscall.go` таким образом, чтобы она отображала каждый системный вызов во время его отслеживания.
- ❑ Измените программу `cat.go` таким образом, чтобы для копирования содержимого файла она просто выполняла `io.Copy(os.Stdout, f)`, вместо того чтобы сканировать его целиком.
- ❑ Используйте Docker API для написания утилиты, которая завершает все контейнеры, начинающиеся с заданной строки.

- ❑ Пакет `cobra` также поддерживает подкоманды — команды, связанные с определенными командами, такими как список команд `go run main.go`. Попробуйте реализовать утилиту с подкомандами.
- ❑ Для чтения строки по словам также можно использовать `bufio.NewScanner()` и `bufio.ScanWords`. Узнайте, как создать, и создайте новую версию утилиты `byWord.go`.

Резюме

В этой важной главе рассмотрено много интересных тем, в том числе чтение файлов, запись в файлы, использование Docker API, а также пакетов `flag`, `cobra` и `viper`. Тем не менее, остается много других тем, связанных с системным программированием, не упомянутых в этой главе, таких как работа с каталогами; копирование, удаление и переименование файлов; работа с пользователями, группами и процессами UNIX; изменение полномочий доступа к файлам UNIX; создание *неполных файлов*; блокировка и создание файлов; использование и замена ваших файлов журнала, а также информация из структуры, возвращаемой при вызове `os.Stat()`.

В конце главы я представил две расширенные утилиты, написанные на Go. Первая из них позволяет проверять состояние реестров, а во второй реализован метод, позволяющий отслеживать системные вызовы любой программы.

В следующей главе речь пойдет о программах, каналах и конвейерах — уникальных и мощных функциях Go.

9 Конкурентность в Go: горутины, каналы и конвейеры

В предыдущей главе рассмотрено системное программирование на Go, в том числе функции и методы Go, которые позволяют взаимодействовать с операционной системой. Но есть еще две области системного программирования, которые не были затронуты в предыдущей главе, — конкурентное программирование, а также способы создания и управления несколькими потоками. Эти темы разберем в настоящей и следующей главах.

В Go реализован собственный уникальный, инновационный способ обеспечения конкурентности в виде *горутин* и *каналов*. Горутины — это самые маленькие сущности Go, которые могут быть выполнены самостоятельно в программе Go. Каналы могут конкурентно и эффективно получать данные из горутин. Благодаря этому у горутин есть некая точка опоры, что позволяет им обмениваться данными между собой. В Go все выполняется посредством горутин. Это имеет смысл, поскольку Go — конкурентный язык программирования. Поэтому, когда начинается выполнение программы Go, ее единственная горутина вызывает функцию `main()`, которая и инициирует фактическое выполнение программы.

Содержание этой главы и представленные здесь фрагменты кода довольно просты, так что у вас не должно возникнуть проблем с их пониманием. Более сложные вопросы, касающиеся горутин и каналов, я оставил для главы 10.

В этой главе рассмотрены следующие темы:

- ❑ в чем различия между процессами, потоками и программами;
- ❑ планировщик Go;
- ❑ конкурентность и параллелизм;
- ❑ модели конкурентности в языках Erlang и Rust;
- ❑ создание горутин;
- ❑ создание каналов;
- ❑ чтение или получение данных из канала;

- ❑ запись или отправка данных в канал;
- ❑ создание конвейеров;
- ❑ и напоследок: ждем ваших горутин!

О процессах, потоках и горутинах

Процесс — это среда выполнения, в которой содержатся инструкции, пользовательские данные и части системных данных, а также другие типы ресурсов, полученные во время выполнения программы, тогда как сама *программа* — это файл, в котором содержатся инструкции и данные, используемые для инициализации инструкций и пользовательских данных процесса.

Поток — это меньшая и более легкая сущность, чем процесс или программа. Потоки создаются процессами, имеют собственный процесс управления и стек. Быстрый и простой способ отличить поток от процесса — рассматривать процесс как исполняемый двоичный файл, а поток — как подмножество процесса.

Горутина — это минимальная сущность Go, которая может быть выполнена конкурентно. Использование слова «минимальная» здесь очень важно, поскольку горутин не являются автономными сущностями, такими как процессы UNIX, — горутин живут в потоках UNIX, которые, в свою очередь, живут в процессах UNIX. Основным преимуществом горутин является то, что они чрезвычайно легкие, так что запуск тысяч или сотен тысяч горутин на одной машине не является проблемой.

Горутин легче потоков, которые, в свою очередь, легче процессов. На практике это означает, что в процессе может быть несколько потоков и множество горутин, в то время как горутина может существовать только в среде процесса. Таким образом, для создания горутин необходим процесс хотя бы с одним потоком — о процессах и управлении потоками позаботится UNIX, а Go и разработчик должны подумать о горутинах.

Теперь, когда вы познакомились с основными свойствами процессов, программ, потоков и горутин, поговорим немного о *планировщике Go*.

Планировщик Go

Планировщик ядра UNIX отвечает за выполнение потоков программы. Однако в среде выполнения Go есть собственный планировщик, который отвечает за выполнение горутин, используя технологию, называемую *планированием $m:n$* , при которой m горутин выполняется в n потоках операционной системы посредством мультиплексирования. Планировщик Go — это компонент Go, отвечающий за способ и последовательность выполнения горутин, из которых состоит программа Go.

Это делает планировщик Go очень важной частью языка программирования Go, поскольку все, что делает программа Go, выполняется в виде горутин.

Следует учитывать, что, поскольку планировщик Go оперирует горутинами только одной программы, он намного проще, дешевле и быстрее, чем планировщик ядра операционной системы.

Подробнее о том, как работает планировщик Go, вы узнаете из главы 10.

Конкурентность и параллелизм

Бытует очень распространенное заблуждение о том, что *конкурентность* и *параллелизм* — одно и то же. Это просто неправда! Параллелизм — это одновременное выполнение нескольких сущностей определенного вида, тогда как конкурентность — способ структурирования компонентов, позволяющий им выполняться независимо, когда это возможно.

Разработка конкурентных программных компонентов — необходимое условие того, что их можно будет безопасно выполнять параллельно, когда и если операционная система и оборудование это позволят. Это давным-давно было реализовано в языке программирования Erlang — задолго до появления многоядерных процессоров и компьютеров с большим объемом оперативной памяти.

В правильной конкурентной архитектуре добавление конкурентных сущностей приводит к тому, что вся система начинает работать быстрее, потому что в ней больше задач может выполняться параллельно. Таким образом, чтобы достичь желаемой степени параллелизма, необходимо обеспечить наилучшее конкурентное описание и реализацию задачи. Разработчик должен учитывать конкурентность на этапе проектирования системы — только тогда он получит выгоду от потенциально параллельного выполнения системных компонентов. Таким образом, разработчик должен думать не о параллелизме, а о том, как разбить систему на независимые компоненты, которые, будучи объединенными, решали бы исходную задачу.

Даже если на вашем компьютере с UNIX разработанные вами функции не могут выполняться параллельно, правильное конкурентное построение программы все равно улучшит ее структуру и удобство сопровождения. Другими словами, конкурентность лучше параллелизма!

Горутины

Чтобы определить, создать и выполнить новую горутину, нужно воспользоваться ключевым словом `go`, после которого поставить имя функции или полное определение *анонимной функции*. Ключевое слово `go` приводит к немедленному возврату (в основной поток) после вызова функции, в то время как сама функция выполняется в фоновом режиме как горутина, а остальная программа тем временем продолжает выполнение.

Однако, как вы скоро узнаете, мы не можем контролировать или делать какие-либо предположения о последовательности выполнения горутин, поскольку это зависит от планировщика операционной системы, планировщика Go и степени загрузки операционной системы.

Создание горутины

В этом подразделе показаны два способа создания горутин. Первый из них заключается в использовании обычных функций, а второй — в использовании анонимных функций. Эти способы эквивалентны.

В данном разделе мы рассмотрим программу `simple.go`. Разделим ее на три части.

Первая часть `simple.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "time"
)

func function() {
    for i := 0; i < 10; i++ {
        fmt.Print(i)
    }
}
```

Кроме блока `import`, в этом коде содержится определение функции с именем `function()`, которой мы вскоре воспользуемся.

Имя функции `function()` не является чем-то особенным — вы можете дать ей любое допустимое имя.

Вторая часть `simple.go` выглядит так:

```
func main() {
    go function()
```

Этот код начинается с выполнения `function()` в качестве горутины. После этого программа продолжает выполнение, в то время как `function()` начинает работать в фоновом режиме.

Последняя часть кода `simple.go` содержит следующий код Go:

```
    go func() {
        for i := 10; i < 20; i++ {
            fmt.Print(i, " ")
        }
    }()
```

```

    time.Sleep(1 * time.Second)
    fmt.Println()
}

```

С помощью этого кода мы создаем горутину, используя анонимную функцию. Этот метод лучше подходит для сравнительно небольших функций. Если же кода много, то рекомендуется создать обычную функцию и выполнить ее с помощью ключевого слова `go`.

Как вы увидите в следующем разделе, мы можем создавать сколько угодно горутин, в том числе используя цикл `for`.

Выполнение `simple.go` три раза подряд приведет к результатам такого вида:

```

$ go run simple.go
10 11 12 13 14 15 16 17 18 19 0123456789
$ go run simple.go
10 11 12 13 14 15 16 0117 2345678918 19
$ go run simple.go
10 11 12 012345678913 14 15 16 17 18 19

```

Обычно важно, чтобы программа генерировала одинаковые выходные данные для одних и тех же входных данных. Однако результаты, которые мы получаем от `simple.go`, не всегда одинаковы. Показанный выше вывод программы подтверждает тот факт, что мы не можем контролировать последовательность выполнения горутин, если специально не проследим за этим, то есть не напишем специальный дополнительный код. О том, как управлять последовательностью выполнения горутин, а также выводить результаты одной подпрограммы перед результатами следующей, вы узнаете из главы 10.

Создание нескольких горутин

В этом подразделе продемонстрировано, как создать переменное число горутин. Программа, которую мы рассмотрим, позволяет разрабатывать динамическое число горутин. Она называется `create.go`. Разделим ее на четыре части. Количество горутин задается как аргумент командной строки программы, а для обработки аргумента командной строки используется пакет `flag`.

Первая часть кода `create.go` выглядит так:

```

package main

import (
    "flag"
    "fmt"
    "time"
)

```

Второй фрагмент `create.go` содержит следующий код Go:

```
func main() {
    n := flag.Int("n", 10, "Number of goroutines")
    flag.Parse()

    count := *n
    fmt.Printf("Going to create %d goroutines.\n", count)
```

В этом коде считывается значение аргумента командной строки `n`, который определяет количество создаваемых горутин. Если аргумент командной строки `n` не задан, то значение переменной `n` будет равно 10.

Третья часть кода `create.go` выглядит так:

```
for i := 0; i < count; i++ {
    go func(x int) {
        fmt.Printf("%d ", x)
    }(i)
}
```

Цикл `for` используется для создания желаемого количества горутин. Еще раз повторю: следует помнить, что вы не можете делать какие-либо предположения о порядке, в котором они будут созданы и выполнены.

Последняя часть кода Go из `create.go` выглядит так:

```
time.Sleep(time.Second)
fmt.Println("\nExiting...")
}
```

Цель оператора `time.Sleep()` — дать горутинам достаточно времени, чтобы они успели завершить свою работу и их результаты можно было увидеть на экране. В реальной программе оператор `time.Sleep()` не нужен, так как вы захотите завершить программу как можно скорее. Кроме того, вы вскоре познакомитесь с более эффективным методом, позволяющим сделать так, чтобы программа ожидала завершения различных горутин, прежде чем завершить работу и вернуть результат функции `main()`.

Выполнение `create.go` несколько раз подряд приводит к результатам такого вида:

```
$ go run create.go -n 100
Going to create 100 goroutines.
5 3 2 4 19 9 0 1 7 11 10 12 13 14 15 31 16 20 17 22 8 18 28 29 21 52 30 45
25 24 49 38 41 46 6 56 57 54 23 26 53 27 59 47 69 66 51 44 71 48 74 33 35
73 39 37 58 40 50 78 85 86 90 67 72 91 32 64 65 95 75 97 99 93 36 60 34 77
94 61 88 89 83 84 43 80 82 87 81 68 92 62 55 98 96 63 76 79 42 70
Exiting...
$ go run create.go -n 100
Going to create 100 goroutines.
2 5 3 16 6 7 8 9 1 22 10 12 13 17 11 18 15 14 19 20 31 23 26 21 29 24 30 25
37 32 36 38 35 33 45 41 43 42 40 39 34 44 48 46 47 56 53 50 0 49 55 59 58
28 54 27 60 4 57 51 52 64 61 65 72 62 63 67 69 66 74 73 71 75 89 70 76 84
85 68 79 80 93 97 83 82 99 78 88 91 92 77 81 95 94 98 87 90 96 86
Exiting...
```

Здесь вы снова видите, что выходные данные программы являются недетерминированными и неупорядоченными в том смысле, что необходимо просмотреть все выходные данные, чтобы найти то, что вам нужно. Кроме того, если не подобрать подходящую задержку при вызове `time.Sleep()`, то мы вообще не увидим результат выполнения горутин. На данный момент нам подошел вариант с `time.Second`, но в будущем такой код может привести к неприятным и непредсказуемым ошибкам.

В следующем разделе показано, как предоставить горутинам достаточно времени, чтобы они успели выполнить свою работу до завершения программы, без необходимости вызывать `time.Sleep()`.

Как дождаться завершения горутин, прежде чем закончить программу

В этом разделе показано, как предотвратить завершение функции `main()`, пока она ожидает завершения своих горутин, с помощью пакета `sync`. Логика программы `syncGo.go` основана на коде из `create.go`, который рассмотрен в предыдущем разделе.

Первая часть `syncGo.go` выглядит так:

```
package main
```

```
import (  
    "flag"  
    "fmt"  
    "sync"  
)
```

Как видим, нам не требуется импортировать и применять пакет `time`, поскольку мы будем использовать функциональность пакета `sync` и ожидать столько времени, сколько необходимо для завершения всех горутин.



В главе 10 вы познакомитесь с двумя технологиями, позволяющими отменить выполнение горутин, если они занимают больше времени, чем рассчитывалось.

Второй фрагмент `syncGo.go` содержит следующий код Go:

```
func main() {  
    n := flag.Int("n", 20, "Number of goroutines")  
    flag.Parse()  
    count := *n  
    fmt.Printf("Going to create %d goroutines.\n", count)  
  
    var waitGroup sync.WaitGroup
```


В этом коде Go мы определяем переменную `sync.WaitGroup`. Если вы заглянете в исходный код Go-пакета `sync`, а именно в файл `waitgroup.go`, который находится в каталоге `sync`, то увидите, что тип `sync.WaitGroup` — это не что иное, как структура с тремя полями:

```
type WaitGroup struct {
    noCopy
    state1 [12]byte
    sema   uint32
}
```

Как работают переменные `sync.WaitGroup`, станет более понятно, когда мы увидим результат работы `syncGo.go`. Количество горутин, принадлежащих группе `sync.WaitGroup`, определяется одним или несколькими вызовами функции `sync.Add()`.

Третья часть `syncGo.go` содержит следующий код Go:

```
fmt.Printf("#v\n", waitGroup)
for i := 0; i < count; i++ {
    waitGroup.Add(1)
    go func(x int) {
        defer waitGroup.Done()
        fmt.Printf("%d ", x)
    }(i)
}
```

Здесь мы создаем желаемое количество горутин, используя цикл `for`. (Вместо этого можно применять несколько последовательных операторов Go.)

Каждый вызов `sync.Add()` увеличивает счетчик в переменной `sync.WaitGroup` на единицу. Обратите внимание: очень важно вызвать `sync.Add(1)` перед оператором `go`, чтобы предотвратить возможное *состояние гонки*. Когда горутина завершает работу, выполняется функция `sync.Done()`, которая уменьшает тот же счетчик на единицу.

Последняя часть кода `syncGo.go` выглядит так:

```
fmt.Printf("#v\n", waitGroup)
waitGroup.Wait()
fmt.Println("\nExiting...")
}
```

Вызов `sync.Wait()` блокируется до тех пор, пока значение счетчика в соответствующей переменной `sync.WaitGroup` не станет равным нулю, что дает возможность всем горутинам завершить работу.

Выполнение `syncGo.go` приведет к результатам такого вида:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
```

```

sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x14, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 7 8 9 10 11 12 13 14 15 16 17 0 1 2 5 18 4 6 3
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
1 0 4 5 17 7 8 9 10 11 12 13 2 sync.WaitGroup{noCopy:sync.noCopy{},
state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x17, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
sema:0x0}
29 15 6 27 24 25 16 22 14 23 18 26 3 19 20 28 21
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x1e,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
29 1 7 8 2 9 10 11 12 4 13 15 0 6 5 22 25 23 16 28 26 20 19 24 21 14 3 17 18 27
Exiting...

```

Результаты `syncGo.go` по-прежнему меняются от запуска к запуску, особенно при большом количестве горутин. В большинстве случаев это приемлемо; однако бывают ситуации, когда такое поведение нежелательно. Кроме того, при числе горутин, равном 30, некоторые из них завершили работу прежде, чем был выполнен второй оператор `fmt.Printf("%#v\n", waitGroup)`. Наконец, обратите внимание: один из элементов поля `state1` в `sync.WaitGroup` содержит счетчик, который увеличивается и уменьшается в соответствии с вызовами `sync.Add()` и `sync.Done()`.

Что происходит, если количество вызовов `Add()` и `Done()` не совпадает

Когда количество вызовов `sync.Add()` и `sync.Done()` одинаково, в программе все хорошо. Однако в этом пункте вы увидите, что случится, если два числа не совпадают.

Если вызовов `sync.Add()` было выполнено больше, чем вызовов `sync.Done()`, то, поставив оператор `waitGroup.Add(1)` перед первым оператором `fmt.Printf("%#v\n", waitGroup)` в программе `syncGo.go` и вызвав команду `go run`, получим следующий результат:

```

$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x1, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x15,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}

```

```

19 10 11 12 13 17 18 8 5 4 6 14 1 0 7 3 2 15 9 16 fatal error: all
goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200120bc)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:28 +0x2d7
exit status 2

```

Сообщение об ошибке говорит само за себя: `fatal error: all goroutines are asleep - deadlock!` Так случилось потому, что мы дали программе указание ждать завершения $n+1$ горутин, вызвав функцию `sync.Add(1)` $n+1$ раз, в то время как всего n горутин выполнили n операторов `sync.Done()`. В результате вызов `sync.Wait()` будет бесконечно и безуспешно ожидать одного или нескольких вызовов `sync.Done()` — это, очевидно, тупиковая ситуация.

Если же вызовов `sync.Add()` было выполнено меньше, чем вызовов `sync.Done()`, которые можно эмулировать, добавив в программе `syncGo.go` оператор `waitGroup.Done()` после цикла `for`, то результат команды `go run` будет примерно таким:

```

$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x12,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 6 1 2 9 7 8 15 13 0 14 16 17 3 11 4 5 12 18 10 panic: sync: negative
WaitGroup counter
goroutine 22 [running]:
sync.(*WaitGroup).Add(0xc4200120b0, 0xffffffffffffffff)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:75 +0x134
sync.(*WaitGroup).Done(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:100 +0x34
main.main.func1(0xc4200120b0, 0x11)
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:25 +0xd8
created by main.main
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:21 +0x206
exit status 2

```

Опять же причина проблемы изложена предельно ясно: `panic: sync: negative WaitGroup counter`.

Несмотря на то что в обоих случаях сообщения об ошибках весьма содержательны и помогут решить реальную проблему, следует быть очень осторожным с количеством вызовов `sync.Add()` и `sync.Done()`, которые мы вводим в свои программы. Кроме того, обратите внимание, что при второй ошибке (`panic: sync: negative WaitGroup counter`) проблема может проявляться не всегда.

Каналы

Канал — это механизм коммуникации, который, помимо прочего, позволяет обмениваться данными между горутинами.

Однако здесь есть ряд правил. Во-первых, каждый канал позволяет обмениваться данными определенного типа, который называется *типом элемента* канала. Во-вторых, для правильной работы канала необходим кто-то, кто будет получать то, что отправляется через канал. Чтобы создать новый канал, нужно использовать ключевое слово `chan`, а чтобы его закрыть — вызвать функцию `close()`.

Наконец, еще одна очень важная деталь: используя канал в качестве аргумента функции, можно указать его направление — будет ли этот канал использоваться для отправки или получения данных. Я считаю, что, если назначение канала заранее известно, следует использовать эту возможность, поскольку это сделает программы надежнее и безопаснее. Вы не сможете случайно отправить данные в канал, из которого можно только получать данные, или получать данные из канала, в который данные можно только отправлять. В результате, если вы объявите, что канал — аргумент функции будет использоваться только для чтения, и попытаетесь записать в него данные, то получите сообщение об ошибке, которое, скорее всего, избавит вас от неприятных программных дефектов. Мы еще вернемся к этому вопросу позже.



В этой главе вы узнаете много нового о каналах. Однако, чтобы в полной мере оценить всю мощь и гибкость, которые каналы предоставляют разработчику Go, вам придется подождать до главы 10.

Запись в канал

Код, представленный в этом подразделе, покажет вам, как записывать данные в канал. Записать значение `x` в канал `c` очень просто: достаточно написать код `c <- x`. Стрелка показывает направление передачи значения, так что у вас не возникнет проблем с этим утверждением, если и `x`, и `c` имеют одинаковый тип. Пример кода, который рассмотрен в данном разделе, хранится в файле `writeCh.go`. Разделим его на три части.

Первый фрагмент кода `writeCh.go` выглядит так:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println(x)
```

```

    c <- x
    close(c)
    fmt.Println(x)
}

```

Ключевое слово `chan` используется для объявления того, что аргумент функции `c` является каналом, и после него должен быть указан тип канала (`int`). Оператор `c <- x` позволяет записать значение `x` в канал `c`, а функция `close()` закрывает канал, то есть делает невозможным запись в него.

Вторая часть `writeCh.go` содержит следующий код Go:

```

func main() {
    c := make(chan int)

```

В этом коде вы видите определение переменной канала с именем `c`, использование функции `make()` — впервые в этой главе, — а также ключевого слова `chan`. С каждым каналом связан определенный тип данных, здесь это `int`.

Остальной код `writeCh.go` выглядит так:

```

    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
}

```

Здесь выполняется функция `writeToChannel()` как горутина и вызывается `time.Sleep()`, чтобы предоставить достаточно времени для выполнения функции `writeToChannel()`.

Выполнение `writeCh.go` приведет к следующему результату:

```

$ go run writeCh.go
10

```

Странно, что функция `writeToChannel()` выводит это значение только один раз. Причиной такого неожиданного результата является то, что второй оператор `fmt.Println(x)` никогда не выполняется. Это объясняется очень просто, если понимать, как работают каналы: оператор `c <- x` блокирует выполнение остальной части функции `writeToChannel()`, так как никто не читает то, что было записано в канал `c`. Поэтому, когда оператор `time.Sleep(1 * time.Second)` завершает работу, программа заканчивает выполнение, не ожидая завершения `writeToChannel()`.

Как читать данные из канала, вы узнаете в следующем подразделе.

Чтение из канала

В этом подразделе вы научитесь читать данные из канала. Чтобы прочитать из канала одно значение с именем `c`, нужно выполнить операцию `<-c`. В этом случае операция направлена из канала наружу.

Программа, которую мы рассмотрим, чтобы помочь вам понять, как читать данные из канала, называется `readCh.go`. Разделим ее на три части.

Первый фрагмент `readCh.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println("1", x)
    c <- x
    close(c)
    fmt.Println("2", x)
}
```

Реализация функции `writeToChannel()` такая же, как и раньше.

Вторая часть `readCh.go` выглядит так:

```
func main() {
    c := make(chan int)
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
    fmt.Println("Read:", <-c)
    time.Sleep(1 * time.Second)
}
```

В этом коде мы читаем данные из канала `c`, используя нотацию `<-c`. Если мы захотим сохранить это значение в переменной с именем `k`, вместо того чтобы просто вывести его на экран, мы можем использовать запись `k := <-c`. Второй оператор `time.Sleep(1 * time.Second)` предоставляет время для чтения из канала.

Последняя часть кода `readCh.go` содержит следующий код Go:

```
_ , ok := <-c
if ok {
    fmt.Println("Channel is open!")
} else {
    fmt.Println("Channel is closed!")
}
}
```

В этом коде показан прием, позволяющий определить, открыт ли данный канал. Этот код Go работает нормально, когда канал закрыт; но если канал открыт, то вследствие использования символа `_` в выражении `_ , ok := <-c` код Go отбросит прочитанное из канала значение. Если хотите сохранить значение, полученное из канала, когда он открыт, используйте вместо `_` правильное имя переменной.

Выполнение `readCh.go` приведет к следующим результатам:

```
$ go run readCh.go
1 10
Read: 10
2 10
```

```
Channel is closed!
$ go run readCh.go
1 10
2 10
Read: 10
Channel is closed!
```

Эти результаты все еще не являются детерминированными, зато теперь выполняются оба оператора `fmt.Println(x)` функции `writeToChannel()`, поскольку при чтении из канала данный канал разблокируется.

Прием данных из закрытого канала

В этом подразделе вы узнаете, что происходит, если попытаться прочитать данные из закрытого канала. Воспользуемся кодом Go из файла `readClose.go`, который мы разделим на две части.

Первая часть `readClose.go` выглядит так:

```
package main

import (
    "fmt"
)

func main() {
    willClose := make(chan int, 10)

    willClose <- -1
    willClose <- 0
    willClose <- 2

    <-willClose
    <-willClose
    <-willClose
```

В этой части программы мы создаем новый канал типа `int` с именем `willClose`, записываем в него данные и считываем все эти данные, ничего не делая с ними.

Вторая часть `readClose.go` содержит следующий код:

```
    close(willClose)
    read := <-willClose
    fmt.Println(read)
}
```

Здесь мы закрываем канал `willClose` и пытаемся прочитать из него данные, не смотря на то что мы очистили этот канал в предыдущей части программы.

Выполнение `readClose.go` приведет к следующим результатам:

```
$ go run readClose.go
0
```

Это означает, что при чтении данных из закрытого канала возвращается нулевое значение соответствующего типа, в данном случае 0.¹

Каналы как аргументы функции

Хоть мы и не обращались к этому свойству Go ни в `readCh.go`, ни в `writeCh.go`, Go позволяет указать направление канала при применении его в качестве аргумента функции, сообщив ей таким образом, будет ли канал использоваться для чтения или для записи. Эти два типа каналов называются однонаправленными каналами, тогда как по умолчанию каналы являются двунаправленными.

Рассмотрим код Go следующих двух функций:

```
func f1(c chan int, x int) {
    fmt.Println(x)
    c <- x
}

func f2(c chan<- int, x int) {
    fmt.Println(x)
    c <- x
}
```

Хотя обе функции реализуют одну и ту же функциональность, их определения немного различаются. Разница заключается в символе `<-`, который стоит справа от ключевого слова `chan` в определении функции `f2()`. Этот символ означает, что канал `c` может использоваться только для записи. Если в коде функции Go мы попытаемся прочитать данные из канала, предназначенного только для записи (только для отправки данных), то компилятор Go выдаст следующее сообщение об ошибке:

```
# command-line-arguments
a.go:19:11: invalid operation: range in (receive from send-only type chan<-int)
```

Подобным же образом можно написать следующие определения функций:

```
func f1(out chan<- int64, in <-chan int64) {
    fmt.Println(x)
    c <- x
}

func f2(out chan int64, in chan int64) {
    fmt.Println(x)
    c <- x
}
```

В определении функции `f2()` присутствует канал, доступный только для чтения, который называется `in`, и канал, доступный только для записи, с именем `out`.

¹ Попытка записи в закрытый канал приведет к панике.

Если попытаться случайно записать данные в канал, который является аргументом функции и предназначен только для чтения (*канал только для приема данных*), и закрыть его, то получим следующее сообщение об ошибке:

```
# command-line-arguments
a.go:13:7: invalid operation: out <- i (send to receive-only type <-chan int)
a.go:15:7: invalid operation: close(out) (cannot close receive-only channel)
```

Конвейеры

Конвейер — это виртуальный метод, предназначенный для соединения горутин и каналов, так что выходные данные одной горутины становятся входными данными для другой горутины, а для передачи данных используются каналы.

Одним из преимуществ использования конвейеров является наличие постоянного потока данных, так что никакие горутины и каналы не должны ожидать, пока завершится все остальное, чтобы можно было начать выполнение. Кроме того, мы используем меньше переменных и, следовательно, меньше памяти, потому что не приходится сохранять все данные в виде переменных. Наконец, использование конвейеров упрощает разработку программ и делает их удобнее для поддержки.

Рассмотрим использование конвейеров на примере кода `pipeline.go`, разделив программу на шесть частей. Задача, выполняемая программой `pipeline.go`, состоит в том, чтобы генерировать случайные числа в заданном диапазоне и останавливаться, когда любое число в этой последовательности встретится во второй раз. При этом, прежде чем завершить работу, программа выводит на экран сумму всех случайных чисел, сгенерированных до того момента, когда впервые появилось повторяющееся случайное число. Для подключения каналов в программе нам понадобятся три функции. В этих функциях заключается логика программы. Данные будут передаваться по каналам конвейера.

Программа будет иметь два канала. Первый канал (канал *A*) будет использоваться для получения случайных чисел из первой функции и передачи их во вторую функцию. Второй канал (канал *B*) будет использоваться второй функцией для передачи приемлемых случайных чисел в третью функцию. Третья функция будет отвечать за получение данных из канала *B*, вычисления и представление результатов.

Первый фрагмент `pipeline.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

```
var CLOSEA = false

var DATA = make(map[int]bool)
```

Поскольку `second()` нужен способ сообщить функции `first()` о закрытии первого канала, я буду использовать для этого глобальную переменную с именем `CLOSEA`. Переменная `CLOSEA` проверяется только функцией `first()` и может быть изменена только `second()`.

Вторая часть `pipeline.go` содержит следующий код Go:

```
func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func first(min, max int, out chan<- int) {
    for {
        if CLOSEA {
            close(out)
            return
        }
        out <- random(min, max)
    }
}
```

В этом коде представлена реализация двух функций: `random()` и `first()`. С функцией `random()`, которая генерирует случайные числа в заданном диапазоне, вы уже знакомы. Главный интерес представляет функция `first()`, поскольку она продолжает выполнять цикл `for` до тех пор, пока логическая переменная (`CLOSEA`) не станет равной `true`, после чего функция закрывает свой канал `out`.

Третий фрагмент кода `pipeline.go` выглядит так:

```
func second(out chan<- int, in <-chan int) {
    for x := range in {
        fmt.Print(x, " ")
        _, ok := DATA[x]
        if ok {
            CLOSEA = true
        } else {
            DATA[x] = true
            out <- x
        }
    }
    fmt.Println()
    close(out)
}
```

Функция `second()` получает данные из канала `in` и передает их в канал `out`. Однако, как только `second()` обнаруживает случайное число, которое уже существует в отображении `DATA`, она присваивает глобальной переменной `CLOSEA` значение `true`

и прекращает передачу любых других чисел в канал `out`. После этого функция закрывает канал `out`.



Цикл `range` читает данные из канала и автоматически завершает работу, когда канал закрывается.

В четвертой части `pipeline.go` содержится следующий код Go:

```
func third(in <-chan int) {
    var sum int
    sum = 0
    for x2 := range in {
        sum = sum + x2
    }
    fmt.Printf("The sum of the random numbers is %d\n", sum)
}
```

Функция `third()` считывает данные из канала, переданного функции в качестве аргумента. Когда `second()` закрывает этот канал, цикл `for` прекращает получать данные и функция выводит результат на экран. На этом этапе вы уже, вероятно, поняли, что здесь всем управляет `second()`.

Пятый фрагмент кода `pipeline.go` выглядит так:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    n1, _ := strconv.Atoi(os.Args[1])
    n2, _ := strconv.Atoi(os.Args[2])

    if n1 > n2 {
        fmt.Printf("%d should be smaller than %d\n", n1, n2)
        return
    }
}
```

Этот код используется для работы с аргументами командной строки программы.

Последняя часть программы `pipeline.go` выглядит следующим образом:

```
rand.Seed(time.Now().UnixNano())
A := make(chan int)
B := make(chan int)

go first(n1, n2, A)
go second(B, A)
third(B)
}
```

Здесь мы определяем требуемые каналы и выполняем две горутины и одну функцию. Функция `third()` не позволяет `main()` немедленно завершиться, поскольку она не выполняется как горутина.

Выполнение `pipeline.go` приведет к результатам такого вида:

```
$ go run pipeline.go 1 10
2 2
The sum of the random numbers is 2
$ go run pipeline.go 1 10
9 7 8 4 3 3
The sum of the random numbers is 31
$ go run pipeline.go 1 10
1 6 9 7 1
The sum of the random numbers is 23
$ go run pipeline.go 10 20
16 19 16
The sum of the random numbers is 35
$ go run pipeline.go 10 20
10 16 17 11 15 10
The sum of the random numbers is 69
$ go run pipeline.go 10 20
12 11 14 15 10 15
The sum of the random numbers is 62
```

Здесь важно, что, хотя функция `first()` генерирует случайные числа в своем темпе, а `second()` выводит их все на экран, нежелательные (повторяющиеся) случайные числа не будут отправлены в `third()` и, следовательно, не будут включены в итоговую сумму.

Состояние гонки

Код `pipeline.go` неидеален: он содержит логическую ошибку, которая в терминологии конкурентности называется состоянием гонки. Чтобы выявить эту ошибку, нужно выполнить следующую команду:

```
$ go run -race pipeline.go 1 10
2 2 =====
WARNING: DATA RACE
Write at 0x00000122bae8 by goroutine 7:
    main.second()
        /Users/mtsouk/ch09/pipeline.go:34 +0x15c
Previous read at 0x00000122bae8 by goroutine 6:
    main.first()
        /Users/mtsouk/ch09/pipeline.go:21 +0xa3
Goroutine 7 (running) created at:
    main.main()
        /Users/mtsouk/ch09/pipeline.go:72 +0x2a1
```

```
Goroutine 6 (running) created at:
  main.main()
    /Users/mtsouk/ch09/pipeline.go:71 +0x275
=====
2
The sum of the random numbers is 2.
Found 1 data race(s)
exit status 66
```

Проблема заключается в том, что горутина, выполняющая функцию `second()`, может изменить значение переменной `CLOSEA` в тот момент, когда `first()` будет читать эту переменную. Поскольку невозможно определить, какое из данных событий произойдет первым, а какое — вторым, такое состояние считается состоянием гонки. Чтобы устранить состояние гонки, необходимо использовать сигнальный канал и ключевое слово `select`.



Подробнее об условиях гонки, сигнальных каналах и ключевом слове `select` вы прочитаете в главе 10.

Чтобы показать, какие изменения мы внесли в `pipeline.go`, воспользуемся командой `diff(1)` — новая версия программы называется `plNoRace.go`:

```
$ diff pipeline.go plNoRace.go
14a15,16
> var signal chan struct{}
>
21c23,24
<   if CLOSEA {
---
>   select {
>   case <-signal:
23a27
>   case out <- random(min, max):
25d28
<   out <- random(min, max)
31d33
<   fmt.Print(x, " ")
34c36
<       CLOSEA = true
---
>       signal <- struct{}{}
35a38
>       fmt.Print(x, " ")
61d63
<
66a69,70
>   signal = make(chan struct{})
>
```

Для того чтобы проверить логическую корректность `plNoRace.go`, можно воспользоваться следующей командой:

```
$ go run -race plNoRace.go 1 10
8 1 4 9 3
The sum of the random numbers is 25.
```

Сравнение моделей конкурентности в Go и Rust

Rust — очень популярный язык системного программирования, который также поддерживает конкурентное программирование. Далее кратко изложены некоторые характеристики Rust и модель конкурентности Rust:

- ❑ потоки Rust — это потоки UNIX, следовательно, они тяжелые, но позволяют делать много интересного;
- ❑ Rust поддерживает конкурентность на основе *обмена сообщениями* и на основе *совместного использования состояний* — то же самое, что в Go реализовано посредством каналов, мьютексов и общих переменных;
- ❑ благодаря строгой типизации и системе владения Rust обеспечивает безопасное изменяемое состояние потока. Эти правила применяются компилятором Rust;
- ❑ существуют структуры Rust, допускающие совместно используемые состояния;
- ❑ если поток начинает вести себя некорректно, с системой не случится фатальный сбой. Это контролируемая ситуация, которая может быть обработана;
- ❑ язык программирования Rust продолжает развиваться, из-за чего некоторые отказываются его использовать, чтобы не пришлось постоянно вносить изменения в существующий код.

Таким образом, у Rust есть гибкая модель конкурентности — еще более гибкая, чем в Go. Однако за эту гибкость приходится платить — необходимо мириться с Rust и его особенностями.

Сравнение моделей конкурентности в Go и Erlang

Erlang — это очень популярный язык конкурентного функционального программирования, разработанный для создания высокодоступных приложений. Далее приведены основные характеристики языка Erlang и реализованной в нем модели конкурентности:

- ❑ Erlang является сформировавшимся и протестированным языком программирования. Это также относится к его модели конкурентности;

- ❑ если вам не нравится, как работает код на Erlang, вы всегда можете попробовать *Elixir*, который основан на Erlang и использует Erlang VM, — его код выглядит более приятно;
- ❑ в Erlang используется только асинхронная коммуникация;
- ❑ в Erlang реализована система обработки ошибок, позволяющая создавать надежные конкурентные системы;
- ❑ процессы Erlang способны привести к сбою системы, но если этот сбой обрабатывается должным образом, то система может продолжить работать без проблем;
- ❑ как и в случае с горутинами, процессы Erlang являются изолированными, у них нет общих состояний. Единственный способ взаимодействия между процессами Erlang — передача сообщений;
- ❑ потоки в Erlang такие же легкие, как и горутины. Это означает, что можно создавать сколько угодно процессов.

Таким образом, Erlang и Elixir являются надежными и высокодоступными системами, если вас устраивает принцип конкурентности, реализованный в Erlang.

Дополнительные ресурсы

Советую вам посетить следующие полезные ресурсы:

- ❑ страницу документации пакета `sync`, которая находится по адресу <https://golang.org/pkg/sync/>;
- ❑ веб-сайт Rust по адресу <https://www.rust-lang.org/>;
- ❑ веб-сайт Erlang по адресу <https://www.erlang.org/>;
- ❑ и снова страницу документации пакета `sync`. Обратите особое внимание на типы `sync.Mutex` и `sync.RWMutex`, которые появятся в следующей главе.

Упражнения

- ❑ Создайте конвейер, который бы читал текстовые файлы, вычислял количество вхождений заданной фразы в каждом текстовом файле и подсчитывал общее количество вхождений этой фразы во всех файлах.
- ❑ Создайте конвейер для вычисления суммы квадратов всех натуральных чисел в заданном диапазоне.
- ❑ Удалите из программы `simple.go` оператор `time.Sleep(1 * time.Second)` и посмотрите, что произойдет. Почему так происходит?

- ❑ Измените код Go в файле `pipeline.go` таким образом, чтобы получить конвейер из пяти функций и соответствующего количества каналов.
- ❑ Измените код Go в файле `pipeline.go` таким образом, чтобы узнать, что произойдет, если вы забудете закрыть канал `out` функции `first()`.

Резюме

В этой главе вы разобрались со множествами уникальных свойств Go, в том числе с горутинами, каналами и конвейерами. Кроме того, вы поняли, как предоставить горутинам достаточно времени, чтобы они успели завершить свою работу, используя функциональность, предлагаемую пакетом `sync`. Наконец, вы узнали, что каналы могут служить аргументами функций Go. Это позволяет разработчикам создавать конвейеры, по которым могут передаваться данные.

В следующей главе тема конкурентности в Go продолжится и вы познакомитесь с грозным ключевым словом `select`. Это слово помогает каналам Go выполнять много интересных задач, и я думаю, что вы будете просто поражены его мощностью.

Вы откроете для себя два способа, позволяющих отключить одну или несколько горутин, которые по какой-то причине остановили работу. Кроме того, вы узнаете о нулевых, сигнальных и буферизованных каналах, каналах каналов, а также о пакете `context`.

В главе 10 раскрыто понятие *общей памяти*, которая является традиционным способом обмена информацией между потоками в рамках одного процесса UNIX, что также применяется и к горутинам. Однако в среде программистов Go общая память менее популярна, поскольку Go предлагает более эффективные, безопасные и быстрые способы обмена данными между горутинами.

10

Конкурентность в Go: расширенные возможности

В предыдущей главе вы познакомились с горутинами, которые являются главным функционалом Go, а также с каналами и конвейерами. В этой главе мы продолжим с того места, на котором остановились в предыдущей, чтобы глубже изучить горутины и каналы, а также познакомиться с ключевым словом `select`, после чего перейдем к обсуждению общих переменных, а также типов `sync.Mutex` и `sync.RWMutex`.

В данной главе рассмотрены примеры кода, в которых продемонстрировано использование *сигнальных*, *буферизованных* и *нулевых каналов*, а также *каналов каналов*. Кроме того, вы узнаете о двух методах прерывания выполнения горутины, если оно превысило заданный лимит, потому что невозможно гарантировать, что все горутины будут завершены за желаемое время.

В завершение главы обсудим пакет `atomic`, состояние гонки, стандартный Go-пакет `context` и пул рабочих потоков.

Эта глава посвящена следующим темам:

- ❑ ключевое слово `select`;
- ❑ как работает планировщик Go;
- ❑ два метода, которые позволяют прервать выполнение программы, которая длится больше времени, чем ожидалось;
- ❑ сигнальные каналы;
- ❑ буферизованные каналы;
- ❑ нулевые каналы;
- ❑ управляющая горутина;
- ❑ общая память и мьютексы;
- ❑ типы `sync.Mutex` и `sync.RWMutex`;
- ❑ пакет `context` и его расширенная функциональность;
- ❑ пакет `atomic`;
- ❑ пулы обработчиков;
- ❑ распознавание состояния гонки.

И снова о планировщике Go

Планировщик отвечает за эффективное распределение работы, которую следует выполнить, между доступными ресурсами. В этом разделе мы гораздо глубже, чем в предыдущей главе, рассмотрим работу планировщика Go. Как вы уже знаете, Go работает с использованием *планировщика m:n* (или *планировщика M:N*). Планировщик Go оперирует горутинами, которые легче потоков ОС, используя при этом потоки ОС. Однако сначала — теория и определения полезных терминов.

В Go используется модель *конкурентности fork-join*. Слово *fork* (вилка) в названии модели говорит о том, что дочерняя ветвь может быть создана в любой точке программы. Аналогично слово *join* («объединение») говорит о том, что в некоторой точке дочерняя ветвь заканчивается и объединяется с родительской. Точками такого объединения, в частности, являются операторы `sync.Wait()` и каналы, которые собирают результаты выполнения горутин, тогда как каждая новая горутина создает дочернюю ветвь.



Фаза *fork* в модели *fork-join* и системный вызов `fork(2)` в C — это совершенно разные понятия.

Стратегия справедливого планирования довольно проста концептуально и имеет простую реализацию. Согласно этой стратегии, вся нагрузка должна равномерно распределяться между доступными процессорами. Поначалу такая стратегия может показаться идеальной, поскольку приходится учитывать не так уж много нюансов, при этом сохраняя одинаковую нагрузку всех процессоров. Однако на поверку оказывается, что это не совсем так, поскольку большинство распределенных задач обычно зависят друг от друга, из-за чего некоторые процессоры оказываются недостаточно загружены, или, что то же самое, одни процессоры используются интенсивнее, чем другие.

Горутина в Go — это *задача*, а все, что происходит после вызова горутин, является *продолжением*. В *стратегии перехвата работы*, используемой планировщиком Go, недостаточно загруженный (логический) процессор ищет дополнительную работу, которую выполняют другие процессоры. Найдя такие задания, он перехватывает их у другого процессора или нескольких процессоров — отсюда и название. Кроме того, алгоритм перехвата работы в Go перехватывает продолжения и ставит их в очередь. *Останавливающее соединение*, как следует из названия, — это место, в котором поток выполнения останавливается в точке соединения и начинает искать себе другую работу.

Несмотря на то что и при перехвате задач, и при перехвате продолжений имеет место останавливающее соединение, продолжения случаются чаще, чем задачи; поэтому алгоритм Go работает не с задачами, а с продолжениями.

Основным недостатком перехвата продолжений является то, что это требует от компилятора языка программирования дополнительной работы. К счастью, язык Go предоставляет эту дополнительную возможность, и поэтому в его алгоритме перехвата работы используется алгоритм *перехвата продолжений*.

Одним из преимуществ перехвата продолжений является то, что он дает одинаковые результаты при использовании функций вместо горутин или одного потока с несколькими горутинами. Это закономерно, так как в каждый момент в обоих случаях выполняется только что-то одно.

Теперь вернемся к используемому в Go алгоритму планирования *m:n*. Строго говоря, в у нас всегда есть *m* горутин, которые выполняются и, следовательно, запланированы для запуска в *n* потоках ОС, использующих не более `GOMAXPROCS` логических процессоров. О том, что такое `GOMAXPROCS`, вы скоро узнаете.

Планировщик Go работает с использованием трех основных типов сущностей: потоков ОС (*M*), которые связаны с используемой операционной системой, горутин (*G*) и логических процессоров (*P*). Количество процессоров, которые могут использоваться программой Go, определяется значением переменной среды `GOMAXPROCS`: в любое время число процессоров не может превышать `GOMAXPROCS`.

Этот момент проиллюстрирован на рис. 10.1.



Рис. 10.1. Так работает планировщик Go

Как видно на рис. 10.1, существует два вида очередей: глобальная и локальная, причем каждая локальная очередь присоединена к соответствующему ло-

гическому процессору. Горутины из глобальной очереди назначаются очередям логических процессоров, где они и выполняются. Таким образом, планировщик Go должен проверить глобальную очередь, чтобы избежать выполнения горутин, расположенных только в локальной очереди какого-либо из логических процессоров. Однако планировщик не проверяет глобальную очередь постоянно. Следовательно, у глобальной очереди нет преимуществ по сравнению с локальной.

Кроме того, каждый логический процессор может иметь несколько потоков и возможен перехват между локальными очередями доступных логических процессоров. Имейте в виду, что планировщик Go может при необходимости создавать дополнительные потоки ОС. Однако потоки ОС весьма затратны, поэтому слишком активные действия с ними могут замедлить работу приложений Go.

Помните, что использование большего количества горутин в программе не пагубно для повышения производительности, так как большое количество горутин в сочетании с многочисленными вызовами `sync.Add()`, `sync.Wait()` и `sync.Done()` может замедлить работу программы из-за дополнительных обслуживающих операций, которые должен выполнять планировщик Go.



Подобно большинству компонентов Go, планировщик Go постоянно развивается. Те, кто работает над планировщиком Go, все время стремятся повысить его производительность, внося небольшие изменения в его работу. Однако основные принципы остаются прежними.

Вам не нужно быть в курсе всего этого, чтобы писать код Go, использующий горутины. Но знание того, что происходит внутри кода, определенно поможет вам разобраться, в чем дело, если вдруг начнут происходить странные вещи или если вам станет интересно, как работает планировщик Go. Благодаря этому вы, безусловно, станете лучше как разработчик!

Переменная среды GOMAXPROCS

Переменная среды (и функция Go) `GOMAXPROCS` позволяет ограничить количество потоков операционной системы, которые могут одновременно выполнять код Go на уровне пользователя. Начиная с версии Go 1.5, по умолчанию значение `GOMAXPROCS` равно количеству логических ядер, доступных на UNIX-компьютере.

Если `GOMAXPROCS` меньше числа ядер на UNIX-компьютере, то это может привести к снижению производительности программы. Однако использование `GOMAXPROCS`, превышающего число доступных ядер, не обязательно заставит программы Go работать быстрее.

Чтобы программно узнать значение переменной среды `GOMAXPROCS`, нужно воспользоваться следующим кодом. Этот код вы найдете в программе с именем `maxprocs.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func getGOMAXPROCS() int {
    return runtime.GOMAXPROCS(0)
}

func main() {
    fmt.Printf("GOMAXPROCS: %d\n", getGOMAXPROCS())
}
```

Выполнение `maxprocs.go` на машине с процессором Intel i7 даст следующий результат:

```
$ go run maxprocs.go
GOMAXPROCS: 8
```

Тем не менее мы можем изменить этот результат, если изменим значение переменной среды `GOMAXPROCS` перед выполнением программы, выполнив следующие команды в UNIX-оболочке `bash(1)`:

```
$ go version
go version go1.12.3 darwin/amd64
$ export GOMAXPROCS=800; go run maxprocs.go
GOMAXPROCS: 800
$ export GOMAXPROCS=4; go run maxprocs.go
GOMAXPROCS: 4
```

Ключевое слово `select`

Как вы вскоре узнаете, ключевое слово `select` довольно мощное; оно может делать многое в самых разных ситуациях. Оператор `select` в Go похож на `switch`, но только для каналов. На практике это означает, что `select` позволяет горутине дожидаться завершения нескольких операций коммуникации. Поэтому основное преимущество `select` состоит в том, что этот оператор дает возможность работать с несколькими каналами, используя один блок `select`. В результате, построив соответствующие блоки `select`, можно выполнять неблокирующие операции с каналами.



Самая большая проблема при использовании нескольких каналов и ключевого слова `select` — это взаимоблокировки. Это означает, что необходимо быть особенно осторожными при проектировании и разработке, чтобы избежать таких взаимоблокировок.

Мы рассмотрим использование ключевого слова `select` на примере программы `select.go`. Разделим ее на пять частей. Первая часть `select.go` содержит следующий код Go:

```
package main
```

```
import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

Вторая часть кода `select.go` выглядит так:

```
func gen(min, max int, createNumber chan int, end chan bool) {
    for {
        select {
            case createNumber <- rand.Intn(max-min) + min:
            case <-end:
                close(end)
                return
            case <-time.After(4 * time.Second):
                fmt.Println("\ntime.After()!")
        }
    }
}
```

Итак, что в действительности происходит в коде этого блока `select`? В данном операторе `select` предусмотрено три варианта. Обратите внимание, что операторы `select` не требуют ветки `default`. В качестве «умной» ветки `default` в этом коде можно рассматривать третью ветку оператора `select`. Так происходит потому, что `time.After()` ожидает истечения заданного интервала, после чего передает значение текущего времени по возвращаемому каналу — это разблокирует оператор `select`, если все остальные каналы по какой-либо причине окажутся заблокированными.

Оператор `select` не выполняется последовательно, так как все его каналы проверяются одновременно. Если ни один из каналов, указанных в операторе `select`, не доступен, то оператор `select` будет заблокирован, пока не освободится один из каналов. Если доступны сразу несколько каналов оператора `select`,

то среда выполнения Go сделает случайный выбор из набора этих доступных каналов.

Среда выполнения Go пытается делать случайный выбор между доступными каналами как можно более равномерным и справедливым.

Третья часть `select.go` выглядит так:

```
func main() {
    rand.Seed(time.Now().Unix())
    createNumber := make(chan int)
    end := make(chan bool)
    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }
}
```

Четвертая часть программы `select.go` содержит следующий код Go:

```
n, _ := strconv.Atoi(os.Args[1])
fmt.Printf("Going to create %d random numbers.\n", n)
go gen(0, 2*n, createNumber, end)
for i := 0; i < n; i++ {
    fmt.Printf("%d ", <-createNumber)
}
}
```



Значение `error`, возвращаемое функцией `strconv.Atoi()`, не проверяется из соображения экономии места. Никогда не поступайте так в реальных приложениях.

Оставшийся код Go программы `select.go` выглядит так:

```
time.Sleep(5 * time.Second)
fmt.Println("Exiting...")
end <- true
}
```

Главное назначение оператора `time.Sleep(5 * time.Second)` — предоставить функциям `time.After()` и `gen()` достаточно времени, чтобы отработать, вернуть результат и таким образом активизировать соответствующую ветвь оператора `select`.

Последний оператор функции `main()` завершает программу, активизируя ветвь `case <-end` оператора `select` в `gen()` и реализуя соответствующий код Go.

Выполнение `select.go` приведет к следующему:

```
$ go run select.go 10
Going to create 10 random numbers.
13 17 8 14 19 9 2 0 19 5
time.After()!
Exiting...
```



Главным преимуществом оператора `select` является то, что он может подключать, распределять нагрузку и управлять несколькими каналами. Поскольку каналы служат для связи между горутинами, `select` соединяет каналы, которые соединяют горутины. Таким образом, оператор `select` является одной из наиболее важных, если не самой важной, частью модели конкурентности в Go.

Принудительное завершение горутины

В этом разделе представлены два очень важных метода, позволяющих прервать горутину, если она выполняется слишком долго. Проще говоря, эти два способа избавят вас от необходимости ждать до бесконечности, пока программа завершит работу, и предоставят полный контроль над временем, в течение которого вы готовы ожидать завершения горутины. В обоих методах используются возможности удобного ключевого слова `select` в сочетании с функцией `time.After()`, которая уже встречалась в предыдущем разделе.

Принудительное завершение горутины, способ 1

Исходный код с примером первого способа хранится в файле `timeOut1.go`. Рассмотрим его, разделив на четыре части.

Первая часть `timeOut1.go` содержит следующий код Go:

```
package main
```

```
import (  
    "fmt"  
    "time"  
)
```

Второй фрагмент кода `timeOut1.go` выглядит так:

```
func main() {  
    c1 := make(chan string)  
    go func() {  
        time.Sleep(time.Second * 3)  
        c1 <- "c1 OK"  
    }()  
}
```

Вызов `time.Sleep()` используется для эмуляции времени, которое обычно занимает выполнение функции. В данном случае выполнение анонимной функции, которая

запускается как горутина, займет около трех секунд (`time.Second * 3`), после чего записывается сообщение в канал `c1`.

Третий фрагмент `timeOut1.go` содержит следующий код Go:

```
select {
case res := <-c1:
    fmt.Println(res)
case <-time.After(time.Second * 1):
    fmt.Println("timeout c1")
}
```

Цель вызова функции `time.After()` — ожидать в течение заданного времени. В данном случае нас интересует не значение, возвращаемое функцией `time.After()`, а факт завершения вызова функции `time.After()`, который означает, что доступное время ожидания истекло. В этом случае, поскольку значение, передаваемое в функцию `time.After()`, меньше, чем значение, используемое в вызове `time.Sleep()`, который был выполнен как горутина в предыдущем фрагменте кода, вы, скорее всего, получите сообщение о превышении времени ожидания.

Остальной код из `timeOut1.go` выглядит так:

```
c2 := make(chan string)
go func() {
    time.Sleep(3 * time.Second)
    c2 <- "c2 OK"
}()

select {
case res := <-c2:
    fmt.Println(res)
case <-time.After(4 * time.Second):
    fmt.Println("timeout c2")
}
}
```

В этом коде запускается горутина, выполнение которой занимает около трех секунд из-за вызова `time.Sleep()`. Затем определяется интервал ожидания длительностью четыре секунды с использованием вызова `time.After(4 * time.Second)`. Если функция `time.After(4 * time.Second)` завершится позже, чем будет получено значение из канала `c2`, указанного в первом варианте блока `select`, то время выполнения горoutines не будет превышено. В противном случае она превысит выделенный ей лимит времени! Однако при этом значение `time.After()` обеспечивает достаточно времени для возврата к вызову `time.Sleep()`, поэтому вы, скорее всего, не получите сообщение о превышении временного лимита.

Выполнение `timeOut1.go` приведет к результатам следующего вида:

```
$ go run timeOut1.go
timeout c1
c2 OK
```

Как и ожидалось, первая горутина не успела завершиться, в то время как у второй было достаточно времени, чтобы закончить работу.

Принудительное завершение горутины, способ 2

Исходный код, демонстрирующий второй способ прерывания горутин, хранится в файле `timeOut2.go`. Рассмотрим его, разделив на пять частей. На этот раз интервал ожидания будет предоставлен в качестве аргумента командной строки программы.

Первая часть `timeOut2.go` выглядит так:

```
package main
```

```
import (  
    "fmt"  
    "os"  
    "strconv"  
    "sync"  
    "time"  
)
```

Во втором фрагменте `timeOut2.go` содержится следующий код Go:

```
func timeout(w *sync.WaitGroup, t time.Duration) bool {  
    temp := make(chan int)  
    go func() {  
        defer close(temp)  
        time.Sleep(5 * time.Second)  
        w.Wait()  
    }()  
  
    select {  
    case <-temp:  
        return false  
    case <-time.After(t):  
        return true  
    }  
}
```

В этом коде временной интервал, который будет использоваться при вызове `time.After()`, является аргументом функции `timeout()`, следовательно, является переменной величиной. Как и в прошлый раз, логика прерывания в случае превышения временного лимита реализована в блоке `select`. Кроме того, вызов `w.Wait()` заставит функцию `timeout()` бесконечно ожидать соответствующую функцию `sync.Done()`, чтобы завершить работу. Когда `w.Wait()` закончит работу, будет выполнена первая ветвь оператора `select`.

Третья часть кода `timeOut2.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need a time duration!")
        return
    }

    var w sync.WaitGroup
    w.Add(1)

    t, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Четвертая часть программы `timeOut2.go`:

```
duration := time.Duration(int32(t)) * time.Millisecond
fmt.Printf("Timeout period is %s\n", duration)

if timeout(&w, duration) {
    fmt.Println("Timed out!")
} else {
    fmt.Println("OK!")
}
```

Функция `time.Duration()` преобразует целочисленное значение в переменную `time.Duration`, которую мы будем использовать позже.

Оставшийся код Go из `timeOut2.go` выглядит так:

```
w.Done()
if timeout(&w, duration) {
    fmt.Println("Timed out!")
} else {
    fmt.Println("OK!")
}
}
```

После завершения вызова `w.Done()` предшествующая ему функция `timeout()` также завершится. Однако у второго вызова `timeout()` нет оператора `sync.Done()`, которого она могла бы ожидать.

Выполнение `timeOut2.go` приведет к результатам следующего вида:

```
$ go run timeOut2.go 10000
Timeout period is 10s
Timed out!
OK!
```

При таком выполнении `timeOut2.go` период ожидания длится больше, чем вызов анонимной горютины `time.Sleep(5 * time.Second)`. Однако без обязательного вызова `w.Done()` анонимная горютина не может завершиться, поэтому `time.After(t)` закончится раньше и функция `timeout()` первого оператора `if` вернет `true`. Во втором операторе `if` анонимной функции не приходится ждать, поэтому функция `timeout()` вернет `false`, так как `time.Sleep(5 * time.Second)` завершится раньше, чем `time.After(t)`.

```
$ go run timeOut2.go 100
Timeout period is 100ms
Timed out!
Timed out!
```

Однако при втором запуске программы период ожидания слишком мал, поэтому в обоих случаях вызова `timeout()` у функции не будет достаточно времени для завершения работы и она будет принудительно прервана. Поэтому, выбирая интервал ожидания, убедитесь, что это подходящее значение, иначе ваши результаты могут не соответствовать ожиданиям.

И снова о Go-каналах

Как только задействуется ключевое слово `select`, появляется несколько уникальных способов использования Go-каналов, позволяющих сделать гораздо больше, чем то, что вы видели в главе 9. В этом разделе вы узнаете о разных способах использования Go-каналов.

Напомню, что нулевым значением для каналов является `nil`, и если отправить сообщение в закрытый канал, то программа перейдет в режим паники. Но если вы попытаетесь прочитать данные из закрытого канала, то получите нулевое значение для данного типа канала. Таким образом, после закрытия канала вы больше не можете записывать в него данные, но все равно можете читать его.

Чтобы канал можно было закрыть, он не должен быть предназначен только для приема данных. Кроме того, нулевой канал всегда блокируется, то есть попытка чтения или записи с нулевого канала заблокирует канал. Это свойство каналов очень полезно в тех случаях, когда нужно отключить ветвь оператора `select`, — для этого достаточно присвоить переменной канала значение `nil`.

Наконец, при попытке закрыть нулевой канал программа поднимет панику. Рассмотрим это на примере программы `closeNilChannel.go`:

```
package main

func main() {
    var c chan string
    close(c)
}
```

Выполнение `closeNilChannel.go` приведет к следующему результату:

```
$ go run closeNilChannel.go
panic: close of nil channel
goroutine 1 [running]:
main.main()
    /Users/mtsouk/closeNilChannel.go:5 +0x2a
exit status 2
```

Сигнальные каналы

Сигнальный канал — это канал, который применяется только для передачи сигналов. Проще говоря, сигнальный канал можно использовать в тех случаях, когда вы хотите проинформировать другую программу о чем-либо. К сигнальным каналам не нужно прибегать для передачи данных.



Не следует путать сигнальные каналы с обработкой сигналов UNIX, о которой говорилось в главе 8, это совершенно разные вещи.

Пример кода, в котором используются сигнальные каналы, мы рассмотрим далее в этой главе.

Буферизованные каналы

Тема этого подраздела — буферизованные каналы. Это каналы, которые позволяют планировщику Go быстро размещать задания в очереди, чтобы обрабатывать больше запросов. Кроме того, их можно использовать в качестве *семафоров*, позволяющих ограничить пропускную способность приложения.

Представленный здесь метод работает так: все входящие запросы перенаправляются на канал, который обрабатывает их по очереди. Когда канал завершает обработку запроса, он отправляет исходному вызывающему объекту сообщение о том, что канал готов обработать новый запрос. Таким образом, емкость буфера канала ограничивает количество одновременных запросов, которые этот канал может хранить.

Мы рассмотрим этот метод на примере кода программы `bufChannel.go`. Разделим его на четыре части.

Первая часть кода `bufChannel.go` выглядит так:

```
package main

import (
    "fmt"
)
```

Второй фрагмент файла `bufChannel.go` содержит следующий код Go:

```
func main() {
    numbers := make(chan int, 5)
    counter := 10
```

Представленное здесь определение `numbers` позволяет хранить в этом канале до пяти целых чисел.

В третьей части `bufChannel.go` содержится следующий код Go:

```
for i := 0; i < counter; i++ {
    select {
        case numbers <- i:
        default:
            fmt.Println("Not enough space for", i)
    }
}
```

В этом коде мы попытались поместить в канал `numbers` десять чисел. Однако, поскольку в `numbers` есть место только для пяти целых чисел, сохранить в нем все десять целых чисел нам не удастся.

Остальной код Go из файла `bufChannel.go` выглядит так:

```
for i := 0; i < counter+5; i++ {
    select {
        case num := <-numbers:
            fmt.Println(num)
        default:
            fmt.Println("Nothing more to be done!")
            break
    }
}
```

В этом коде Go мы попытались прочитать содержимое канала `numbers` с помощью цикла `for` и оператора `select`. Пока в канале `numbers` есть что читать, будет выполняться первая ветвь оператора `select`. Когда канал `numbers` пустой, выполняется ветвь `default`.

Выполнение `bufChannel.go` приведет к результату следующего вида:

```
$ go run bufChannel.go
Not enough space for 5
Not enough space for 6
Not enough space for 7
Not enough space for 8
Not enough space for 9
0
1
2
3
4
```

```
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
```

Нулевые каналы

В этом разделе вы познакомитесь с нулевыми каналами. Это особый вид каналов, который всегда заблокирован. Мы рассмотрим эти каналы на примере программы `nilChannel.go`. Разделим ее на четыре фрагмента кода.

Первая часть `nilChannel.go` выглядит так:

```
package main
```

```
import (
    "fmt"
    "math/rand"
    "time"
)
```

Во второй части `nilChannel.go` содержится следующий код Go:

```
func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)

    for {
        select {
        case input := <-c:
            sum = sum + input
        case <-t.C:
            c = nil
            fmt.Println(sum)
        }
    }
}
```

Здесь на примере функции `add()` показано, как используется нулевой канал. Оператор `<-t.C` блокирует канал `c` таймера `t` на время, указанное в вызове `time.NewTimer()`. Не путайте канал `c`, который является аргументом функции, с каналом `t.C`, который принадлежит таймеру `t`. По истечении времени таймер отправляет значение в канал `t.C`, чем инициирует выполнение соответствующей ветви оператора

`select` — он присваивает каналу `c` значение `nil` и выводит на экран значение переменной `sum`.

Третий фрагмент кода `nilChannel.go` выглядит так:

```
func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
}
```

Цель функции `send()` — генерировать случайные числа и отправлять их в канал до тех пор, пока канал открыт.

Остальной код Go в `nilChannel.go` выглядит так:

```
func main() {
    c := make(chan int)
    go add(c)
    go send(c)

    time.Sleep(3 * time.Second)
}
```

Функция `time.Sleep()` нужна для того, чтобы у двух горутин было достаточно времени для выполнения.

Запуск `nilChannel.go` приведет к следующим результатам:

```
$ go run nilChannel.go
13167523
$ go run nilChannel.go
12988362
```

Поскольку количество выполнений первой ветви оператора `select` в функции `add()` не является фиксированным, запустив `nilChannel.go` несколько раз, мы получим разные результаты.

Каналы каналов

Канал каналов — это особая разновидность переменной канала, которая вместо обычных типов переменных работает с другими каналами. Тем не менее для канала каналов все равно нужно объявить тип данных. Для того чтобы определить канал каналов, нужно использовать ключевое слово `chan` два раза подряд, как показано в следующем выражении:

```
c1 := make(chan chan int)
```



Другие типы каналов, представленные в этой главе, более популярны и полезны, чем каналы каналов.

Мы рассмотрим использование каналов каналов на примере кода, который находится в файле `chSquare.go`. Разделим его на четыре части.

Первая часть `chSquare.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "time"
)

var times int
```

Вторая часть `chSquare.go` содержит следующий код Go:

```
func f1(cc chan chan int, f chan bool) {
    c := make(chan int)
    cc <- c
    defer close(c)

    sum := 0
    select {
    case x := <-c:
        for i := 0; i <= x; i++ {
            sum = sum + i
        }
        c <- sum
    case <-f:
        return
    }
}
```

Объявив обычный канал типа `int`, передаем его переменной канала каналов. Затем с помощью оператора `select` мы получаем возможность читать данные из обычного канала типа `int` или выходить из функции, используя сигнальный канал `f`.

Прочитав одно значение из канала `c`, мы запускаем цикл `for`, вычисляющий сумму всех целых чисел от 0 до целочисленного значения, которое мы только что прочитали. Затем отправляем вычисленное значение в канал `c` типа `int` и на этом завершаем работу.

В третьей части `chSquare.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need just one integer argument!")
        return
    }
}
```

```

times, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println(err)
    return
}

cc := make(chan chan int)

```

В последней строке этого фрагмента кода мы объявляем переменную канала каналов с именем `cc`. Эта переменная является звездой данной программы, потому что именно от нее здесь все зависит. Переменная `cc` передается в функцию `f1()` и используется в следующем цикле `for`.

Оставшийся код Go программы `chSquare.go` выглядит так:

```

for i := 1; i < times+1; i++ {
    f := make(chan bool)
    go f1(cc, f)
    ch := <-cc
    ch <- i
    for sum := range ch {
        fmt.Print("Sum(", i, ")=", sum)
    }
    fmt.Println()
    time.Sleep(time.Second)
    close(f)
}
}

```

Канал `f` является сигнальным каналом для завершения горутины, когда вся работа закончена. Инструкция `ch := <-cc` позволяет получить из переменной канала каналов обычный канал, чтобы передать туда значение типа `int` с помощью оператора `ch <- i`. После этого мы читаем данные из канала, используя цикл `for`. Функция `f1()` запрограммирована на возврат одного значения, однако мы можем прочесть и несколько значений. Обратите внимание, что каждое значение `i` обслуживается своей горутинной.

Тип сигнального канала может быть любым, включая `bool`, примененный в предыдущем коде, и `struct{}`, который будет использоваться для сигнального канала в следующем разделе. Главным преимуществом сигнального канала типа `struct{}` является то, что в такой канал нельзя отправлять данные, это предотвращает возникновение ошибок.

Выполнение `chSquare.go` приведет к результатам такого вида:

```

$ go run chSquare.go 4
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10

```

```
$ go run chSquare.go 7
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
Sum(5)=15
Sum(6)=21
Sum(7)=28
```

Выбор последовательности исполнения горутин

Вы не обязаны делать какие-либо предположения относительно последовательности выполнения горутин. Однако бывают случаи, когда необходимо контролировать этот порядок. В данном подразделе вы узнаете, как это делать с помощью сигнальных каналов.



Вы спросите: «Зачем создавать горутин, а затем выполнять их в заданном порядке, если то же самое гораздо легче сделать с помощью обычных функций?» Ответ прост: горутин способны работать одновременно и ожидать завершения других горутин, тогда как функции не могут этого делать, поскольку выполняются последовательно.

В этом подразделе мы рассмотрим программу Go, которая называется `defineOrder.go`. Разделим ее на пять частей. Первая часть `defineOrder.go` выглядит так:

```
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}
```

Функция `A()` заблокирована каналом, который хранится в параметре `a`. Как только этот канал будет разблокирован в `main()`, функция `A()` начнет работать. В конце она закроет канал `b`, тем самым разблокировав другую функцию — в данном случае `B()`.

Вторая часть `defineOrder.go` содержит следующий код Go:

```
func B(a, b chan struct{}) {
    <-a
    fmt.Println("B()!")
    close(b)
}
```

Логика функции `B()` такая же, как и у `A()`. Эта функция блокируется, пока не будет закрыт канал `a`. Затем она выполняет свою работу и закрывает канал `b`. Обратите внимание, что каналы `a` и `b` ссылаются на имена параметров функции.

Третий фрагмент кода `defineOrder.go` выглядит так:

```
func C(a chan struct{}) {
    <-a
    fmt.Println("C()!")
}
```

Функция `C()` заблокирована и ожидает закрытия канала `a`, чтобы начать работу.

В четвертой части `defineOrder.go` содержится следующий код:

```
func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})
```

Эти три канала станут параметрами трех функций.

В последнем фрагменте `defineOrder.go` содержится следующий код Go:

```
go C(z)
go A(x, y)
go C(z)
go B(y, z)
go C(z)

close(x)
time.Sleep(3 * time.Second)
}
```

Здесь программа выполняет все нужные функции, а потом закрывает канал `x` и засыпает на три секунды.

Выполнение `defineOrder.go` приведет к желаемому результату, несмотря на то что функция `C()` будет вызвана несколько раз:

```
$ go run defineOrder.go
A()!
B()!
C()!
C()!
C()!
```

Многократный вызов функции `C()` как горутины не вызовет проблем, потому что `C()` не закрывает никаких каналов. Но если вызвать `A()` или `B()` более одного раза, то, скорее всего, выведется сообщение об ошибке, например такое:

```
$ go run defineOrder.go
A()!
A()!
B()!
C()!
C()!
C()!
panic: close of closed channel
goroutine 7 [running]:
main.A(0xc420072060, 0xc4200720c0)
/Users/mtsouk/Desktop/defineOrder.go:12 +0x9d
created by main.main
/Users/mtsouk/Desktop/defineOrder.go:33 +0xfa
exit status 2
```

Как видим, здесь функция `A()` была вызвана дважды. Однако, когда `A()` закрывает канал, одна из ее горутин обнаруживает, что канал уже закрыт, и создает ситуацию паники, когда пытается закрыть этот канал снова. Если мы попытаемся более одного раза вызвать функцию `B()`, то получим похожую ситуацию паники.

Как не надо использовать горутины

В этом разделе вы познакомитесь с наивным способом сортировки натуральных чисел с помощью горутин. Программа, которую мы рассмотрим, называется `sillySort.go`. Разделим ее на две части. Первая часть `sillySort.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

func main() {
    arguments := os.Args

    if len(arguments) == 1 {
        fmt.Println(os.Args[0], "n1, n2, [n]")
        return
    }
```

```
var wg sync.WaitGroup
for _, arg := range arguments[1:] {
    n, err := strconv.Atoi(arg)
    if err != nil || n < 0 {
        fmt.Print(". ")
        continue
    }
}
```

Вторая часть `sillySort.go` содержит следующий код Go:

```
wg.Add(1)
go func(n int) {
    defer wg.Done()
    time.Sleep(time.Duration(n) * time.Second)
    fmt.Print(n, " ")
}(n)
}

wg.Wait()
fmt.Println()
}
```

Сортировка выполняется посредством вызова функции `time.Sleep()` — чем больше натуральное число, тем больше проходит времени перед выполнением оператора `fmt.Print()`!

Выполнение `sillySort.go` приведет к результатам такого вида:

```
$ go run sillySort.go a -1 1 2 3 5 0 100 20 60
. . 0 1 2 3 5 20 60 100
$ go run sillySort.go a -1 1 2 3 5 0 100 -1 a 20 hello 60
. . . . 0 1 2 3 5 20 60 100
$ go run sillySort.go 0 0 10 2 30 3 4 30
0 0 2 3 4 10 30 30
```

Общая память и общие переменные

Общая память и общие переменные — наиболее распространенные способы взаимодействия потоков UNIX между собой.

Переменные *мьютекса* (сокращенное *mutual exclusion* — «взаимное исключение») используются главным образом для синхронизации потоков и для защиты общих данных, когда нужно исключить возможность одновременного выполнения нескольких операций записи. Мьютекс работает как буферизованный канал с емкостью, равной единице, так что в любой момент только одна горутина имеет доступ к общей переменной. Это означает, что две и более горутин не могут изменить эту переменную одновременно.

Код, который не может выполняться одновременно всеми процессами, потоками или в данном случае горутинами, называется *критическим разделом* конкурентной программы. Такой код должен быть защищен мьютексами. Таким образом,

определение критических разделов кода упростит весь процесс программирования, поэтому следует уделить особое внимание данной задаче.



Критический раздел не может быть встроен в другой критический раздел, если оба критических раздела используют одну и ту же переменную `sync.Mutex` или `sync.RWMutex`. Проще говоря, избегайте распределения мьютексов между функциями, так как это мешает понять, внедряете вы один критический раздел в другой или нет.

В следующих двух подразделах продемонстрировано использование типов `sync.Mutex` и `sync.RWMutex`.

Тип `sync.Mutex`

Тип `sync.Mutex` — это реализация мьютекса в Go. Определение этого типа находится в файле `mutex.go` каталога `sync` и выглядит так:

```
// Мьютекс (mutex) – сокращение mutual exclusion lock, предохранитель
// от взаимных блокировок.
// Нулевое значение для мьютекса означает открытый мьютекс.
// Однажды использованный мьютекс нельзя копировать.
type Mutex struct {
    state int32
    sema uint32
}
```



Если вам интересно, что делает этот код в стандартной библиотеке, помните, что Go является полностью открытым исходным кодом, так что вы можете спокойно открыть и прочитать его.

В определении типа `sync.Mutex` нет ничего необычного. Основная работа выполняется функциями `sync.Lock()` и `sync.Unlock()`, первая из которых блокирует, а вторая разблокирует мьютекс `sync.Mutex`. Когда мьютекс заблокирован, это означает, что никто другой не может заблокировать этот мьютекс, пока он не будет освобожден с помощью функции `sync.Unlock()`.

Мы рассмотрим использование типа `sync.Mutex` на примере программы `mutex.go`. Разделим ее на пять частей.

Первый фрагмент кода `mutex.go` выглядит так:

```
package main
```

```
import (
    "fmt"
```

```

    "os"
    "strconv"
    "sync"
    "time"
)

var (
    m sync.Mutex
    v1 int
)

```

Во второй части `mutex.go` содержится следующий код Go:

```

func change(i int) {
    m.Lock()
    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1%10 == 0 {
        v1 = v1 - 10*i
    }
    m.Unlock()
}

```

Важнейшей частью этой функции является код Go между операторами `m.Lock()` и `m.Unlock()`.

Третья часть `mutex.go`:

```

func read() int {
    m.Lock()
    a := v1
    m.Unlock()
    return a
}

```

Здесь, как и в предыдущем фрагменте, главный раздел функции находится между операторами `m.Lock()` и `m.Unlock()`.

Четвертый фрагмент кода `mutex.go` выглядит так:

```

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }

    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var waitGroup sync.WaitGroup

```


Последняя часть `mutex.go` содержит следующий код Go:

```
fmt.Printf("%d ", read())
for i := 0; i < numGR; i++ {
    waitGroup.Add(1)
    go func(i int) {
        defer waitGroup.Done()
        change(i)
        fmt.Printf("-> %d", read())
    }(i)
}

waitGroup.Wait()
fmt.Printf("-> %d\n", read())
}
```

Выполнение `mutex.go` приведет к результатам следующего вида:

```
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -30-> -29-> -28-> -27-> -26->
-25-> -24-> -23-> -22-> -21-> -210-> -209-> -209
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -130-> -129-> -128-> -127-> -126->
-125-> -124-> -123-> -122-> -121-> -220-> -219-> -219
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -100-> -99-> -98-> -97-> -96->
-95-> -94-> -93-> -92-> -91-> -260-> -259-> -259
```

Если удалить операторы `m.Lock()` и `m.Unlock()` из функции `change()`, то программа выдаст результат, подобный следующему:

```
$ go run mutex.go 21
0 -> 1-> 6-> 7-> 5-> -60-> -59-> 9-> 2-> -58-> 3-> -52-> 4-> -57-> 8->
-55-> -90-> -54-> -89-> -53-> -56-> -51-> -89
$ go run mutex.go 21
0 -> 1-> 7-> 8-> 9-> 5-> -99-> 4-> 2-> -97-> -96-> 3-> -98-> -95-> -100->
-93-> -94-> -92-> -91-> -230-> 6-> -229-> -229
$ go run mutex.go 21
0 -> 3-> 7-> 8-> 9-> -120-> -119-> -118-> -117-> 1-> -115-> -114-> -116->
4-> 6-> -112-> 2-> -111-> 5-> -260-> -113-> -259-> -259
```

Причиной такого изменения результатов программы является то, что все процедуры изменяют общую переменную одновременно, из-за чего результаты выводятся в случайном порядке.

Что произойдет, если вы забудете разблокировать мьютекс

В этом пункте вы узнаете, что произойдет, если забыть разблокировать `sync.Mutex`. Для этого мы рассмотрим пример кода Go из файла `forgetMutex.go`. Разделим его на две части.

Первая часть `forgetMutex.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "sync"
)

var m sync.Mutex

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```

Причиной всех проблем в этой программе является то, что разработчик забыл снять блокировку с `sync.Mutex`. Однако если программа вызывает `function()` только один раз, то все будет отлично.

Вторая часть `forgetMutex.go` выглядит так:

```
func main() {
    var w sync.WaitGroup

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    w.Wait()
}
```

В функции `main()`, которая генерирует только две горутины и ожидает их завершения, нет ничего плохого.

Выполнение `forgetMutex.go` приведет к следующим результатам:

```
$ go run forgetMutex.go
Locked!
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001209c)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420012090)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/waitgroup.go:131 +0x72
```

```

main.main()
/Users/mtsouk/forgetMutex.go:30 +0xb6
goroutine 5 [semacquire]:
sync.runtime_SemacquireMutex(0x115c6fc, 0x0)
/usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:71 +0x3d
sync.(*Mutex).Lock(0x115c6f8)
/usr/local/Cellar/go/1.12.3/libexec/src/sync/mutex.go:134 +0xee
main.function()
/Users/mtsouk/forgetMutex.go:11 +0x2d
main.main.func1(0xc420012090)
/Users/mtsouk/forgetMutex.go:20 +0x48
created by main.main
/Users/mtsouk/forgetMutex.go:18 +0x58
exit status 2

```

Таким образом, если вы забудете разблокировать мьютекс `sync.Mutex`, то создадите ситуацию паники даже в самой простой программе. То же самое относится к мьютексам типа `sync.RWMutex`. Они рассмотрены в следующем подразделе.

Тип `sync.RWMutex`

Тип `sync.RWMutex` — это еще один вид мьютекса. Фактически это улучшенная версия `sync.Mutex`, определение которой находится в файле `rwmutex.go` каталога `sync`:

```

type RWMutex struct {
    w      Mutex // held if there are pending writers
    writerSem uint32 // semaphore for writers to wait for completing readers
    readerSem uint32 // semaphore for readers to wait for completing writers
    readerCount int32 // number of pending readers
    readerWait int32 // number of departing readers
}

```

Другими словами, тип `sync.RWMutex` основан на `sync.Mutex` с необходимыми дополнениями и улучшениями.

Теперь поговорим о том, что в `sync.RWMutex` улучшено по сравнению с `sync.Mutex`. Несмотря на то что только одной функции разрешается выполнять операции записи с мьютексом `sync.RWMutex`, у нас может быть несколько функций, владеющих мьютексом `sync.RWMutex` и выполняющих чтение. Однако следует учитывать, что, пока все функции, читающие мьютекс типа `sync.RWMutex`, не разблокируют этот мьютекс, вы не сможете заблокировать его для записи — не особенно большая цена за возможность разрешить несколько операций чтения.

Работать с мьютексом `sync.RWMutex` вам помогут функции `RLock()` и `RUnlock()`, которые блокируют и разблокируют мьютекс для чтения соответственно. Функции `Lock()` и `Unlock()`, используемые в мьютексе `sync.Mutex`, по-прежнему при-

меняются для того, чтобы заблокировать и разблокировать мьютекс `sync.RWMutex` для записи. Таким образом, вызов функции `RLock()`, которая блокирует мьютекс для чтения, должен быть связан с вызовом функции `Runlock()`. Наконец, как вы, очевидно, понимаете, не следует изменять какие-либо общие переменные внутри блока кода, расположенного между `RLock()` и `Runlock()`.

Мы рассмотрим применение и полезные свойства типа `sync.RWMutex` на примере кода Go, который находится в файле `rwMutex.go`. Эта программа содержит две отличающиеся версии одной и той же функции. Рассмотрим ее, разделив на шесть частей. В первой из этих функций для чтения данных используется мьютекс типа `sync.RWMutex`, а второй — мьютекс типа `sync.Mutex`. Разница в производительности между этими двумя функциями поможет вам лучше понять преимущества использования мьютекса `sync.RWMutex` для чтения.

Первая часть `rwMutex.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "sync"
    "time"
)

var Password = secret{password: "myPassword"}

type secret struct {
    RWM    sync.RWMutex
    M      sync.Mutex
    password string
}
```

Структура `secret` содержит общую переменную, мьютекс `sync.RWMutex` и мьютекс `sync.Mutex`.

Во второй части файла `rwMutex.go` содержится следующий код:

```
func Change(c *secret, pass string) {
    c.RWM.Lock()
    fmt.Println("LChange")
    time.Sleep(10 * time.Second)
    c.password = pass
    c.RWM.Unlock()
}
```

Функция `Change()` изменяет общую переменную. Следовательно, в ней нужно использовать монопольную блокировку, для чего применяется функции `Lock()` и `Unlock()`. Если мы изменяем переменную, то монопольной блокировки не избежать!

Третья часть `rwMutex.go` выглядит так:

```
func show(c *secret) string {
    c.RWM.RLock()
    fmt.Print("show")
    time.Sleep(3 * time.Second)
    defer c.RWM.RUnlock()
    return c.password
}
```

В функции `show()` используются `RLock()` и `RUnlock()`, поскольку ее критический раздел нужен для чтения общей переменной. Таким образом, несмотря на то, что несколько горутин могут читать общую переменную, эту переменную нельзя изменить без использования функций `Lock()` и `Unlock()`. Однако пока какая-либо из горутин читает общую переменную с помощью мьютекса, `Lock()` будет заблокирована.

В четвертом фрагменте файла `rwMutex.go` содержится следующий код Go:

```
func showWithLock(c *secret) string {
    c.RWM.Lock()
    fmt.Println("showWithLock")
    time.Sleep(3 * time.Second)
    defer c.RWM.Unlock()
    return c.password
}
```

Единственное различие между кодом функций `showWithLock()` и `show()` состоит в том, что в `showWithLock()` применяется монопольная блокировка для чтения данных. Следовательно, только одна функция `showWithLock()` может прочитать поле `password` структуры `secret`.

В пятой части `rwMutex.go` содержится следующий код Go:

```
func main() {
    var showFunction = func(c *secret) string { return "" }
    if len(os.Args) != 2 {
        fmt.Println("Using sync.RWMutex!")
        showFunction = show
    } else {
        fmt.Println("Using sync.Mutex!")
        showFunction = showWithLock
    }

    var waitGroup sync.WaitGroup

    fmt.Println("Pass:", showFunction(&Password))
```

Оставшийся код `rwMutex.go` выглядит так:

```
for i := 0; i < 15; i++ {
    waitGroup.Add(1)
```

```

    go func() {
        defer waitGroup.Done()
        fmt.Println("Go Pass:", showFunction(&Password))
    }()

    go func() {
        waitGroup.Add(1)
        defer waitGroup.Done()
        Change(&Password, "123456")
    }()

    waitGroup.Wait()
    fmt.Println("Pass:", showFunction(&Password))
}

```

Если дважды выполнить `rwMutex.go` и воспользоваться утилитой командной строки `time(1)` для сравнения двух версий программы, то получим результат такого вида:

```

$ time go run rwMutex.go 10 >/dev/null
real    0m51.206s
user    0m0.130s
sys     0m0.074s
$ time go run rwMutex.go >/dev/null
real    0m22.191s
user    0m0.135s
sys     0m0.071s

```

Обратите внимание, что `> /dev/null` в конце обеих команд нужен для того, чтобы пропустить вывод. Следовательно, версия, в которой применяется мьютекс `sync.RWMutex`, работает гораздо быстрее, чем версия с `sync.Mutex`.

Пакет `atomic`

Атомарная операция — это операция, которая выполняется за один шаг относительно других потоков или в данном случае других горутин. Таким образом, атомарная операция не может быть прервана на середине.

В стандартную библиотеку Go входит пакет `atomic`, который иногда позволяет обойтись без использования мьютекса. Однако мьютексы более универсальны, чем атомарные операции. С помощью пакета `atomic` можно создавать атомные счетчики, доступные для нескольких горутин, не рискуя получить проблемы синхронизации и состояние гонки.

Обратите внимание, что в случае атомарной переменной все операции чтения и записи должны выполняться с применением атомарных функций, предоставляемых пакетом `atomic`. Мы рассмотрим использование пакета `atomic` на примере программы `atom.go`. Разделим ее на три части.

Первая часть `atom.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "sync"
    "sync/atomic"
)

type atomCounter struct {
    val int64
}

func (c *atomCounter) Value() int64 {
    return atomic.LoadInt64(&c.val)
}
```

Вторая часть `atom.go` содержит следующий код:

```
func main() {
    minusX := flag.Int("x", 100, "Goroutines")
    minusY := flag.Int("y", 200, "Value")
    flag.Parse()
    X := *minusX
    Y := *minusY

    var waitGroup sync.WaitGroup
    counter := atomCounter{}
```

Последняя часть `atom.go`:

```
    for i := 0; i < X; i++ {
        waitGroup.Add(1)
        go func(no int) {
            defer waitGroup.Done()
            for i := 0; i < Y; i++ {
                atomic.AddInt64(&counter.val, 1)
            }
        }(i)
    }

    waitGroup.Wait()
    fmt.Println(counter.Value())
}
```

Требуемая переменная изменяется с помощью функции `atomic.AddInt64()`.

Выполнение `atom.go` приведет к следующим результатам:

```
$ go run atom.go
20000
$ go run atom.go -x 4000 -y 10
40000
```

Результаты `atom.go` подтверждают, что счетчик, применяемый в программе, безопасен. Интересно попробовать изменить программу `atom.go` так, чтобы изменить переменную `counter`, используя вместо `atomic.AddInt64()` обычную арифметику (`counter.val++`). В этом случае результаты программы были бы похожи на следующие:

```
$ go run atom.go -x 4000 -y 10
37613
$ go run atom.go
15247
```

Как видно из результатов, из-за `counter.val++` поток программы становится небезопасным.

В главе 12 мы рассмотрим похожий пример, в котором задействован написанный на Go HTTP-сервер.

Совместное использование памяти с помощью горутин

В последнем подразделе этого раздела показано, как можно организовать совместный доступ к данным с помощью выделенной для этого горутин. Несмотря на то что общая память является традиционным способом взаимодействия потоков между собой, в стандартный комплект поставки Go входят встроенные функции синхронизации, которые позволяют одной горутине владеть общей частью данных. Это означает, что другие горутин должны отправлять сообщения этой горути, которая владеет общими данными, что предотвращает повреждение данных. Такая горутина называется *управляющей горутин*. Согласно терминологии Go, это **не коммуникация посредством общей памяти, а общая память посредством коммуникации**.

Рассмотрим эту методику более подробно на примере программы `monitor.go`. Разделим ее на пять частей. Программа `monitor.go` генерирует случайные числа посредством управляющей горутин.

Первая часть `monitor.go` выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "sync"
    "time"
)

var readValue = make(chan int)
var writeValue = make(chan int)
```


Канал `readValue` используется для чтения случайных чисел, а канал `writeValue` — для получения новых случайных чисел.

Вторая часть `monitor.go` содержит следующий код:

```
func set(newValue int) {
    writeValue <- newValue
}

func read() int {
    return <-readValue
}
```

Назначение функции `set()` — установить значение общей переменной, тогда как `read()` предназначена для чтения значения сохраненной переменной.

Третий фрагмент кода программы `monitor.go` выглядит так:

```
func monitor() {
    var value int
    for {
        select {
            case newValue := <-writeValue:
                value = newValue
                fmt.Printf("%d ", value)
            case readValue <- value:
        }
    }
}
```

Вся логика программы заключена в реализации функции `monitor()`, а точнее, в операторе `select`, который управляет работой всей программы.

Когда поступает запрос на чтение, функция `read()` пытается выполнить операцию чтения из канала `readValue`, который управляется функцией `monitor()`. Результатом операции является текущее значение, которое хранится в переменной `value`. И наоборот, когда мы хотим изменить сохраненное значение, то вызываем функцию `set()`. Она записывает данные в канал `writeValue`, который также обрабатывается оператором `select`. В результате никто не может обратиться к общей переменной в обход функции `monitor()`.

Четвертый фрагмент кода `monitor.go` выглядит так:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give an integer!")
        return
    }
    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

```
fmt.Printf("Going to create %d random numbers.\n", n)
rand.Seed(time.Now().Unix())
go monitor()
```

Последняя часть `monitor.go` содержит следующий код Go:

```
var w sync.WaitGroup

for r := 0; r < n; r++ {
    w.Add(1)
    go func() {
        defer w.Done()
        set(rand.Intn(10 * n))
    }()
}
w.Wait()
fmt.Printf("\nLast value: %d\n", read())
}
```

Выполнение `monitor.go` приведет к такому результату:

```
$ go run monitor.go 20
Going to create 20 random numbers.
89 88 166 42 149 89 20 84 44 178 184 28 52 121 62 91 31 117 140 106
Last value: 106
$ go run monitor.go 10
Going to create 10 random numbers.
30 16 66 70 65 45 31 57 62 26
Last value: 26
```



Я предпочитаю использовать управляющую горутину вместо традиционных методов общего доступа к памяти, потому что реализация с использованием управляющей горуты безопаснее, ближе к философии Go и намного чище.

И снова об операторе go

Горутины выполняются быстро, и на одной машине могут выполняться тысячи горутин, однако это имеет свою цену. В данном разделе мы поговорим об операторе `go`, его поведении и о том, что происходит, когда мы запускаем очередную горутину в программе Go.

Обратите внимание, что замкнутые переменные (closed variables) в горутинках определяются в момент фактического выполнения горутинки, когда выполняется оператор `go`, создающий новую горутину. Это означает, что замкнутые переменные заменяются их значениями в тот момент, когда планировщик Go решит

выполнить соответствующий код. Эта особенность продемонстрирована в следующем коде Go, который хранится в файле `cloGo.go`:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i <= 20; i++ {
        go func() {
            fmt.Print(i, " ")
        }()
    }
    time.Sleep(time.Second)
    fmt.Println()
}
```

Было бы неплохо в этом месте прервать чтение и попробовать угадать, каким будет результат этого кода.

Многократное выполнение `cloGo.go` выявляет проблему, о которой мы говорили:

```
$ go run cloGo.go
9 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21
$ go run cloGo.go
4 21 21 21 21 21 21 21 21 6 21 21 21 21 21 21 21 21
$ go run cloGo.go
6 21 6 6 21 21 21 21 21 21 21 21 21 21 21 21 21 6 21
```

Программа по большей части выводит число 21, которое является последним значением переменной цикла `for`, и изредка — другие числа. Дело в том, что `i` — *замкнутая переменная* и поэтому определяется во время выполнения программы. Горутины создаются, но не запускаются, ожидая, пока их запустит планировщик Go. Тем временем цикл `for` заканчивается, поэтому значение `i` равно 21. Кстати, эта же проблема относится и к каналам Go — будьте осторожны.

Существует довольно забавный и неожиданный способ решить эту проблему, в котором используется идиоматическая особенность Go. Это решение показано в программе `cloGoCorrect.go`. Она содержит следующий код:

```
package main

import (
    "fmt"
    "time"
)
```

```
func main() {
    for i := 0; i <= 20; i++ {
        i := i
        go func() {
            fmt.Print(i, " ")
        }()
        time.Sleep(time.Second)
        fmt.Println()
    }
}
```

Корректный, но странный оператор `i := i` создает новый экземпляр переменной для горутины, благодаря чему результат работы `cloGoCorrect.go` выглядит примерно так:

```
$ go run cloGoCorrect.go
1 5 4 3 6 0 13 7 8 9 10 11 12 17 14 15 16 19 18 20 2
$ go run cloGoCorrect.go
5 2 20 13 6 7 1 9 10 11 0 3 17 14 15 16 4 19 18 8 12
```

Рассмотрим еще один необычный случай использования оператора `go`. Посмотрите на следующий код Go, который хранится в файле `endlessComp.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    var i byte
    go func() {
        for i = 0; i <= 255; i++ {
        }
    }()
    fmt.Println("Leaving goroutine!")
    runtime.Gosched()
    runtime.GC()

    fmt.Println("Good bye!")
}
```

Результат выполнения `endlessComp.go` вас удивит: эта программа никогда не заканчивается, потому что блокируется на неопределенный срок. Ее приходится останавливать вручную:

```
$ go run endlessComp.go
Leaving goroutine!
^Csignal: interrupt
```

Как вы уже догадались, главная причина проблемы связана со сборщиком мусора Go и принципом его работы. При вызове функции `runtime.Gosched()` планировщику дается задание выполнить другую горутину, после чего мы вызываем сборщик мусора Go, который пытается выполнить свою работу.

Чтобы сборщик мусора смог выполнить свою работу и заснуть, он должен перебрать все горутин. Однако проблема в том, что цикл `for` не закончится никогда, потому что переменная `for` имеет тип `byte`. Это означает, что `for` не позволяет системе делать что-либо еще, так как горутин, в которой находится цикл `for`, не заснет никогда. Этого неприятного обстоятельства не избежать, даже если у вашей машины несколько ядер.

Обратите внимание: если бы `for` не был пустым, то программа была бы выполнена и успешно завершилась, потому что сборщик мусора мог бы остановиться¹.

И наконец, имейте в виду, что для того чтобы горутин завершилась, нужен четкий сигнал о ее завершении. Проще всего завершить горутин с помощью обычного оператора `return`.

Распознавание состояния гонки

Состояние гонки по данным — это ситуация, когда два и более работающих элемента, таких как потоки и горутин, пытаются получить контроль над общим ресурсом или переменной программы либо изменить их. Строго говоря, гонка по данным происходит тогда, когда две или более инструкции обращаются к одному и тому же адресу памяти и хотя бы одна из них выполняет операцию записи. Если все операции являются чтением, то состояния гонки нет.

Использование флага `-race` при запуске или сборке исходного файла Go включает *детектор гонки Go*, что заставляет компилятор создать модифицированную версию исполняемого файла. Эта измененная версия может записывать все операции доступа к общим переменным, а также все события синхронизации, включая обращения к `sync.Mutex` и `sync.WaitGroup`. Проанализировав соответствующие события, детектор гонки выводит отчет, который может помочь распознать потенциальные проблемы, чтобы их можно было вовремя исправить.

Рассмотрим следующий код Go, который хранится в файле `raceC.go`. Мы разделим эту программу на три части.

Первая часть `raceC.go` выглядит так:

```
package main

import (
    "fmt"
```

¹ На момент подготовки русскоязычного издания уже вышла версия Go 1.14, в которой решена проблема блокировки планировщика «пустыми» циклами.

```

    "os"
    "strconv"
    "sync"
)

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

Вторая часть `raceC.go`:

```

var waitGroup sync.WaitGroup
var i int
k := make(map[int]int)
k[1] = 12

for i = 0; i < numGR; i++ {
    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        k[i] = i
    }()
}

```

Остальной код Go в `raceC.go` выглядит следующим образом:

```

k[2] = 10
waitGroup.Wait()
fmt.Printf("k = %v\n", k)
}

```

Посчитав недостаточным то, что к отображению `k` одновременно обращаются несколько горутин, я добавил еще один оператор, который обращается к отображению `k` перед вызовом функции `sync.Wait()`.

Если запустить `raceC.go`, то получим результат следующего вида, без каких-либо предупреждений или сообщений об ошибках:

```

$ go run raceC.go 10
k = map[int]int{7:10, 2:10, 10:10, 1:12}
$ go run raceC.go 10
k = map[int]int{2:10, 10:10, 1:12, 8:8, 9:9}
$ go run raceC.go 10
k = map[int]int{10:10, 1:12, 6:7, 7:7, 2:10}

```

Если выполнить `raceC.go` только один раз, то все будет выглядеть нормально, несмотря на то что мы не получим того, что ожидали при выводе содержимого отображения `k`. Однако многократное выполнение `raceC.go` говорит о том, что здесь что-то не так, — главным образом потому, что при каждом выполнении программы мы получаем разные результаты.

Из программы `raceC.go` и ее неожиданных результатов можно извлечь еще много интересного — если мы решим использовать детектор гонки Go для ее анализа:

```
$ go run -race raceC.go 10
=====
WARNING: DATA RACE
Read at 0x00c00001a0a8 by goroutine 6:
    main.main.func1()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x66
Previous write at 0x00c00001a0a8 by main goroutine:
    main.main()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:28 +0x23f
Goroutine 6 (running) created at:
    main.main()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====
=====
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 7:
    runtime.mapassign_fast64()
        /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
    main.main.func1()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Previous write at 0x00c0000bc000 by goroutine 6:
    runtime.mapassign_fast64()
        /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
    main.main.func1()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Goroutine 7 (running) created at:
    main.main()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
Goroutine 6 (finished) created at:
    main.main()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====
=====
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 8:
    runtime.mapassign_fast64()
        /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
    main.main.func1()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Previous write at 0x00c0000bc000 by goroutine 6:
    runtime.mapassign_fast64()
```

```

/usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92
main.main.func1()
/Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Goroutine 8 (running) created at:
main.main()
/Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
Goroutine 6 (finished) created at:
main.main()
/Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====
k = map[int]int{1:1, 2:10, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:10}
Found 3 data race(s)
exit status 66

```

Итак, детектор гонки обнаружил три состояния гонки по данным. Каждое из них в его результатах начинается с сообщения **WARNING: DATA RACE**.

Первая ситуация гонки по данным происходит в функции `main.main.func1()` в строке 32, которая вызывается из строки 28 цикла `for`. Последний, в свою очередь, вызывается горутинной, созданной в строке 30. Эта проблема обозначена в сообщении **Previous write**. Изучив соответствующий код, легко понять, что реальная проблема заключается в том, что анонимная функция не принимает параметров. Следовательно, значение `i`, которое используется в цикле `for`, не может быть детерминистически определено, поскольку оно изменяется в цикле `for`, который является операцией записи.

Сообщение о втором состоянии гонки по данным гласит: **Write at 0x00c0000bc000 by goroutine 7**. Если мы прочитаем соответствующие результаты программы, то увидим, что здесь гонка по данным связана с операцией записи, и это происходит в отображении Go в строке 32 по крайней мере с двумя горутинами, которые запускаются в строке 30. Поскольку эти две горутинны имеют одинаковое имя (`main.main.func1()`), это значит, что мы говорим об одной и той же горутине. Когда две горутинны пытаются изменить одну и ту же переменную, возникает состояние гонки по данным. Третье состояние гонки по данным похоже на второе.



Запись `main.main.func1()` используется в Go для внутреннего присвоения имени анонимной функции. Если у нас разные анонимные функции, то их имена также будут разными.

Вы спросите: «Что я могу сделать уже сейчас, чтобы исправить проблемы, вызвавшие два состояния гонки по данным?»

Вы можете переписать функцию `main()` в `raceC.go` следующим образом:

```

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
    }
}

```



```

        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    var waitGroup sync.WaitGroup
    var i int

    k := make(map[int]int)
    k[1] = 12

    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(j int) {
            defer waitGroup.Done()
            aMutex.Lock()
            k[j] = j
            aMutex.Unlock()
        }(i)
    }

    waitGroup.Wait()
    k[2] = 10
    fmt.Printf("k = %#v\n", k)
}

```

Переменная `aMutex` является глобальной переменной типа `sync.Mutex`, она определена вне функции `main()` и доступна из любой точки программы. Хотя это и не обязательно, наличие такой глобальной переменной избавляет от необходимости постоянно передавать ее в функции.

Сохранив новую версию `raceC.go` под именем `noRaceC.go` и выполнив ее, получим ожидаемый результат:

```

$ go run noRaceC.go 10
k = map[int]int{1:1, 0:0, 5:5, 3:3, 6:6, 9:9, 2:10, 4:4, 7:7, 8:8}

```

Обработка `noRaceC.go` с помощью детектора гонки Go приведет к следующему:

```

$ go run -race noRaceC.go 10
k = map[int]int{5:5, 7:7, 9:9, 1:1, 0:0, 4:4, 6:6, 8:8, 2:10, 3:3}

```

Обратите внимание, что при доступе к хеш-таблице `k` нужно использовать механизм блокировки. Если этого не делать и просто изменить реализацию анонимной функции, которая выполняется как горутина, то при выполнении `go run noRaceC.go` получим следующий результат:

```

$ go run noRaceC.go 10
fatal error: concurrent map writes
goroutine 10 [running]:

```

```
runtime.throw(0x10ca0bd, 0x15)
/usr/local/Cellar/go/1.9.3/libexec/src/runtime/panic.go:605 +0x95
fp=0xc420024738 sp=0xc420024718 pc=0x10276b5
runtime.mapassign_fast64(0x10ae680, 0xc420074180, 0x5, 0x0)
/usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashmap_fast.go:607
+0x3d2 fp=0xc420024798 sp=0xc420024738 pc=0x100b582
main.main.func1(0xc420010090, 0xc420074180, 0x5)
/Users/mtsouk/ch10/code/noRaceC.go:35 +0x6b fp=0xc4200247c8
sp=0xc420024798 pc=0x1096f5b
runtime.goexit()
/usr/local/Cellar/go/1.9.3/libexec/src/runtime/asm_amd64.s:2337 +0x1
fp=0xc4200247d0 sp=0xc4200247c8 pc=0x1050c21
created by main.main
/Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001009c)
/usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420010090)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
/Users/mtsouk/ch10/code/noRaceC.go:40 +0x17a
goroutine 12 [runnable]:
sync.(*WaitGroup).Done(0xc420010090)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:99 +0x43
main.main.func1(0xc420010090, 0xc420074180, 0x7)
/Users/mtsouk/ch10/code/noRaceC.go:37 +0x79
created by main.main
/Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
exit status 2
```

Корень проблемы ясен: одновременная запись в хеш-таблицу.

Пакет context

Главное назначение пакета `context` — определение типа `Context` и поддержка *аннулирования*. Да, вы все правильно поняли: бывают случаи, когда по какой-либо причине необходимо отказаться от того, что вы делаете. Однако очень хорошо, когда есть возможность добавить дополнительную информацию о ваших решениях об аннулировании. Именно это позволяет сделать пакет `context`.

Если вы посмотрите в исходный код пакета `context`, то поймете, что его реализация довольно проста — даже реализация типа `Context` проста, тем не менее, пакет `context` очень важен.



В течение некоторого времени пакет `context` был внешним пакетом Go; впервые он стал стандартным в версии Go 1.7. Таким образом, если у вас более старая версия Go, то вы не сможете выполнить код из этого раздела, не загрузив предварительно пакет `context` или не установив более новую версию Go.

Тип `Context` — это интерфейс, имеющий четыре метода: `Deadline()`, `Done()`, `Err()` и `Value()`. К счастью, мы не обязаны реализовывать все функции интерфейса `Context` — достаточно лишь изменить переменную `Context`, используя такие функции, как `context.WithCancel()`, `context.WithDeadline()` и `context.WithTimeout()`. Эти три функции возвращают производный (дочерний) объект `Context` и функцию `CancelFunc`. Вызов функции `CancelFunc` удаляет ссылку родителя на дочерний объект и останавливает все связанные с ним таймеры. Следовательно, после этого сборщик мусора Go может свободно собрать дочерние горутин, у которых больше нет связанных с ними родительских горутин. Чтобы сборка мусора работала правильно, родительская горутина должна хранить ссылку на все дочерние горутин. Если дочерняя горутина завершится без ведома родительской, начинается утечка памяти, которая продолжается до тех пор, пока родительская горутина также не будет аннулирована.

Рассмотрим простой пример использования пакета `context` с применением кода Go из файла `simpleContext.go`. Разделим его на шесть частей.

Первый фрагмент файла `simpleContext.go` содержит следующий код:

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"
)
```

Вторая часть `simpleContext.go` выглядит так:

```
func f1(t int) {
    c1 := context.Background()
    c1, cancel := context.WithCancel(c1)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()
}
```

Функция `f1()` принимает всего один аргумент, который является временной задержкой, — все остальное определено внутри функции. Обратите внимание, что тип переменной `cancel` — `context.CancelFunc`.

Для инициализации пустого объекта `Context` нужно вызвать `context.Background()`. Функция `context.WithCancel()` использует существующий объект `Context` и создает его потомка с возможностью аннулирования. Функция `context.WithCancel()` также создает канал `Done`, который можно закрыть либо при вызове функции

`cancel()`, как показано в предыдущем коде, либо когда закрывается канал `Done` родительского контекста.

Третья часть `simpleContext.go` содержит остальной код функции `f1()`:

```
select {
case <-c1.Done():
    fmt.Println("f1():", c1.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f1():", r)
}
return
}
```

Здесь мы видим использование функции `Done()` для переменной типа `Context`. При вызове этой функции происходит аннулирование. Возвращаемое значение `Context.Done()` является каналом, потому что в противном случае мы не смогли бы применить его в операторе `select`.

Четвертая часть `simpleContext.go` содержит следующий код Go:

```
func f2(t int) {
    c2 := context.Background()
    c2, cancel := context.WithTimeout(c2, time.Duration(t)*time.Second)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
case <-c2.Done():
    fmt.Println("f2():", c2.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f2():", r)
}
return
}
```

В этом фрагменте показано применение функции `context.WithTimeout()`, которая принимает два аргумента: один типа `Context`, а второй — типа `time.Duration`. По окончании времени ожидания автоматически вызывается функция `cancel()`.

Пятая часть `simpleContext.go` выглядит так:

```
func f3(t int) {
    c3 := context.Background()
    deadline := time.Now().Add(time.Duration(2*t) * time.Second)
    c3, cancel := context.WithDeadline(c3, deadline)
    defer cancel()
}
```

```

go func() {
    time.Sleep(4 * time.Second)
    cancel()
}()

select {
case <-c3.Done():
    fmt.Println("f3():", c3.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f3():", r)
}
return
}

```

В этом коде Go показано использование функции `context.WithDeadline()`, которая принимает два аргумента: `Context` и время в будущем, соответствующее крайнему сроку окончания операции. По истечении этого времени автоматически вызывается функция `cancel()`.

Оставшийся код Go файла `simpleContext.go` выглядит так:

```

func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a delay!")
        return
    }

    delay, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Delay:", delay)

    f1(delay)
    f2(delay)
    f3(delay)
}

```

Назначение функции `main()` — все инициализировать.

Выполнение `simpleContext.go` приведет к результатам следующего вида:

```

$ go run simpleContext.go 4
Delay: 4
f1(): context canceled
f2(): 2019-04-11 18:18:43.327345 +0300 EEST m+=8.004588898
f3(): 2019-04-11 18:18:47.328073 +0300 EEST m+=12.005483099
$ go run simpleContext.go 2
Delay: 2
f1(): 2019-04-11 18:18:53.972045 +0300 EEST m+=2.005231943
f2(): context deadline exceeded

```

```
f3(): 2019-04-11 18:18:57.974337 +0300 EEST m=+6.007690061
$ go run simpleContext.go 10
Delay: 10
f1(): context canceled
f2(): context canceled
f3(): context canceled
```

Длинные строки этих выходных данных являются возвращаемыми значениями, получаемыми при вызовах функции `time.After()`. Они соответствуют нормальной работе программы. Дело в том, что работа программы аннулируется при задержках в ее выполнении.

Использование пакета `context` действительно так просто, как показано здесь, несмотря на то что представленный код не выполняет сколь-нибудь серьезной работы с интерфейсом `Context`. Однако код Go, который мы рассмотрим в следующем подразделе, представляет более реалистичный пример.

Расширенный пример использования пакета context

Гораздо лучше и глубже функциональность пакета `context` видна на примере кода Go из программы `useContext.go`. Разделим ее на пять частей. В этом примере мы создадим HTTP-клиент, который не желает слишком долго ждать ответа от HTTP-сервера, — вполне обычный сценарий. Фактически, поскольку почти все HTTP-клиенты поддерживают такую функциональность, мы рассмотрим другой способ отмены HTTP-запроса в главе 12.

Программа `useContext.go` принимает два аргумента командной строки: URL-адрес сервера, к которому она будет подключаться, и время, в течение которого представленная утилита должна ожидать отклика сервера. Если передать программе только один аргумент командной строки, то время ожидания составит пять секунд.

Первый фрагмент кода `useContext.go` выглядит так:

```
package main

import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    myUrl    string
```

```

    delay    int = 5
    w        sync.WaitGroup
)

type myData struct {
    r    *http.Response
    err   error
}

```

И `myURL`, и `delay` являются глобальными переменными, поэтому к ним можно обращаться из любой точки кода. Кроме того, здесь еще есть переменная типа `sync.WaitGroup` с именем `w`, также с глобальной областью действия, и определена структура с именем `myData` для хранения ответа веб-сервера и переменной `error` на случай, если где-то возникнет ошибка. Вторая часть `useContext.go` содержит следующий код Go:

```

func connect(c context.Context) error {
    defer w.Done()
    data := make(chan myData, 1)

    tr := &http.Transport{}
    httpClient := &http.Client{Transport: tr}

    req, _ := http.NewRequest("GET", myUrl, nil)

```

Этот код Go отвечает за HTTP-соединение.



Подробнее о разработке HTTP-серверов и клиентов вы прочитаете в главе 12.

В третьей части `useContext.go` содержится такой код Go:

```

go func() {
    response, err := httpClient.Do(req)
    if err != nil {
        fmt.Println(err)
        data <- myData{nil, err}
        return
    } else {
        pack := myData{response, err}
        data <- pack
    }
}()

```

Четвертый фрагмент `useContext.go` содержит следующий код Go:

```

select {
case <-c.Done():
    tr.CancelRequest(req)
    <-data

```

```

        fmt.Println("The request was cancelled!")
        return c.Err()
    case ok := <-data:
        err := ok.err
        resp := ok.r
        if err != nil {
            fmt.Println("Error select:", err)
            return err
        }
        defer resp.Body.Close()

        realHTTPData, err := ioutil.ReadAll(resp.Body)
        if err != nil {
            fmt.Println("Error select:", err)
            return err
        }
        fmt.Printf("Server Response: %s\n", realHTTPData)
    }
    return nil
}

```

Остальной код `useContext.go` является реализацией функции `main()` и выглядит так:

```

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need a URL and a delay!")
        return
    }

    myUrl = os.Args[1]
    if len(os.Args) == 3 {
        t, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println(err)
            return
        }
        delay = t
    }

    fmt.Println("Delay:", delay)
    c := context.Background()
    c, cancel := context.WithTimeout(c, time.Duration(delay)*time.Second)
    defer cancel()

    fmt.Printf("Connecting to %s \n", myUrl)
    w.Add(1)
    go connect(c)
    w.Wait()
    fmt.Println("Exiting...")
}

```


Период ожидания определяется методом `context.WithTimeout()`. Функция `connect()`, которая выполняется как горутина, завершится либо нормально, либо посредством выполнения `cancel()`. Обратите внимание, что принято использовать `context.Background()` в функции `main()`, или `init()` пакета, или в тестах.

Несмотря на то что вам не обязательно знать о серверной части операции, следует видеть, как Go-версия веб-сервера может замедляться случайным образом. Так, генератор случайных чисел решает, насколько медленным будет веб-сервер, — реальные веб-серверы могут оказаться слишком загруженными и поэтому не ответить. Или же могут возникнуть проблемы с сетью, которые вызовут задержку. Рассмотрим исходный файл `slowWWW.go`. Его содержимое выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func myHandler(w http.ResponseWriter, r *http.Request) {
    delay := random(0, 15)
    time.Sleep(time.Duration(delay) * time.Second)

    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Fprintf(w, "Delay: %d\n", delay)
    fmt.Printf("Served: %s\n", r.Host)
}

func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", myHandler)
    err := http.ListenAndServe(PORT, nil)
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
}

```

Как видите, в файле `slowWWW.go` можно не применять пакет `context`, потому что именно веб-клиент решает, сколько времени он готов ждать ответа.

Код функции `myHandler()` определяет, насколько медленно будет работать программа веб-сервера. Задержка может составлять от 0 до 14 секунд, что определяется вызовом функции `random(0, 15)`.

Если использовать веб-сервер `slowWWW.go` с помощью такого инструмента, как `wget(1)`, то получим результат следующего вида:

```

$ wget -qO- http://localhost:8001/
Serving: /
Delay: 4
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 13

```

Так происходит потому, что значение времени ожидания по умолчанию для `wget(1)` больше, чем у `slowWWW.go`. Если выполнить `useContext.go`, когда `slowWWW.go` уже запущен в другой оболочке UNIX, после обработки с помощью удобной утилиты `time(1)` получим следующий результат:

```

$ time go run useContext.go http://localhost:8001/ 1
Delay: 1
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...
real    0m1.374s
user    0m0.304s
sys     0m0.117s
$ time go run useContext.go http://localhost:8001/ 10
Delay: 10
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...
real    0m10.381s
user    0m0.314s
sys     0m0.125s
$ time go run useContext.go http://localhost:8001/ 15
Delay: 15
Connecting to http://localhost:8001/
Server Response: Serving: /
Delay: 13
Exiting...

```

```
real    0m13.379s
user    0m0.309s
sys     0m0.118s
```

Как видно из этих результатов, только третья команда действительно получила ответ от HTTP-сервера — для двух первых команд время ожидания истекло раньше.

Еще один пример использования пакета context

В этом подразделе вы еще ближе познакомитесь с пакетом `context` и узнаете, насколько это мощный и уникальный пакет стандартной библиотеки Go. На этот раз мы создадим контекст, в котором вместо функции `context.Background()` будет использоваться `context.TODO()`. Обе эти функции возвращают ненулевой пустой контекст, однако назначение у них разное. Мы также продемонстрируем применение функции `context.WithValue()`. Программа, которую мы рассмотрим, называется `moreContext.go`. Разделим ее на четыре части.

Первая часть `moreContext.go` выглядит так:

```
package main

import (
    "context"
    "fmt"
)

type aKey string
```

Вторая часть `moreContext.go` содержит следующий код Go:

```
func searchKey(ctx context.Context, k aKey) {
    v := ctx.Value(k)
    if v != nil {
        fmt.Println("found value:", v)
        return
    } else {
        fmt.Println("key not found:", k)
    }
}
```

Эта функция извлекает значение из контекста и проверяет, существует ли такое значение.

Третья часть `moreContext.go` содержит следующий код Go:

```
func main() {
    myKey := aKey("mySecretValue")
    ctx := context.WithValue(context.Background(), myKey, "mySecretValue")

    searchKey(ctx, myKey)
```

Функция `context.WithValue()` предоставляет способ связывания значения с объектом `Context`.

Обратите внимание, что контексты не должны храниться в структурах — их следует передавать в функции в виде отдельных аргументов. Рекомендуется передавать контекст в качестве первого аргумента функции.

Последняя часть `moreContext.go` выглядит так:

```
searchKey(ctx, aKey("notThere"))
emptyCtx := context.TODO()
searchKey(emptyCtx, aKey("notThere"))
}
```

В данном случае мы заявляем, что, несмотря на то что мы намерены использовать контекст операции, мы все еще не уверены в этом и потому используем функцию `context.TODO()`. К счастью, `TODO()` распознается средствами статического анализа, что позволяет им определить, правильно ли распространяются контексты.

Выполнение `moreContext.go` приведет к следующим результатам:

```
$ go run moreContext.go
found value: mySecretValue
key not found: notThere
key not found: notThere
```

Запомните: никогда не передавайте нулевой контекст — используйте функцию `context.TODO()` для создания подходящего контекста. И учтите, что `context.TODO()` следует использовать, если вы не уверены в том, какой объект `Context` хотите применить.

Пулы обработчиков

Пул обработчиков — это множество потоков, предназначенных для обработки назначаемых им заданий. Веб-сервер Apache и Go-пакет `net/http` работают приблизительно так: основной процесс принимает все входящие запросы, которые затем перенаправляются рабочим процессам для обработки. Как только рабочий процесс завершает свою работу, он готов к обслуживанию нового клиента.

Однако здесь есть главное различие: пул обработчиков использует не потоки, а горутин. Кроме того, потоки обычно не умирают после обработки запросов, потому что затраты на завершение потока и создание нового слишком высоки, тогда как горутина прекращает существовать после завершения работы. Как вы вскоре увидите, пулы обработчиков в Go реализованы с помощью буферизованных каналов, поскольку они позволяют ограничить число одновременно выполняемых горутин.

Рассмотрим программу `workerPool.go`. Разделим ее на пять частей. Эта программа решает простую задачу: она обрабатывает целые числа и выводит их квадраты

посредством отдельной горутины для обслуживания каждого запроса. Несмотря на простоту `workerPool.go`, код Go программы вполне можно использовать как шаблон для реализации более сложных задач.



Это расширенный метод, он поможет вам разрабатывать серверные процессы на Go, способные принимать и обслуживать несколько клиентов с использованием горутин.

Первая часть `workerPool.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

type Client struct {
    id      int
    integer int
}

type Data struct {
    job      Client
    square   int
}
```

Здесь мы видим метод с применением структуры `Client` для назначения уникального идентификатора каждому запросу, который мы намерены обработать. Структура `Data` нужна для группировки данных `Client` с реальными результатами, сгенерированными программой. Проще говоря, структура `Client` содержит входные данные каждого запроса, а структура `Data` — результаты запроса.

Вторая часть `workerPool.go` содержит следующий код Go:

```
var (
    size      = 10
    clients = make(chan Client, size)
    data = make(chan Data, size)
)

func worker(w *sync.WaitGroup) {
    for c := range clients {
        square := c.integer * c.integer
        output := Data{c, square}
    }
}
```

```

        data <- output
        time.Sleep(time.Second)
    }
    w.Done()
}

```

Этот код состоит из двух интересных частей. Первая часть создает три глобальные переменные. Буферизованные каналы `clients` и `data` используются для получения новых клиентских запросов и записи результатов соответственно. Если вы хотите, чтобы программа работала быстрее, можете увеличить значение параметра `size`.

Вторая часть — это реализация функции `worker()`, которая читает канал `clients` и получает оттуда новые запросы на обслуживание. После завершения обработки результат записывается в канал `data`. Задержка, которая вводится с помощью оператора `time.Sleep(time.Second)`, не является необходимой, но дает лучшее представление о том, как будут выводиться на экран результаты программы.

Наконец, не забудьте использовать в функции `worker()` указатель для параметра `sync.WaitGroup`, иначе переменная `sync.WaitGroup` скопируется и будет бесполезной.

Третья часть `workerPool.go` содержит следующий код Go:

```

func makeWP(n int) {
    var w sync.WaitGroup
    for i := 0; i < n; i++ {
        w.Add(1)
        go worker(&w)
    }
    w.Wait()
    close(data)
}

func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}

```

В этом коде реализованы две функции — `makeWP()` и `create()`. Назначение функции `makeWP()` состоит в том, чтобы сгенерировать необходимое количество горутин `worker()` для обработки всех запросов. Функция `w.Add(1)` вызывается из `makeWP()`, однако `w.Done()` вызывается из `worker()` после того, как обработчик завершит свою работу. Назначение функции `create()` — правильно создать все запросы, используя тип `Client`, а затем записать их в канал `clients` для обработки. Обратите внимание, что канал `clients` читается функцией `worker()`.

Четвертый фрагмент кода `workerPool.go` выглядит так:

```
func main() {
    fmt.Println("Capacity of clients:", cap(clients))
    fmt.Println("Capacity of data:", cap(data))

    if len(os.Args) != 3 {
        fmt.Println("Need #jobs and #workers!")
        os.Exit(1)
    }

    nJobs, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    nWorkers, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

В этом коде мы считываем аргументы командной строки. Однако сначала здесь показано, как можно использовать функцию `cap()`, чтобы определить пропускную способность канала.

Если количество обработчиков больше, чем размер буферизованного канала `clients`, то количество создаваемых горутин будет равно размеру канала `clients`. Аналогично если количество заданий больше, чем количество обработчиков, то задания будут обрабатываться меньшими группами.

Программа позволяет определить количество обработчиков и количество заданий с помощью аргументов командной строки. Однако, чтобы изменить размер каналов `clients` и `data`, нужно внести изменения в исходный код программы.

Остальной код `workerPool.go` выглядит так:

```
go create(nJobs)
finished := make(chan interface{})
go func() {
    for d := range data {
        fmt.Printf("Client ID: %d\tint: ", d.job.id)
        fmt.Printf("%d\tsquare: %d\n", d.job.integer, d.square)
    }
    finished <- true
}()
makeWP(nWorkers)
fmt.Printf(": %v\n", <-finished)
}
```

Сначала мы вызываем функцию `create()`, чтобы имитировать клиентские запросы, которые следует обработать. Для чтения канала `data` и вывода результатов на экран применяется анонимная горутина. Канал `finished` используется для блокировки программы до тех пор, пока анонимная программа не закончит чтение канала `data`. Поэтому канал `finished` не нуждается в конкретном типе.

Затем мы вызываем функцию `makeWP()`, которая и выполняет обработку запросов. Оператор `<-finished` в блоках `fmt.Printf()` не позволяет программе завершиться, пока кто-нибудь не напишет что-либо в канал `finished`. Этот «кто-то» на самом деле — анонимная горутина в `main()`. Кроме того, хотя анонимная горутина и записывает значение `true` в канал `finished`, мы могли бы записать туда значение `false` и получить тот же результат, который разблокирует функцию `main()`. Не верите — попробуйте сами!

Выполнение `workerPool.go` приведет к результатам следующего вида:

```
$ go run workerPool.go 15 5
Capacity of clients: 10
Capacity of data: 10
Client ID: 0    int: 0      square: 0
Client ID: 4    int: 4      square: 16
Client ID: 1    int: 1      square: 1
Client ID: 3    int: 3      square: 9
Client ID: 2    int: 2      square: 4
Client ID: 5    int: 5      square: 25
Client ID: 6    int: 6      square: 36
Client ID: 7    int: 7      square: 49
Client ID: 8    int: 8      square: 64
Client ID: 9    int: 9      square: 81
Client ID: 10   int: 10     square: 100
Client ID: 11   int: 11     square: 121
Client ID: 12   int: 12     square: 144
Client ID: 13   int: 13     square: 169
Client ID: 14   int: 14     square: 196
```

Если вы хотите обслуживать каждый запрос, не ожидая ответа от него в функции `main()`, как это сделано в `workerPool.go`, то у вас будет меньше поводов для беспокойства. Простой способ применять горутин для обработки запросов и получения ответа от них в функции `main()` — использовать общую память или управляющий процесс, который будет собирать данные, а не просто выводить их на экран.

Наконец, работа программы `workerPool.go` сильно упрощена, поскольку функция `worker()` не может завершиться с ошибкой. Это придется учитывать при работе с реальными компьютерными сетями или с другими видами ресурсов, которые могут дать сбой.

Дополнительные ресурсы

Следующие ресурсы будут вам очень полезны:

- ❑ посетите страницу документации пакета `sync`, которую вы найдете по адресу <https://golang.org/pkg/sync/>;
- ❑ посетите страницу документации пакета `context` по адресу <https://golang.org/pkg/context/>;
- ❑ чтобы больше узнать о реализации планировщика Go, посетите страницу <https://golang.org/src/runtime/proc.go>;
- ❑ страницу документации пакета `atomic` вы найдете по адресу <https://golang.org/pkg/sync/atomic/>;
- ❑ просмотрите проектную документацию планировщика Go по адресу <https://golang.org/s/go11sched>.

Упражнения

- ❑ Попробуйте реализовать конкурентную версию `wc(1)`, которая бы использовала буферизованный канал.
- ❑ Затем попробуйте реализовать конкурентную версию `wc(1)`, которая бы использовала общую память.
- ❑ Наконец, попробуйте реализовать конкурентную версию `wc(1)`, в которой бы использовалась управляющая горутина.
- ❑ Измените код Go файла `workerPool.go` таким образом, чтобы сохранять результаты в файле. При работе с файлом примените мьютекс и критический раздел или управляющую горутину, которая будет записывать данные на диск.
- ❑ Что произойдет с программой `workerPool.go`, если значение глобальной переменной `size` станет равным 1? Почему?
- ❑ Измените код Go программы `workerPool.go` таким образом, чтобы реализовать функциональность утилиты командной строки `wc(1)`.
- ❑ Измените код Go программы `workerPool.go` таким образом, чтобы размер буферизованных каналов `clients` и `data` можно было задавать в виде аргументов командной строки.
- ❑ Попробуйте написать конкурентную версию утилиты командной строки `find(1)`, в которой бы применялась управляющая горутина.
- ❑ Измените код `simpleContext.go` таким образом, чтобы анонимная функция, используемая в функциях `f1()`, `f2()` и `f3()`, стала отдельной функцией. Какая основная проблема возникает при таком изменении кода?

- ❑ Измените код Go программы `simpleContext.go` таким образом, чтобы функции `f1()`, `f2()` и `f3()` использовали созданную извне переменную `Context` вместо определения собственной такой переменной.
- ❑ Измените код Go программы `useContext.go` таким образом, чтобы вместо функции `context.WithTimeout()` использовать `context.WithDeadline()` или `context.WithCancel()`.
- ❑ Наконец, попробуйте реализовать конкурентную версию утилиты командной строки `find(1)` с помощью мьютекса `sync.Mutex`.

Резюме

В этой главе рассмотрено много важных тем, связанных с горутинами и каналами. Однако в основном это касалось функциональности оператора `select`. Благодаря возможностям оператора `select` каналы являются предпочтительным способом Go для организации взаимодействия между компонентами конкурентной программы Go, в которой задействовано несколько горутин. Кроме того, в этой главе продемонстрировано использование стандартного Go-пакета `context`, который иногда бывает незаменим.

В конкурентном программировании есть много правил; однако самое важное из них заключается в том, что следует избегать общих данных, если только у вас нет на то веских причин! Общие данные являются причиной всех ошибок при конкурентном программировании.

Главное, что следует запомнить, прочитав эту главу, — раньше общая память была единственным способом обмена данными между потоками одного процесса, сейчас же Go предлагает более удобные способы коммуникации. Поэтому подумайте с точки зрения Go, прежде чем принять решение об использовании общей памяти в вашем коде Go. Если же вам действительно нужно это использовать, вы можете задействовать управляющую горутину.

Основные темы следующей главы — тестирование, оптимизация и профилирование кода средствами Go. Кроме этого, вы познакомитесь с тестированием кода Go, кросс-компиляцией и поиском недоступного кода в Go.

В конце вы узнаете, как документировать код Go и генерировать документацию в формате HTML с помощью утилиты `godoc`.

11

Тестирование, оптимизация и профилирование кода

В предыдущей главе мы обсудили конкурентность в Go, мьютексы, пакет `atomic`, различные типы каналов, состояние гонки и то, как оператор `select` позволяет использовать каналы в качестве связующего звена для управления горутинами и предоставления им возможности обмена информацией.

Темы, которые рассмотрены в этой главе, очень практичны и важны, особенно если вы хотите повысить производительность ваших программ Go и быстро обнаруживать ошибки. Эта глава прежде всего посвящена оптимизации, тестированию, документированию и профилированию кода.

Оптимизация кода — это процесс, при котором один или несколько разработчиков стремятся ускорить выполнение определенных частей программы, повысить их эффективность или использовать меньше ресурсов. Проще говоря, оптимизация кода заключается в устранении недостатков программы.

Тестирование кода — это проверка того, что код выполняет то, что от него требуется. В этой главе вы познакомитесь с методом тестирования кода в Go. Лучше всего писать код для тестирования программ на этапе разработки, поскольку это помогает выявить ошибки в коде на самых ранних стадиях.

Профилирование кода означает измерение определенных аспектов программы, что позволяет составить подробно представление о том, как работает код. Результаты профилирования кода помогают принять решение о том, какие части кода нуждаются в изменении.

Я надеюсь, что вы уже осознали важность документирования кода для описания тех решений, которые вы приняли в процессе разработки реализации программы. В этой главе вы увидите, как Go помогает сгенерировать документацию для реализуемых модулей.



Документация настолько важна, что некоторые разработчики сначала пишут документацию и только потом код! Однако действительно важно, чтобы документация программы говорила о том, что делает функциональность программы.

В этой главе рассмотрены следующие темы:

- ❑ профилирование кода Go;
- ❑ утилита `go tool pprof`;
- ❑ использование веб-интерфейса профилирования Go;
- ❑ тестирование кода Go;
- ❑ команда `go test`;
- ❑ утилита `go tool trace`;
- ❑ удобный пакет `testing/quick`;
- ❑ бенчмаркинг кода Go;
- ❑ кросс-компиляция;
- ❑ тестирование покрытия кода;
- ❑ автоматизированное создание документации для кода Go; создание примеров функций;
- ❑ поиск недоступного кода Go в программах.

Оптимизация

Оптимизация кода — это и искусство, и наука. Не существует четко определенного способа, который бы помог вам оптимизировать код Go или любой другой код на любом языке программирования. Вам придется хорошо подумать и многое сделать, если вы хотите, чтобы ваш код заработал быстрее.



Убедитесь, что оптимизируете код, который не содержит ошибок: нет смысла оптимизировать ошибки. Если в вашей программе есть ошибки, ее нужно сначала отладить.

Если вы действительно интересуетесь оптимизацией кода, то советую прочитать книгу «Компиляторы. Принципы, технологии и инструментарий»¹, в которой основное внимание уделяется строению компилятора. Кроме того, все тома серии «Искусство программирования»² Дональда Кнута являются отличными источниками знаний по всем аспектам программирования.

¹ Ахо А. В., Лам М. С., Сети Р., Ульман Д. Д. Компиляторы. Принципы, технологии и инструментарий, 2-е изд. — М.: Вильямс, 2017.

² Кнут Д. Искусство программирования. В 4 т. — М.: Вильямс.

Всегда помните, что сказал Кнут об оптимизации:

«Настоящая проблема заключается в том, что программисты тратят слишком много времени на повышение эффективности не в тех местах и не в тот момент; преждевременная оптимизация – корень всех бед в программировании (или по крайней мере большей их части)».

Также помните, что сказал об оптимизации Джо Армстронг, один из разработчиков Erlang:

«Сначала сделай так, чтобы это работало, потом сделай так, чтобы оно было красивым, и только потом, если это действительно нужно, сделай так, чтобы оно работало быстро. В 90 % случаев, если ты сделаешь это красивым, оно само по себе будет работать быстро. Поэтому просто сделай это по-настоящему красивым!»

К тому же лишь небольшая часть программы должна быть оптимизирована. В таких случаях для реализации определенных функций Go можно использовать язык ассемблера. Это по-настоящему хороший вариант, он окажет огромное влияние на производительность ваших программ.

Оптимизация кода Go

Как уже отмечалось, оптимизация кода — это процесс, в ходе которого мы пытаемся обнаружить части кода, влияющие на производительность всей программы, и стараемся сделать так, чтобы они работали быстрее или использовали меньше ресурсов.

Один из следующих разделов этой главы, посвященный *бенчмаркингу*, поможет вам понять, что именно происходит внутри программы при выполнении кода и какие параметры программы оказывают наиболее сильное влияние на ее производительность. Однако не стоит недооценивать важность здравого смысла. Проще говоря, если одна из функций выполняется в 10 000 раз чаще, чем остальные функции программы, попробуйте для начала оптимизировать ее.



Общий совет по оптимизации: оптимизируйте только тот код, который не содержит ошибок. Другими словами, следует оптимизировать только рабочий код. Поэтому сначала напишите правильный код, пускай он даже работает медленно. В конце концов, самая распространенная ошибка, которую допускают программисты, — это попытка оптимизировать первую же версию своего кода. А это влечет за собой остальные ошибки!

Напомню: оптимизация кода — это одновременно и искусство, и наука, а следовательно, довольно сложная задача. Следующий раздел, посвященный профилированию кода Go, определенно поможет вам в оптимизации кода, потому что основная цель профилирования — найти недостатки в коде, чтобы оптимизировать самые важные части программы.

Профилирование кода Go

Профилирование — это динамический анализ программы, в ходе которого измеряются различные величины, связанные с выполнением программы, что позволяет лучше понять поведение программы. В этом разделе вы узнаете, как профилировать код Go, чтобы лучше понять, как он работает, и повысить его производительность. Иногда профилирование кода даже помогает найти ошибки!

Для начала мы воспользуемся интерфейсом командной строки профилировщика Go. Затем мы будем использовать веб-интерфейс профилировщика Go.

Запомните самое важное: если вы хотите профилировать код Go, для этого вам нужно импортировать стандартный Go пакет `runtime/pprof`, прямо или косвенно. Чтобы открыть страницу справки инструмента `pprof`, нужно выполнить команду `go tool pprof -help`. В ответ вы получите много интересных сведений.

Стандартный Go-пакет `net/http/pprof`

В комплект поставки Go входит стандартный Go-пакет низкого уровня `runtime/pprof`, однако существует также пакет высокого уровня `net/http/pprof`. Именно его следует использовать, если вы хотите профилировать веб-приложение, написанное на Go. В этой главе не будет обсуждаться создание HTTP-серверов на Go, подробнее о пакете `net/http/pprof` вы узнаете в главе 12.

Простой пример профилирования

Go поддерживает два вида профилирования: *профилирование процессора* и *профилирование памяти*. Выполнять для одного приложения оба вида профилирования одновременно не рекомендуется, поскольку они плохо совместимы. Однако рассмотренное далее приложение `profileMe.go` является исключением из правила, поскольку используется для иллюстрации обоих методов.

Мы сохраним профилируемый код Go в файле `profileMe.go` и рассмотрим его, разделив на пять частей.

Первая часть `profileMe.go` содержит следующий код Go:

```
package main

import (
```

```

    "fmt"
    "math"
    "os"
    "runtime"
    "runtime/pprof"
    "time"
)

func fibo1(n int) int64 {
    if n == 0 || n == 1 {
        return int64(n)
    }
    time.Sleep(time.Millisecond)
    return int64(fibo2(n-1)) + int64(fibo2(n-2))
}

```

Обратите внимание, что для создания данных профилирования необходимо импортировать в вашу программу пакет `runtime/pprof`, явно или неявно. Функция `time.Sleep()` вызывается из функции `fibo1()` для того, чтобы немного ее замедлить. Зачем это нужно, вы узнаете в конце этого подраздела.

Второй фрагмент кода `profileMe.go` выглядит так:

```

func fibo2(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    time.Sleep(50 * time.Millisecond)
    return fn[n]
}

```

Этот код представляет собой реализацию еще одной функции Go, в которой использован другой алгоритм определения чисел из последовательности Фибоначчи.

В третьей части `profileMe.go` содержится следующий код Go:

```

func N1(n int) bool {
    k := math.Floor(float64(n/2 + 1))
    for i := 2; i < int(k); i++ {
        if (n % i) == 0 {
            return false
        }
    }
}

```

```

    }
    return true
}

func N2(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}

```

Функции N1() и N2() используются для того, чтобы определить, является ли данное целое число простым. Первая функция оптимизирована, поскольку в ней цикл `for` выполняет примерно вдвое меньше итераций, чем в цикле `for` функции N2(). Поскольку обе функции относительно медленные, здесь нет необходимости обращаться к `time.Sleep()`.

Четвертый фрагмент кода `profileMe.go` выглядит так:

```

func main() {
    cpuFile, err := os.Create("/tmp/cpuProfile.out")
    if err != nil {
        fmt.Println(err)
        return
    }
    pprof.StartCPUProfile(cpuFile)
    defer pprof.StopCPUProfile()
    total := 0
    for i := 2; i < 100000; i++ {
        n := N1(i)
        if n {
            total = total + 1
        }
    }
    fmt.Println("Total primes:", total)
    total = 0
    for i := 2; i < 100000; i++ {
        n := N2(i)
        if n {
            total = total + 1
        }
    }
    fmt.Println("Total primes:", total)
    for i := 1; i < 90; i++ {
        n := fibo1(i)
        fmt.Print(n, " ")
    }
}

```



```

fmt.Println()
for i := 1; i < 90; i++ {
    n := fibo2(i)
    fmt.Print(n, " ")
}
fmt.Println()
runtime.GC()

```

Вызов функции `os.Create()` нужен для получения файла, в который записываются данные профилирования. Вызов `pprof.StartCPUProfile()` начинает профилирование процессора для программы, а вызов `pprof.StopCPUProfile()` останавливает его.

Если вы хотите многократно создавать и использовать временные файлы и каталоги, обратите внимание на функции `ioutil.TempFile()` и `ioutil.TempDir()` соответственно.

Последняя часть `profileMe.go` выглядит так:

```

// Профилирование памяти!
memory, err := os.Create("/tmp/memoryProfile.out")
if err != nil {
    fmt.Println(err)
    return
}
defer memory.Close()
for i := 0; i < 10; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(50 * time.Millisecond)
}
err = pprof.WriteHeapProfile(memory)
if err != nil {
    fmt.Println(err)
    return
}
}

```

В этой последней части программы показано, как работает метод *профилирования памяти*. Он очень похож на профилирование процессора, и для записи данных профилирования также нужен файл.

Выполнение `profileMe.go` приведет к следующим результатам:

```

$ go run profileMe.go
Total primes: 9592
Total primes: 9592

```

```

1 2 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
956722026041 1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853 72723460248141
117669030460994 190392490709135 308061521170129 498454011879264
806515533049393 1304969544928657 2111485077978050 3416454622906707
5527939700884757 8944394323791464 14472334024676221 23416728348467685
37889062373143906 61305790721611591 99194853094755497 160500643816367088
259695496911122585 420196140727489673 679891637638612258
1100087778366101931 1779979416004714189
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
956722026041 1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853 72723460248141
117669030460994 190392490709135 308061521170129 498454011879264
806515533049393 1304969544928657 2111485077978050 3416454622906707
5527939700884757 8944394323791464 14472334024676221 23416728348467685
37889062373143906 61305790721611591 99194853094755497 160500643816367088
259695496911122585 420196140727489673 679891637638612258
1100087778366101931 1779979416004714189

```

Помимо этого, программа будет записывать данные профилирования в два файла:

```

$ cd /tmp
$ ls -l *Profile*
-rw-r--r-- 1 mtsouk wheel 1557 Apr 24 16:37 cpuProfile.out
-rw-r--r-- 1 mtsouk wheel 438 Apr 24 16:37 memoryProfile.out

```

Только после того, как данные профилирования будут собраны, можно приступить к их анализу. Итак, теперь можно запустить профилировщик командной строки, чтобы проверить данные процессора:

```

$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)

```

Если нажать `help` в оболочке профилировщика, то получим следующий результат:

(pprof) help

Commands:

<code>callgrind</code>	Outputs a graph in callgrind format
<code>comments</code>	Output all profile comments
<code>disasm</code>	Output assembly listings annotated with samples
<code>dot</code>	Outputs a graph in DOT format
<code>eog</code>	Visualize graph through eog
<code>evince</code>	Visualize graph through evince
<code>gif</code>	Outputs a graph image in GIF format
<code>gv</code>	Visualize graph through gv
<code>kcachegrind</code>	Visualize report in KCachegrind
<code>list</code>	Output annotated source for functions matching regexp
<code>pdf</code>	Outputs a graph in PDF format
<code>peek</code>	Output callers/callees of functions matching regexp
<code>png</code>	Outputs a graph image in PNG format
<code>proto</code>	Outputs the profile in compressed protobuf format
<code>ps</code>	Outputs a graph in PS format
<code>raw</code>	Outputs a text representation of the raw profile
<code>svg</code>	Outputs a graph in SVG format
<code>tags</code>	Outputs all tags in the profile
<code>text</code>	Outputs top entries in text form
<code>top</code>	Outputs top entries in text form
<code>topproto</code>	Outputs top entries in compressed protobuf format
<code>traces</code>	Outputs all profile samples in text form
<code>tree</code>	Outputs a text rendering of call graph
<code>web</code>	Visualize graph through web browser
<code>weblist</code>	Display annotated source in a web browser
<code>o/options</code>	List options and their current values
<code>quit/exit/^D</code>	Exit pprof

Options:

<code>call_tree</code>	Create a context-sensitive call tree
<code>compact_labels</code>	Show minimal headers
<code>divide_by</code>	Ratio to divide all samples before visualization
<code>drop_negative</code>	Ignore negative differences
<code>edgefraction</code>	Hide edges below <f>*total
<code>focus</code>	Restricts to samples going through a node matching regexp
<code>hide</code>	Skips nodes matching regexp
<code>ignore</code>	Skips paths going through any nodes matching regexp
<code>mean</code>	Average sample value over first value (count)
<code>nodecount</code>	Max number of nodes to show
<code>nodefraction</code>	Hide nodes below <f>*total
<code>noinlines</code>	Ignore inlines.
<code>normalize</code>	Scales profile based on the base profile.
<code>output</code>	filename for file-based outputs
<code>prune_from</code>	Drops any functions below the matched frame.
<code>relative_percentages</code>	Show percentages relative to focused subgraph
<code>sample_index</code>	Sample value to report (0-based index or name)
<code>show</code>	Only show nodes matching regexp

```

show_from      Drops functions above the highest matched frame.
source_path    Search path for source files
tagfocus      Restricts to samples with tags in range or matched by
regexp
taghide        Skip tags matching this regexp
tagignore      Discard samples with tags in range or matched by regexp
tagshow        Only consider tags matching this regexp
trim           Honor nodefraction/edgefraction/nodecount defaults
trim_path      Path to trim from source paths before search
unit           Measurement units to display
Option groups (only set one per group):
cumulative
  cum          Sort entries based on cumulative weight
  flat         Sort entries based on own weight
granularity
  addresses    Aggregate at the address level.
  filefunctions Aggregate at the function level.
  files        Aggregate at the file level.
  functions    Aggregate at the function level.
  lines        Aggregate at the source code line level.
:             Clear focus/ignore/hide/tagfocus/tagignore
type "help <cmd|option>" for more information
(pprof)

```



Найдите время для ознакомления со всеми командами утилиты go tool pprof и проверьте их.

Команда top возвращает десять лучших записей в текстовой форме:

```

(pprof) top
Showing nodes accounting for 4.42s, 99.10% of 4.46s total
Dropped 14 nodes (cum <= 0.02s)
Showing top 10 nodes out of 19

```

flat	flat%	sum%	cum	cum%	
2.69s	60.31%	60.31%	2.69s	60.31%	main.N2
1.41s	31.61%	91.93%	1.41s	31.61%	main.N1
0.19s	4.26%	96.19%	0.19s	4.26%	runtime.nanotime
0.10s	2.24%	98.43%	0.10s	2.24%	runtime.usleep
0.03s	0.67%	99.10%	0.03s	0.67%	runtime.memclrNoHeapPointers
0	0%	99.10%	4.14s	92.83%	main.main
0	0%	99.10%	0.03s	0.67%	runtime.(*mheap).alloc
0	0%	99.10%	0.03s	0.67%	runtime.largeAlloc
0	0%	99.10%	4.14s	92.83%	runtime.main
0	0%	99.10%	0.03s	0.67%	runtime.makeslice

Функции, представленные в первых строках выходных состояний, отвечают за 99,10 % общего времени выполнения программы.

В частности, функция main.N2 отвечает за 60,31% всего времени выполнения программы.

Команда `top10 --cum` возвращает совокупное время выполнения для каждой функции:

```
(pprof) top10 --cum
Showing nodes accounting for 4390ms, 98.43% of 4460ms total
Dropped 14 nodes (cum <= 22.30ms)
Showing top 10 nodes out of 19
```

flat	flat%	sum%	cum	cum%	
0	0%	0%	4140ms	92.83%	main.main
0	0%	0%	4140ms	92.83%	runtime.main
2690ms	60.31%	60.31%	2690ms	60.31%	main.N2
1410ms	31.61%	91.93%	1410ms	31.61%	main.N1
0	0%	91.93%	290ms	6.50%	runtime.mstart
0	0%	91.93%	270ms	6.05%	runtime.mstart1
0	0%	91.93%	270ms	6.05%	runtime.sysmon
190ms	4.26%	96.19%	190ms	4.26%	runtime.nanotime
100ms	2.24%	98.43%	100ms	2.24%	runtime.usleep
0	0%	98.43%	50ms	1.12%	runtime.systemstack

А как узнать, что происходит с каждой отдельной функцией? Для этого можно использовать команду `list`, в которой указать имя функции и имя пакета, и тогда мы получим более развернутую информацию о производительности этой функции:

```
(pprof) list main.N1
Total: 4.18s
ROUTINE ===== main.N1 in /Users/mtsouk/ch11/code/profileMe.go
  1.41s      1.41s  (flat, cum) 31.61% of Total
    .        .      32:  return fn[n]
    .        .      33:  }
    .        .      34:
    .        .      35: func N1(n int) bool {
    .        .      36:     k := math.Floor(float64(n/2 + 1))
  60ms      60ms    37:     for i := 2; i < int(k); i++ {
  1.35s     1.35s    38:         if (n % i) == 0 {
    .        .      39:             return false
    .        .      40:         }
    .        .      41:     }
    .        .      42:     return true
    .        .      43: }
(pprof)
```

Как видно из этого результата, цикл `for` функции `main.N1` отвечает почти за все время выполнения этой функции. В частности, оператор `if (n % i) == 0` занимает 1.35s из 1.41s времени выполнения всей функции.

Из оболочки профилировщика Go также можно создать PDF-файл данных профилирования, воспользовавшись командой `pdf`:

```
(pprof) pdf
Generating report in profile001.pdf
```

Обратите внимание, что для создания PDF-файла, который потом можно посмотреть с помощью любого PDF-ридера, вам понадобится программа *Graphviz*.

И последнее предупреждение: если ваша программа выполняется слишком быстро, то у профилировщика не будет достаточно времени, чтобы сделать необходимые измерения, и при загрузке файла данных вы можете увидеть сообщение `Total samples = 0`. В этом случае вам не удастся получить сколько-нибудь полезную информацию в результате профилирования. Именно поэтому в некоторых функциях `profileMe.go` мы использовали `time.Sleep()`:

```
$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Удобный внешний пакет для профилирования

В этом подразделе продемонстрировано использование внешнего пакета Go, который настраивает среду профилирования. Он гораздо удобнее, чем стандартный Go-пакет `runtime/pprof`. Мы рассмотрим внешний пакет на примере программы `betterProfile.go`. Разделим ее на три части.

Первая часть `betterProfile.go` выглядит так:

```
package main

import (
    "fmt"
    "github.com/pkg/profile"
)

var VARIABLE int

func N1(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

В этом коде показано использование внешнего Go-пакета, который размещен по адресу github.com/pkg/profile. Вы можете его скачать с помощью команды `go get`:

```
$ go get github.com/pkg/profile
```

Второй фрагмент файла `bestProfile.go` содержит следующий код Go:

```
func Multiply(a, b int) int {
    if a == 1 {
        return b
    }
    if a == 0 || b == 0 {
        return 0
    }
    if a < 0 {
        return -Multiply(-a, b)
    }
    return b + Multiply(a-1, b)
}

func main() {
    defer profile.Start(profile.ProfilePath("/tmp")).Stop()
```

Для *профилирования процессора* при помощи пакета `github.com/pkg/profile`, разработанного Дэйвом Чейни (Dave Cheney), достаточно добавить в приложение Go всего один оператор. Для того чтобы выполнить *профилирование памяти*, вместо этого оператора нужно вставить следующий:

```
defer profile.Start(profile.MemProfile).Stop()
```

Оставшийся код Go программы выглядит так:

```
total := 0
for i := 2; i < 200000; i++ {
    n := N1(i)
    if n {
        total++
    }
}

fmt.Println("Total: ", total)
total = 0
for i := 0; i < 5000; i++ {
    for j := 0; j < 400; j++ {
        k := Multiply(i, j)
        VARIABLE = k
        total++
    }
}
fmt.Println("Total: ", total)
}
```

Выполнение `betterProfile.go` приведет к следующим результатам:

```
$ go run betterProfile.go
2019/04/24 16:44:05 profile: cpu profiling enabled, /tmp/cpu.pprof
Total: 17984
Total: 2000000
2019/04/24 16:44:33 profile: cpu profiling disabled, /tmp/cpu.pprof
```



Пакет `github.com/pkg/profile` нужен для сбора данных; обработка данных выполняется так же, как и раньше.

```
$ go tool pprof /tmp/cpu.pprof
Type: cpu
Time: Apr 24, 2019 at 4:44pm (EEST)
Duration: 27.40s, Total samples = 25.10s (91.59%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

Веб-интерфейс Go-профилеровщика

К счастью, в Go версии 1.10 у команды `go tool pprof` появился пользовательский веб-интерфейс.



Чтобы пользовательский веб-интерфейс смог работать, необходимо установить `Graphviz`, а ваш веб-браузер должен поддерживать JavaScript. В целях безопасности используйте Chrome или Firefox.

Интерактивный профилировщик Go запускается с помощью такой команды:

```
$ go tool pprof -http=[host]:[port] aProfile
```

Пример профилирования с использованием веб-интерфейса

Для изучения веб-интерфейса профилировщика Go мы будем использовать данные, полученные при выполнении `profileMe.go`, поскольку для этого не нужно создавать отдельный пример кода.

Как вы уже знаете, для этого сначала нужно выполнить следующую команду:

```
$ go tool pprof -http=localhost:8080 /tmp/cpuProfile.out
Main binary filename not available.
```

На рис. 11.1 представлен начальный экран пользовательского веб-интерфейса профилировщика Go после выполнения предыдущей команды.

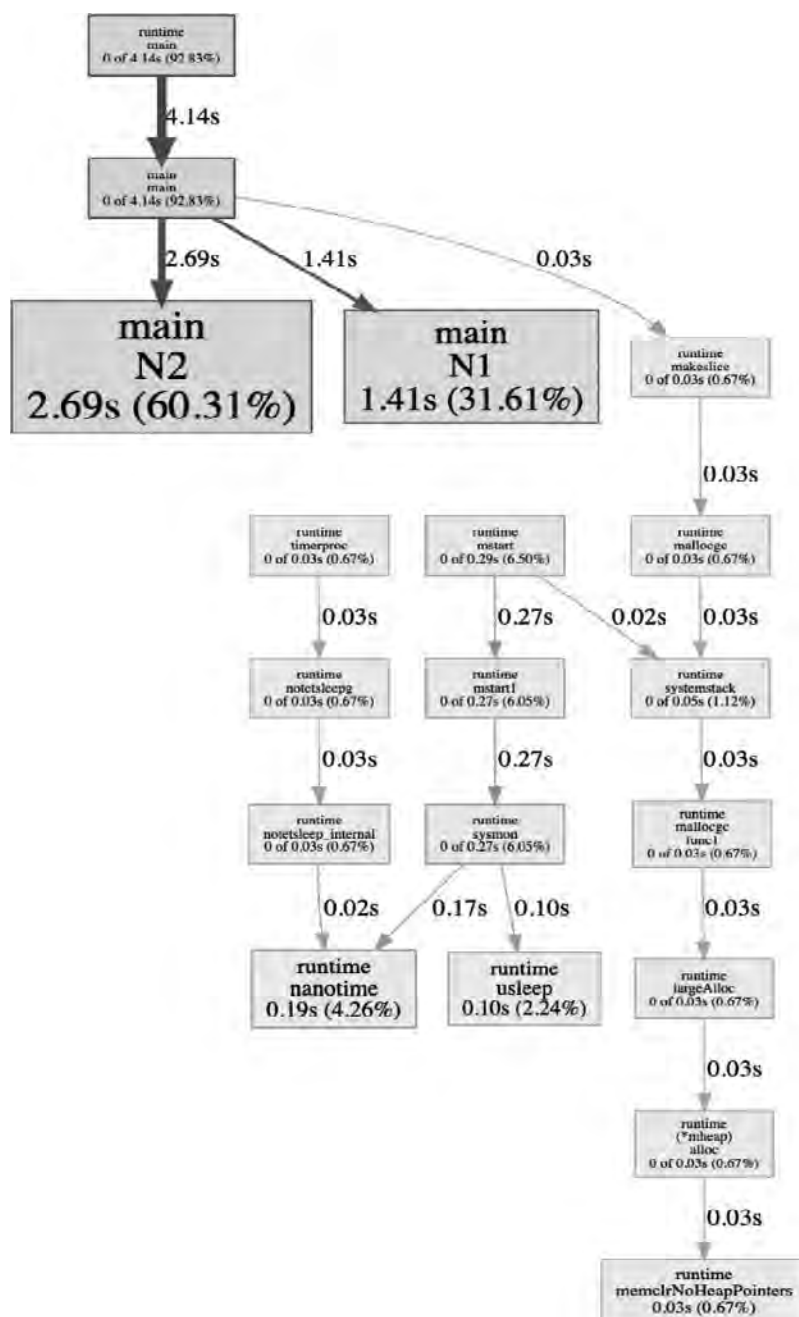


Рис. 11.1. Веб-интерфейс профилировщика Go

Аналогично на рис. 11.2 показан URL-адрес профилировщика Go `http://localhost:8080/ui/source`, по которому выводится аналитическая информация для каждой функции программы.

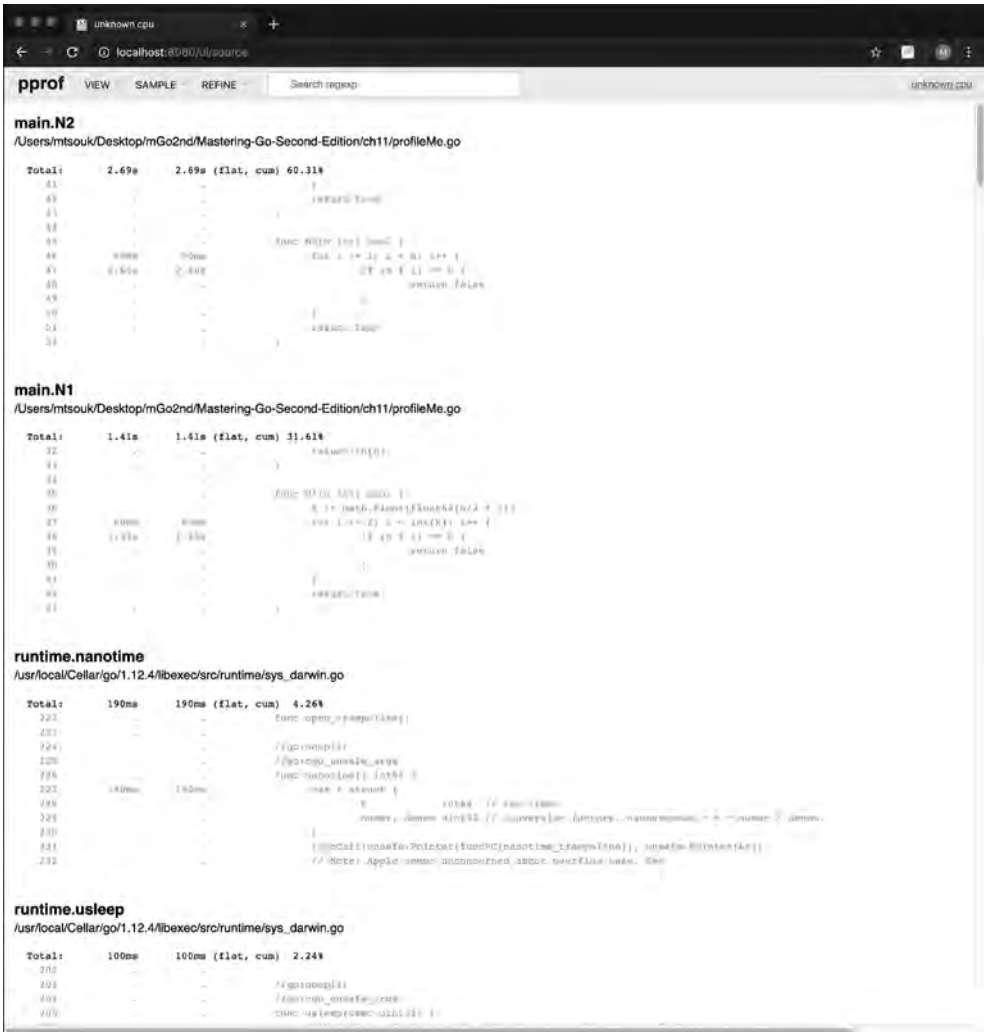


Рис. 11.2. Использование URL-адреса `/source` профилировщика Go



Поскольку я не могу показать каждую страницу веб-интерфейса профилировщика Go, вам следует познакомиться с ним самостоятельно — это отличный инструмент для проверки работы ваших программ.

Коротко об основах Graphviz

Graphviz — это очень удобное сочетание утилит и компьютерного языка, который позволяет рисовать сложные графики. Строго говоря, Graphviz — это набор инструментов для манипулирования направленными и ненаправленными *графовыми* структурами и создания графовых структур.

У Graphviz есть собственный язык *DOT* — простой, изящный и мощный. Преимущество Graphviz состоит в том, что вы можете писать его код в простом текстовом редакторе. Замечательным вторичным эффектом этого свойства является возможность легко разрабатывать сценарии, которые генерируют код Graphviz. Кроме того, большинство языков программирования, включая Python, Ruby, C++ и Perl, имеют собственные интерфейсы для создания файлов Graphviz с использованием собственного кода.



Вы не обязаны знать все о Graphviz, чтобы использовать веб-интерфейс профилировщика Go. Но полезно понимать, как работает Graphviz и как выглядит его код.

Следующий код Graphviz, который сохраняется как `graph.dot`, кратко иллюстрирует работу Graphviz и внешний вид языка Graphviz:

```
digraph G
{
    graph [dpi = 300, bgcolor = "gray"];
    rankdir = LR;
    node [shape=record, width=.2, height=.2, color="white" ];
    node0 [label = "<p0>; |<p1>|<p2>|<p3>|<p4>| | ", height = 3];
    node[ width=2 ];
    node1 [label = "{<e> r0 | 123 | <p> }", color="gray" ];
    node2 [label = "{<e> r10 | 13 | <p> }" ];
    node3 [label = "{<e> r11 | 23 | <p> }" ];
    node4 [label = "{<e> r12 | 326 | <p> }" ];
    node5 [label = "{<e> r13 | 1f3 | <p> }" ];
    node6 [label = "{<e> r20 | 143 | <p> }" ];
    node7 [label = "{<e> r40 | b23 | <p> }" ];
    node0:p0 -> node1:e [dir=both color="red:blue"];
    node0:p1 -> node2:e [dir=back arrowhead=diamond];
    node2:p -> node3:e;
    node3:p -> node4:e [dir=both arrowtail=box color="red"];
    node4:p -> node5:e [dir=forward];
    node0:p2 -> node6:e [dir=none color="orange"];
    node0:p4 -> node7:e;
}
```

Атрибут `color` меняет цвет узла, а атрибут `shape` — форму узла. Кроме того, атрибут `dir`, который можно применить к ребрам, определяет, будет ли у ребра две

стрелки, одна или ни одной. Стиль стрелки определяется атрибутами `arrowhead` и `arrowtail`.

Скомпилировав этот код с использованием одного из инструментов командной строки Graphviz, получим изображение в формате PNG. Для этого нужно выполнить следующую команду в любой оболочке UNIX:

```
$ dot -T png graph.dot -o graph.png
$ ls -l graph.png
-rw-r--r--@ 1 mtsouk staff 94862 Apr 24 16:48 graph.png
```

На рис. 11.3 показан графический файл, сгенерированный в результате выполнения этой команды:

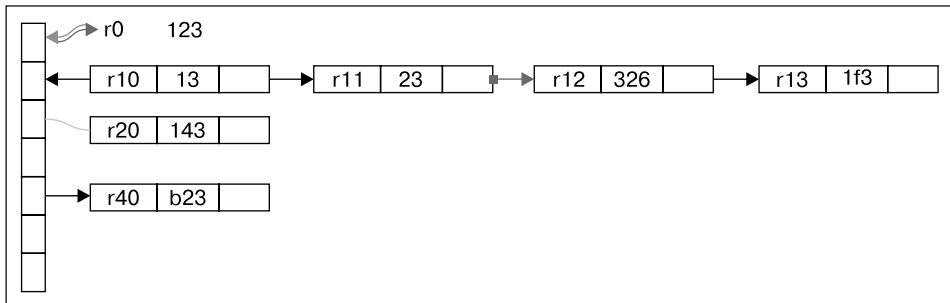


Рис. 11.3. Использование Graphviz для создания графов

Таким образом, если мы хотим визуализировать какую-либо структуру, определенно следует подумать об использовании Graphviz и его инструментов, особенно если нам нужно автоматизировать эти действия с помощью собственных сценариев.

Утилита go tool trace

Утилита `go tool trace` — это средство просмотра файлов трассировки, которые можно сгенерировать одним из трех способов:

- ☐ с помощью пакета `runtime/trace`;
- ☐ с помощью пакета `net/http/pprof`;
- ☐ с помощью команды `go test -trace`.

В этом разделе мы будем использовать только первый способ. Выходные данные следующей команды помогут вам понять, что делает *трассировщик выполнения программы Go*:

```
$ go doc runtime/trace
package trace // import "runtime/trace"
```

Package trace contains facilities for programs to generate traces for the Go execution tracer.

Tracing runtime activities

The execution trace captures a wide range of execution events such as goroutine creation/blocking/unblocking, syscall enter/exit/block, GC-related events, changes of heap size, processor start/stop, etc. A precise nanosecond-precision timestamp and a stack trace is captured for most events. The generated trace can be interpreted using ``go tool trace``. The trace tool computes the latency of a task by measuring the time between the task creation and the task end and provides latency distributions for each task type found in the trace.

```
func IsEnabled() bool
func Log(ctx context.Context, category, message string)
func Logf(ctx context.Context, category, format string, args
...interface{})
func Start(w io.Writer) error
func Stop()
func WithRegion(ctx context.Context, regionType string, fn func())
type Region struct{ ... }
func StartRegion(ctx context.Context, regionType string) *Region
type Task struct{ ... }
func NewTask(pctx context.Context, taskType string) (ctx context.Context, task *Task)
```

В главе 2 мы говорили о сборщике мусора Go и познакомились с Go-утилитой `gColl.go`, которая позволила увидеть некоторые переменные сборщика мусора Go. В этом разделе мы соберем еще больше информации о работе `gColl.go` с помощью утилиты `go tool trace`.

Сначала мы рассмотрим модифицированную версию программы `gColl.go`, которая дает команду Go собирать данные о производительности. Эта утилита хранится в файле `goGC.go`. Рассмотрим ее, разделив на три части.

Первая часть `goGC.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "runtime"
    "runtime/trace"
    "time"
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("-----")
}
```

Как мы уже знаем, чтобы начать сбор данных для утилиты `go tool trace`, сначала нужно импортировать стандартный Go-пакет `runtime/trace`.

Второй фрагмент кода `goGC.go` содержит следующий код Go:

```
func main() {
    f, err := os.Create("/tmp/traceFile.out")
    if err != nil {
        panic(err)
    }
    defer f.Close()
    err = trace.Start(f)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer trace.Stop()
```

Эта часть программы получает данные для утилиты `go tool trace` и не имеет ничего общего с функциональностью реальной программы. Сначала мы создаем новый файл, который будет содержать данные трассировки для утилиты `go tool trace`. Затем с помощью функции `trace.Start()` мы запускаем процесс трассировки. После того как она закончит работу, мы вызываем функцию `trace.Stop()`. Вызов этой функции с использованием ключевого слова `defer` означает, что мы хотим прекратить трассировку, когда программа завершит работу.



Использование утилиты `go tool trace` — двухэтапный процесс, который требует дополнительного кода Go. Сначала мы собираем данные, а затем отображаем и обрабатываем их.

Оставшийся код Go выглядит так:

```
var mem runtime.MemStats
printStats(mem)
for i := 0; i < 3; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
}
printStats(mem)
for i := 0; i < 5; i++ {
    s := make([]byte, 100000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(time.Millisecond)
}
printStats(mem)
}
```

Выполнение программы `goGC.go` приведет к следующим результатам, а также к созданию нового файла с именем `/tmp/traceFile.out`, в котором содержится информация о трассировке:

```
$ go run goGC.go
mem.Alloc: 108592
mem.TotalAlloc: 108592
mem.HeapAlloc: 108592
mem.NumGC: 0
-----
mem.Alloc: 109736
mem.TotalAlloc: 150127000
mem.HeapAlloc: 109736
mem.NumGC: 3
-----
mem.Alloc: 114672
mem.TotalAlloc: 650172952
mem.HeapAlloc: 114672
mem.NumGC: 8
-----
$ cd /tmp
$ ls -l traceFile.out
-rw-r--r-- 1 mtsouk wheel 10108 Apr 24 16:51 /tmp/traceFile.out
$ file /tmp/traceFile.out
/tmp/traceFile.out: data
```

Утилита `go tool trace` использует веб-интерфейс, который автоматически запускается при выполнении следующей команды:

```
$ go tool trace /tmp/traceFile.out
2019/04/24 16:52:06 Parsing trace...
2019/04/24 16:52:06 Splitting trace...
2019/04/24 16:52:06 Opening browser. Trace viewer is listening on
http://127.0.0.1:50383
```

На рис. 11.4 показан начальный вид веб-интерфейса утилиты `go tool trace` при просмотре файла трассировки `/tmp/traceFile.out`.

Теперь нужно выбрать ссылку `View trace`. Она приведет нас к виду веб-интерфейса утилиты `go tool trace`, использующей данные из `/tmp/traceFile.out`, который показан на рис. 11.5.

Как видно на рис. 11.5, сборщик мусора Go работает в собственной горутине, но он работает не постоянно. Кроме того, здесь мы видим количество горутин, задействованных программой. Чтобы узнать об этом больше, можно выбрать отдельные части интерактивного представления. Поскольку нас интересует сборщик мусора, то полезна информация о том, как часто и как долго работает сборщик мусора.

Обратите внимание, что, хотя `go tool trace` — очень удобная и мощная утилита, она не решает все проблемы производительности. Бывают случаи, когда уместнее использовать `go tool pprof`, особенно если вы хотите узнать, на что программа тратит большую часть своего времени, исследуя ее отдельные функции.

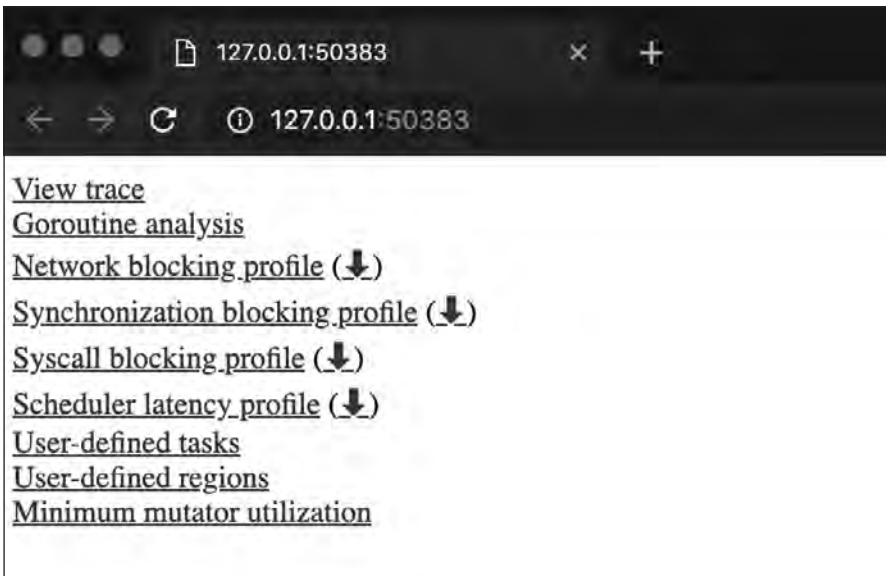


Рис. 11.4. Начальный вид веб-интерфейса утилиты go tool trace

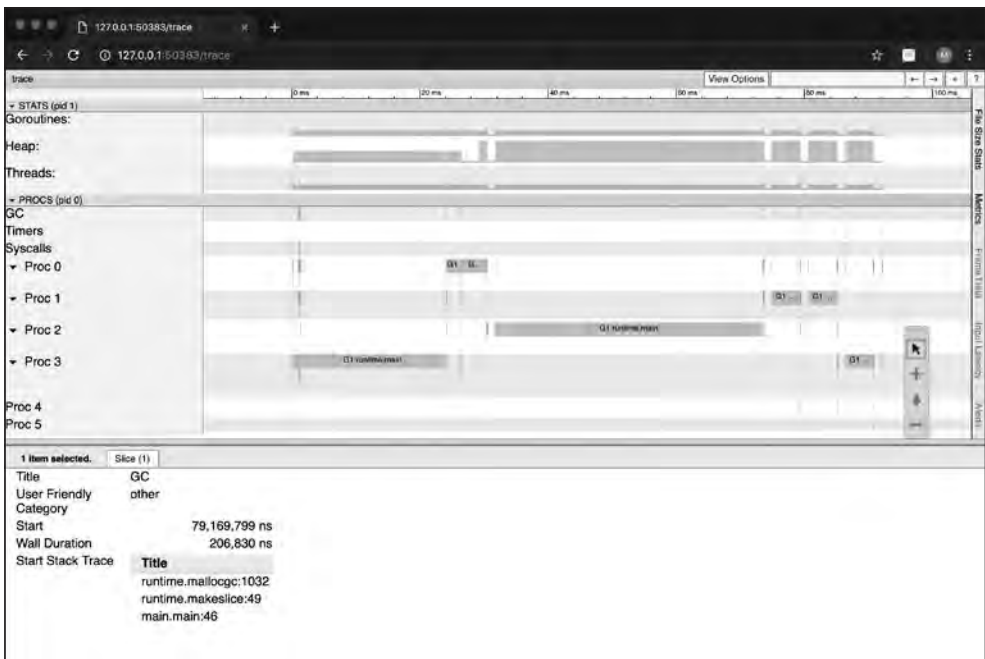


Рис. 11.5. Исследование работы сборщика мусора Go с использованием утилиты go tool trace

Тестирование кода Go

Тестирование программного обеспечения — очень обширная тема, ее невозможно охватить в одном разделе главы. Поэтому я постараюсь вкратце предоставить вам как можно больше практической информации.

Go позволяет писать тесты для кода, чтобы выявлять ошибки. Этот раздел посвящен *автоматизированному тестированию*, которое включает в себя написание дополнительного кода для проверки того, работает ли реальный, то есть промышленный, код так, как ожидается. Таким образом, результатом функции тестирования является одно из значений: либо PASS, либо FAIL. Как именно это работает, вы увидите в ближайшее время.

Несмотря на то что подход Go к тестированию поначалу может показаться простым, особенно если сравнить его с методами тестирования других языков программирования, он очень продуктивен и эффективен, поскольку не требует от разработчика слишком много времени.

В отношении тестирования Go следует определенным соглашениям. Прежде всего, функции тестирования должны содержаться в исходных файлах Go, имена которых заканчиваются на `_test.go`. Так, если у вас есть пакет с именем `aGoPackage.go`, то тесты должны находиться в файле с именем `aGoPackage_test.go`. Функция тестирования начинается с `Test` и проверяет правильность выполнения функции реального пакета. Наконец, чтобы подкоманда `go test` работала правильно, вам нужно импортировать стандартный Go-пакет `testing`. Как вы скоро увидите, это требование импорта также распространяется на два дополнительных случая.

После того как будет написан корректный код тестирования, подкоманда `go test` выполнит грязную работу за вас, включая сканирование всех файлов `*_test.go` для специальных функций, создание соответствующего временного пакета `main`, вызов специальных функций, получение результатов и генерирование окончательных результатов.



Всегда помещайте код тестирования в отдельный исходный файл. Нет необходимости создавать огромный исходный файл, который потом будет трудно читать и обслуживать.

Написание тестов для существующего кода Go

В этом подразделе показано, как писать тесты для существующего приложения Go, которое состоит из двух функций: одна из них вычисляет числа последовательности Фибоначчи, а вторая — определяет длину строки. Основной причиной использования этих двух функций, которые реализуют относительно тривиальные задачи, является простота. Сложность состоит в том, что у каждой функции будет две реализации: одна из них будет работать хорошо, а вторая — иметь некоторые проблемы.

Пакет Go, рассмотренный в этом примере, называется `testMe` и хранится в файле `testMe.go`. Мы рассмотрим код этого пакета, разделив его на три части.

Первая часть `testMe.go` содержит следующий код Go:

```
package testMe
```

```
func f1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 1
    }
    return f1(n-1) + f1(n-2)
}
```

В этом коде мы видим определение функции `f1()`, которая определяет натуральные числа последовательности Фибоначчи.

Вторая часть `testMe.go` содержит следующий код Go:

```
func f2(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 2
    }
    return f2(n-1) + f2(n-2)
}
```

В этом коде мы видим реализацию функции `f2()`, которая также определяет числа последовательности Фибоначчи. Однако эта функция содержит ошибку: она не возвращает 1, если значение `n` равно 1, что разрушает всю функциональность этой функции.

Остальной код `testMe.go` выглядит так:

```
func s1(s string) int {
    if s == "" {
        return 0
    }
    n := 1
    for range s {
        n++
    }
    return n
}

func s2(s string) int {
    return len(s)
}
```

Здесь мы реализовали две функции, `s1()` и `s2()`, которые работают со строками. Обе функции вычисляют длину строки. Однако реализация `s1()` неверна, поскольку начальное значение `n` равно 1, а не 0.

Теперь пора подумать о тестах и тестовых случаях. Прежде всего нам нужно создать файл `testMe_test.go` для хранения тестовых функций. Затем важно понимать, что мы не должны вносить какие-либо изменения в код `testMe.go`. Наконец, помните, что нам нужно стремиться написать столько тестов, сколько требуется, чтобы охватить все возможные сочетания входных и выходных данных.

Первая часть `testMe_test.go` содержит следующий код Go:

```
package testMe

import "testing"

func TestS1(t *testing.T) {
    if s1("123456789") != 9 {
        t.Error(`s1("123456789") != 9`)
    }
    if s1("") != 0 {
        t.Error(`s1("") != 0`)
    }
}
```

Эта функция выполняет два теста для функции `s1()`: один из них использует в качестве входных данных строку "123456789", а второй — строку "".

Вторая часть `testMe_test.go` выглядит так:

```
func TestS2(t *testing.T) {
    if s2("123456789") != 9 {
        t.Error(`s2("123456789") != 9`)
    }
    if s2("") != 0 {
        t.Error(`s2("") != 0`)
    }
}
```

Этот тестовый код выполняет те же два теста для функции `s2()`.

Остальной код `testMe_test.go`:

```
func TestF1(t *testing.T) {
    if f1(0) != 0 {
        t.Error(`f1(0) != 0`)
    }
    if f1(1) != 1 {
        t.Error(`f1(1) != 1`)
    }
    if f1(2) != 1 {
        t.Error(`f1(2) != 1`)
    }
}
```

```

    if f1(10) != 55 {
        t.Error(`f1(10) != 55`)
    }
}

func TestF2(t *testing.T) {
    if f2(0) != 0 {
        t.Error(`f2(0) != 0`)
    }
    if f2(1) != 1 {
        t.Error(`f2(1) != 1`)
    }
    if f2(2) != 1 {
        t.Error(`f2(2) != 1`)
    }
    if f2(10) != 55 {
        t.Error(`f2(10) != 55`)
    }
}

```

Этот код проверяет работу функций `f1()` и `f2()`.

Выполнение тестов приведет к результатам такого вида:

```

$ go test testMe.go testMe_test.go -v
=== RUN TestS1
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
=== RUN TestS2
--- PASS: TestS2 (0.00s)
=== RUN TestF1
--- PASS: TestF1 (0.00s)
=== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments  0.005s

```

Если не включить параметр `-v`, который дает расширенный вывод, то получим следующие результаты:

```

$ go test testMe.go testMe_test.go
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments  0.005s

```

Если вы хотите запустить тест несколько раз подряд, то можете использовать параметр `-count`:

```
$ go test testMe.go testMe_test.go -count 2
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments  0.005s
```

Если вы хотите выполнить только некоторые тесты, следует использовать параметр командной строки `-run`, значением которого является регулярное выражение. Тогда будут выполнены все тесты, для которых имя функции соответствует заданному регулярному выражению:

```
$ go test testMe.go testMe_test.go -run='F2' -v
=== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments  0.005s
$ go test testMe.go testMe_test.go -run='F1'
ok     command-line-arguments  (cached)
```

Последняя команда позволяет убедиться, что команда `go test` использовала кэширование.



Предупреждение. Тестирование программного обеспечения может показать только наличие одной или нескольких ошибок, но не отсутствие ошибок. Это означает, что вы никогда не можете быть абсолютно уверены, что в вашем коде нет ошибок!

Тестовое покрытие кода

В этом подразделе показано, как получить больше информации о тестовом покрытии кода ваших программ. Бывают случаи, когда тестовое покрытие кода помогает выявить проблемы и ошибки в коде, поэтому не стоит недооценивать его полезность.

В качестве примера мы рассмотрим программу `codeCover.go`, код Go которой выглядит так:

```
package codeCover
```

```
func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}

func fibo2(n int) int {
    if n >= 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

Реализация `fibo2()` содержит ошибку, потому что эта функция все время возвращает 0. В результате большая часть кода Go для `fibo2()` никогда не выполняется.

Тестовый код, который находится в файле `codeCover_test.go`, выглядит так:

```
package codeCover
```

```
import (
    "testing"
)

func TestFibo1(t *testing.T) {
    if fibo1(1) != 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}

func TestFibo2(t *testing.T) {
    if fibo2(0) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}

func TestFibo1_10(t *testing.T) {
    if fibo1(10) == 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}
```

```
func TestFibo2_10(t *testing.T) {
    if fibo2(10) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}
```

Реализации этих функций весьма примитивны, потому что их цель не генерация тестовых функций, а демонстрация использования инструмента покрытия кода.

Теперь мы проверим покрытие предыдущего кода. Основной способ сделать это — выполнить команду `go test` с параметром `-cover`:

```
$ go test -cover -v
Err:510    TestFibo1
--- PASS:  TestFibo1 (0.00s)
Err:510    TestFibo2
--- PASS:  TestFibo2 (0.00s)
Err:510    TestFibo1_10
--- PASS:  TestFibo1_10 (0.00s)
Err:510    TestFibo2_10
--- PASS:  TestFibo2_10 (0.00s)
PASS
coverage: 70.0% of statements
ok  _/Users/mtsouk/cover  0.005s
```

Как видим, охват кода составляет **70.0%**, а это означает, что где-то есть проблема. Обратите внимание, что для `-cover` не требуется флаг `-v`.

Однако есть вариант основной команды для определения тестового покрытия, которая генерирует профиль покрытия:

```
$ go test -coverprofile=coverage.out
PASS
coverage: 70.0% of statements
ok  _/Users/mtsouk/cover  0.005s
```

Создав специальный файл, мы можем проанализировать его следующим образом:

```
$ go tool cover -func=coverage.out
/Users/mtsouk/cover/codeCover.go:3: fibo1    100.0%
/Users/mtsouk/cover/codeCover.go:13: fibo2   40.0%
total: (statements) 70.0%
```

Кроме того, мы можем использовать для анализа этого файла тестового покрытия веб-браузер:

```
$ go tool cover -html=coverage.out
```

В окне браузера, которое откроется автоматически, вы увидите строки кода, окрашенные в зеленый или красный цвет. Те строки кода, которые выделены красным цветом, не охватываются тестами, а это означает, что нужно либо покрыть их большим количеством тестовых случаев, либо что-то исправить в тестируемом

коде Go. Так или иначе, следует внимательно присмотреться к этим строкам кода и выявить проблему.

Наконец, можно сохранить этот HTML-отчет:

```
$ go tool cover -html=coverage.out -o output.html
```

Тестирование HTTP-сервера с базой данных

В текущем разделе показано, как протестировать сервер базы данных — сервер *PostgreSQL*, который работает с HTTP-сервером, написанным на Go. Это особый случай, так как для того, чтобы проверить правильность работы HTTP-сервера, нам нужно получить данные из базы данных.

В этом разделе мы рассмотрим пример из двух файлов Go с именами `webServer.go` и `webServer_test.go`. Перед выполнением кода вам необходимо загрузить Go-пакет, позволяющий работать с PostgreSQL из Go. Для начала нужно выполнить следующую команду:

```
$ go get github.com/lib/pq
```

Код Go из `webServer.go` выглядит так:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}

func getData(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
```



```

connStr := "user=postgres dbname=s2 sslmode=disable"
db, err := sql.Open("postgres", connStr)
if err != nil {
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
    return
}

rows, err := db.Query("SELECT * FROM users")
if err != nil {
    fmt.Fprintf(w, "<h3 align=\"center\">%s</h3>\n", err)
    return
}
defer rows.Close()

for rows.Next() {
    var id int
    var firstName string
    var lastName string
    err = rows.Scan(&id, &firstName, &lastName)
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>\n", err)
        return
    }
    fmt.Fprintf(w, "<h3 align=\"center\">%d, %s, %s</h3>\n", id, firstName,
        lastName)
}

err = rows.Err()
if err != nil {
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
    return
}
}

func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
    fmt.Println("Using port number: ", PORT)
    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/getdata", getData)
    http.HandleFunc("/", myHandler)

    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

Код Go из `webServer_test.go` мы разделим на шесть частей. Особенность тестирования веб-сервера, который получает данные из базы данных, заключается в том, что нам потребуется написать много кода, для того чтобы протестировать базу данных.

Первая часть `webServer_test.go` выглядит так:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

Вторая часть `webServer_test.go` содержит следующий код Go:

```
func create_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
    }

    const query = `
        CREATE TABLE IF NOT EXISTS users (
            id SERIAL PRIMARY KEY,
            first_name TEXT,
            last_name TEXT
        )`

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}
```

Эта функция устанавливает соединение с PostgreSQL и создает в базе данных таблицу с именем `users`. Эта таблица будет использоваться исключительно для тестирования.

Третья часть `webServer_test.go` выглядит так:

```
func drop_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
```

```

        fmt.Println(err)
        return
    }

    _, err = db.Exec("DROP TABLE IF EXISTS users")
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

func insert_record(query string) {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

```

Это две вспомогательные функции. Первая из них удаляет таблицу, созданную функцией `create_table()`, а вторая вставляет запись в таблицу PostgreSQL.

Четвертая часть `webServer_test.go` выглядит так:

```

func Test_count(t *testing.T) {
    var count int
    create_table()

    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Epifanios',
    'Doe')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Mihalis',
    'Tsoukalos')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('Mihalis',
    'Unknown')")

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    row := db.QueryRow("SELECT COUNT(*) FROM users")
    err = row.Scan(&count)
}

```

```

db.Close()

if count != 3 {
    t.Errorf("Select query returned %d", count)
}
drop_table()
}

```

Это первая тестовая функция пакета, и она работает в два этапа: сначала вставляет три записи в таблицу базы данных; затем проверяет, что эта таблица базы данных содержит ровно три записи.

Пятая часть `webServer_test.go` содержит следующий код Go:

```

func Test_queryDB(t *testing.T) {
    create_table()

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    query := "INSERT INTO users (first_name, last_name) VALUES ('Random Text',
    '123456')"
    insert_record(query)

    rows, err := db.Query(`SELECT * FROM users WHERE last_name=$1`, `123456`)
    if err != nil {
        fmt.Println(err)
        return
    }
    var col1 int
    var col2 string
    var col3 string
    for rows.Next() {
        rows.Scan(&col1, &col2, &col3)
    }
    if col2 != "Random Text" {
        t.Errorf("first_name returned %s", col2)
    }

    if col3 != "123456" {
        t.Errorf("last_name returned %s", col3)
    }

    db.Close()
    drop_table()
}

```

Это еще одна тестовая функция, которая вставляет запись в таблицу базы данных и проверяет, что данные были записаны правильно.

Последняя часть `webServer_test.go` выглядит так:

```
func Test_record(t *testing.T) {
    create_table()
    insert_record("INSERT INTO users (first_name, last_name) VALUES ('John', 'Doe')")

    req, err := http.NewRequest("GET", "/getdata", nil)
    if err != nil {
        fmt.Println(err)
        return
    }
    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(getData)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("Handler returned %v", status)
    }

    if rr.Body.String() != "<h3 align=\"center\">1, John, Doe</h3>\n" {
        t.Errorf("Wrong server response!")
    }
    drop_table()
}
```

Последняя тестовая функция пакета взаимодействует с веб-сервером и посещает URL-адрес `/getdata`. Затем она проверяет, соответствует ли возвращаемое значение тому, что ожидается.

Теперь нам нужно создать базу данных PostgreSQL с именем `s2`, потому что именно такая база используется в тестовом коде. Это можно сделать следующим образом:

```
$ psql -p 5432 -h localhost -U postgres -c "CREATE DATABASE s2" CREATE DATABASE
```

Если все в порядке, то вы получите такой результат выполнения теста:

```
$ go test webServer* -v
=== RUN      Test_count
--- PASS:    Test_count (0.05s)
=== RUN      Test_queryDB
--- PASS:    Test_queryDB (0.04s)
=== RUN      Test_record
Serving: /getdata
Served:
--- PASS:    Test_record (0.04s)
PASS
ok   command-line-arguments 0.138s
```

Если опустить параметр `-v`, то получим более короткий вывод:

```
$ go test webServer*
ok      command-line-arguments 0.160s
```

Обратите внимание, что не следует запускать веб-сервер перед командой `go test`, так как он автоматически запускается командой `go test`.

Если сервер PostgreSQL не работает, то тестирование завершится неудачно и выведутся сообщения об ошибках:

```
$ go test webServer* -v
=== RUN      Test_count
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- FAIL:    Test_count (0.01s)
webServer_test.go:85: Select query returned 0
=== RUN      Test_queryDB
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- PASS:    Test_queryDB (0.00s)
=== RUN      Test_record
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
Serving: /getdata
Served:
dial tcp [::1]:5432: connect: connection refused
--- FAIL:    Test_record (0.00s)
webServer_test.go:145: Wrong server response!
FAIL
FAIL      command-line-arguments 0.024s
```

В главе 12 мы протестируем обработчики событий и код HTTP-сервера, написанного на Go.

Пакет `testing/quick`

В стандартную библиотеку Go входит пакет `testing/quick`, который можно использовать для *тестирования методом «черного ящика»*. В некоторой степени пакет `testing/quick` похож на пакет `QuickCheck` из языка программирования *Haskell* — оба пакета реализуют служебные функции, помогающие в тестировании методом «черного ящика». В Go можно генерировать случайные значения встроенных типов. Пакет `testing/quick` позволяет использовать эти случайные значения для тестирования, что избавляет от необходимости генерировать все значения самостоятельно.

Рассмотрим небольшую программу `randomBuiltin.go`, на примере которой покажем, как Go позволяет генерировать случайные значения:

```
package main

import (
    "fmt"
    "math/rand"
    "reflect"
    "testing/quick"
    "time"
)

func main() {
    type point3D struct {
        X, Y, Z int8
        S
        float32
    }

    ran := rand.New(rand.NewSource(time.Now().Unix()))
    myValues := reflect.TypeOf(point3D{})
    x, _ := quick.Value(myValues, ran)
    fmt.Println(x)
}
```

Сначала с помощью функции `rand.New()` мы создаем генератор случайных чисел, а затем используем *рефлексию*, чтобы получить информацию о типе `point3D`. После этого мы вызываем функцию `quick.Value()` из пакета `testing/quick` с дескриптором типа и генератором случайных чисел, чтобы поместить в переменную `myValues` некоторые случайные данные. Обратите внимание, что для генерации произвольных значений структур необходимо экспортировать все поля структуры.

Выполнение `randomBuiltin.go` приведет к результатам следующего вида:

```
$ go run randomBuiltin.go
{65 8 75 -3.3435536e+38}
$ go run randomBuiltin.go
{-38 33 36 3.2604468e+38}
```

Обратите внимание, что если вы решите создать случайные строки, то у вас получатся строки Unicode со странными символами.

Теперь вы умеете создавать случайные значения для встроенных типов. Продолжим знакомство с пакетом `testing/quick`. Для примера, рассматриваемого в этом подразделе, мы будем использовать два исходных файла Go с именами `quick.go` и `quick_test.go`.

Код Go в файле `quick.go` выглядит так:

```
package main

import (
```

```

    "fmt"
)

func Add(x, y uint16) uint16 {
    var i uint16
    for i = 0; i < x; i++ {
        y++
    }
    return y
}

func main() {
    fmt.Println(Add(0, 0))
}

```

В функции `add()` в `quick.go` реализовано суммирование целых чисел без знака (`uint16`) с использованием цикла `for`. Эта функция используется для иллюстрации тестирования методом «черного ящика» с помощью пакета `testing/quick`.

Код `quick_test.go` выглядит так:

```

package main

import (
    "testing"
    "testing/quick"
)

var N = 1000000

func TestWithSystem(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == (b + a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

func TestWithItself(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == Add(b, a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

```


Два вызова функции `quick.Check()` автоматически генерируют случайные числа на основе сигнатуры первого аргумента, который является функцией, определенной ранее. Создавать случайные входные числа самостоятельно не нужно, и это облегчает чтение и запись кода. Фактические тесты в обоих случаях выполняются в функции `condition`.

Выполнение тестов приведет к следующим результатам:

```
$ go test -v quick*
=== RUN    TestWithSystem
--- PASS:  TestWithSystem (8.36s)
=== RUN    TestWithItself
--- PASS:  TestWithItself (17.41s)
PASS
ok   command-line-arguments  (cached)
```

Если вы не хотите использовать кэшированное тестирование, можете выполнить команду `go test` следующим образом:

```
$ go test -v quick* -count=1
=== RUN    TestWithSystem
--- PASS:  TestWithSystem (8.15s)
=== RUN    TestWithItself
--- PASS:  TestWithItself (15.95s)
PASS
ok   command-line-arguments  24.104s
```



Характерный для Go способ обойти тестовое кэширование заключается в использовании параметра `-count = 1` в команде `go test`, поскольку, начиная с версии Go 1.12, `GOCACHE = off` больше не работает.

Если вы попытаетесь использовать `GOCACHE = off`, то получите сообщение об ошибке: `build cache is disabled by GOCACHE=off, but required as of Go 1.12`. Чтобы больше узнать об этом, воспользуйтесь командой `go help testflag`.

Как быть, если тестирование выполняется слишком долго или вообще не заканчивается

На тот случай, если инструменту `go test` требуется слишком много времени для завершения работы или по какой-либо причине он вообще ее не заканчивает, существует параметр `-timeout`. Чтобы проиллюстрировать его использование, мы создадим новую программу Go и тесты для нее.

Код Go пакета `main`, который хранится в файле `too_long.go`, выглядит так:

```
package main

import (
```

```

    "time"
)

func sleep_with_me() {
    time.Sleep(5 * time.Second)
}

func get_one() int {
    return 1
}

func get_two() int {
    return 2
}

func main() {
}

```

Функции тестирования, которые хранятся в файле `too_long_test.go`, выглядят так:

```

package main

import (
    "testing"
)

func Test_test_one(t *testing.T) {
    sleep_with_me()
    value := get_one()
    if value != 1 {
        t.Errorf("Function returned %v", value)
    }
    sleep_with_me()
}

func Test_test_two(t *testing.T) {
    sleep_with_me()
    value := get_two()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}

func Test_that_will_fail(t *testing.T) {
    value := get_one()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}

```

Тестовая функция `Test_that_will_fail()` всегда будет терпеть неудачу, тогда как две остальные функции работают правильно, но медленно.

Выполнение команды `go test` с параметром `-timeout` и без него приведет к следующим результатам:

```
$ go test too_long* -v
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
--- PASS: Test_test_two (5.00s)
=== RUN   Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL command-line-arguments 15.019s
$ go test too_long* -v -timeout 20s
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
--- PASS: Test_test_two (5.01s)
=== RUN   Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL command-line-arguments 15.021s
$ go test too_long* -v -timeout 15s
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
panic: test timed out after 15s
goroutine 34 [running]:
testing.(*M).startAlarm.func1()
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1334 +0xdf
created by time.goFunc
    /usr/local/Cellar/go/1.12.4/libexec/src/time/sleep.go:169 +0x44
goroutine 1 [chan receive]:
testing.(*T).Run(0xc0000dc000, 0x113a46d, 0xd, 0x1141a08, 0x1069b01)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:917 +0x381
testing.runTests.func1(0xc0000c0000)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1157 +0x78
testing.tRunner(0xc0000c0000, 0xc00009fe30)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
testing.runTests(0xc0000ba000, 0x1230280, 0x3, 0x3, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1155 +0x2a9
testing.(*M).Run(0xc0000a8000, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1072 +0x162
```

```

main.main()
  _testmain.go:46 +0x13e
goroutine 5 [runnable]:
runtime.goparkunlock(...)
  /usr/local/Cellar/go/1.12.4/libexec/src/runtime/proc.go:307
time.Sleep(0x12a05f200)
  /usr/local/Cellar/go/1.12.4/libexec/src/runtime/time.go:105 +0x159
command-line-arguments.sleep_with_me(...)
  /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/
too_long.go:8
command-line-arguments.Test_test_two(0xc0000dc000)
  /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-Edition/ch11/
too_long_test.go:17 +0x31
testing.tRunner(0xc0000dc000, 0x1141a08)
  /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
created by testing.(*T).Run
  /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:916 +0x35a
FAIL    command-line-arguments  15.015s

```

Как видим, сбой случился только при выполнении третьей команды `go test`, потому что для ее завершения требуется больше 15 секунд.

Бенчмаркинг кода Go

При бенчмаркинге измеряется производительность функции или программы, что позволяет сравнить разные реализации и понять, как именно изменения в коде влияют на производительность. Используя эту информацию, мы можем легко узнать, какую часть кода Go следует переписать, чтобы повысить производительность программы.



Никогда не тестируйте код Go на загруженной машине UNIX, которая в то же время используется для более важных задач, если только на то нет веских причин! В противном случае вы нарушите процесс сравнения и получите неточные результаты.

В Go приняты определенные соглашения относительно бенчмаркинга. Самым важным соглашением является то, что имя функции для бенчмаркинга должно начинаться с `Benchmark`.

За выполнение бенчмаркинга программы отвечает все та же подкоманда `go test`. Поэтому вам снова придется импортировать стандартный Go-пакет `testing` и включить функции бенчмаркинга в файлы с программами Go, имена которых заканчиваются на `_test.go`.

Простой пример бенчмаркинга

В этом разделе я покажу вам простейший пример бенчмаркинга. Мы измерим производительность трех алгоритмов, генерирующих числа последовательности Фибоначчи. К счастью, такие алгоритмы требуют большого количества математических вычислений, что делает их идеальными кандидатами для бенчмаркинга.

Для данного примера создадим новый пакет `main`, сохраним его в файле `benchmarkMe.go`. Мы рассмотрим его, разделив на три части.

Первая часть `benchmarkMe.go` выглядит так:

```
package main

import (
    "fmt"
)

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

В этом коде содержится реализация функции `fibo1()`, которая использует рекурсию для вычисления чисел последовательности Фибоначчи. Алгоритм работает нормально, однако он довольно примитивный и несколько медленный.

Второй фрагмент файла `benchmarkMe.go` содержит следующий код Go:

```
func fibo2(n int) int {
    if n == 0 || n == 1 {
        return n
    }
    return fibo2(n-1) + fibo2(n-2)
}
```

Здесь мы видим реализацию функции `fibo2()`, которая практически не отличается от уже знакомой нам функции `fibo1()`. Однако интересно посмотреть, как незначительное изменение кода — один оператор `if` вместо блока `if else if` — влияет на производительность функции.

В третьей части кода `benchmarkMe.go` содержится еще одна реализация функции, которая вычисляет числа последовательности Фибоначчи:

```
func fibo3(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

В представленной здесь функции `fibo3()` реализован совершенно иной подход с использованием отображения Go и цикла `for`. Неизвестно, действительно ли этот подход быстрее, чем две другие реализации. С алгоритмом, представленным в `fibo3()`, мы еще встретимся в главе 13, где изучим его работу более подробно. Как вы скоро убедитесь, выбор эффективного алгоритма способен избавить вас от множества хлопот!

Остальной код `benchmarkMe.go` выглядит так:

```
func main() {
    fmt.Println(fibo1(40))
    fmt.Println(fibo2(40))
    fmt.Println(fibo3(40))
}
```

Выполнение `benchmarkMe.go` приведет к следующим результатам:

```
$ go run benchmarkMe.go
102334155
102334155
102334155
```

Хорошие новости: все три реализации вернули одно и то же число. Теперь пора добавить в `benchmarkMe.go` несколько тестов, чтобы определить эффективность каждого из трех алгоритмов.

Согласно правилам Go, версия файла `benchmarkMe.go`, содержащая функции тестирования, должна быть сохранена под именем `benchmarkMe_test.go`. Мы разделим эту программу на пять частей. Первый фрагмент `benchmarkMe_test.go` содержит следующий код Go:

```
package main

import (
    "testing"
)
```

```
var result int

func benchmarkfibol(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibol(n)
    }
    result = r
}
```

В этом коде мы видим реализацию функции, имя которой начинается с `bench-`, а не с `Benchmark`. В результате эта функция не будет запускаться автоматически, поскольку ее имя начинается со строчной буквы `b` вместо прописной `B`.

Результат `fibol(n)` сначала сохраняется в переменной `r`, а потом используется другой глобальной переменной с именем `result` по одной хитрой причине: этот прием не позволяет компилятору выполнять любые оптимизации, которые бы отменили запуск функции, производительность которой мы хотим измерить, из-за того, что результаты этой функции никогда не используются. Тот же прием будет применен к функциям `benchmarkfibol2()` и `benchmarkfibol3()`, которые мы рассмотрим далее.

Вторая часть файла `benchmarkMe_test.go` содержит следующий код Go:

```
func benchmarkfibol2(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibol2(n)
    }
    result = r
}

func benchmarkfibol3(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibol3(n)
    }
    result = r
}
```

В этом коде определены еще две тестовые функции, которые не будут запускаться автоматически, поскольку их имена начинаются со строчной буквы `b`, а не с прописной `B`.

Теперь я открою вам большой секрет: даже если бы эти три функции назывались `BenchmarkFibol()`, `BenchmarkFibol2()` и `BenchmarkFibol3()`, они бы не вызывались автоматически командой `go test`, поскольку не соответствуют сигнатуре `func(*testing.B)`. Именно поэтому их имена начинаются со строчных букв `b`. Однако, как вы вскоре увидите, ничто не мешает нам вызывать эти функции из других функций бенчмаркинга.

Третья часть `benchmarkMe_test.go` выглядит так:

```
func Benchmark30fibo1(b *testing.B) {
    benchmarkfibo1(b, 30)
}
```

Это корректная тестовая функция с правильным именем и правильной сигнатурой, следовательно, она будет выполнена командой `go tool`.

Обратите внимание, что `Benchmark30fibo1()` является допустимым именем функции бенчмаркинга, а `BenchmarkfiboIII()` — нет, потому что после слова `Benchmark` не стоит заглавная буква или цифра. Это очень важно, так как функция бенчмаркинга с некорректным именем не будет запускаться автоматически. Это же правило применяется и к тестовым функциям.

Четвертый фрагмент `benchmarkMe_test.go` содержит следующий код Go:

```
func Benchmark30fibo2(b *testing.B) {
    benchmarkfibo2(b, 30)
}

func Benchmark30fibo3(b *testing.B) {
    benchmarkfibo3(b, 30)
}
```

Функции `Benchmark30fibo2()` и `Benchmark30fibo3()` аналогичны функции `Benchmark30fibo1()`.

Последняя часть файла `benchmarkMe_test.go` выглядит так:

```
func Benchmark50fibo1(b *testing.B) {
    benchmarkfibo1(b, 50)
}

func Benchmark50fibo2(b *testing.B) {
    benchmarkfibo2(b, 50)
}

func Benchmark50fibo3(b *testing.B) {
    benchmarkfibo3(b, 50)
}
```

Здесь мы видим еще три функции бенчмаркинга, которые вычисляют 50-е число в последовательности Фибоначчи.



Учтите, что каждый бенчмаркинг по умолчанию выполняется в течение не менее одной секунды. Если функция бенчмаркинга завершает работу меньше чем за одну секунду, то значение `b.N` увеличивается и функция запускается снова. В первый раз значение `b.N` равно 1, затем оно становится равным 2, 5, 10, 20, 50 и т. д. Так сделано потому, что чем быстрее функция, тем больше раз ее нужно запустить, чтобы получить точные результаты.

Выполнение `benchmarkMe_test.go` приведет к следующим результатам:

```
$ go test -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fib1-8      300      4494213 ns/op
Benchmark30fib2-8      300      4463607 ns/op
Benchmark30fib3-8     500000      2829 ns/op
Benchmark50fib1-8       1      67272089954 ns/op
Benchmark50fib2-8       1      67300080137 ns/op
Benchmark50fib3-8     300000      4138 ns/op
PASS
ok   command-line-arguments  145.827s
```

Здесь два важных момента: во-первых, значение параметра `-bench` определяет, какие именно функции бенчмаркинга будут выполняться. Использованное значение `.` — это регулярное выражение, которое соответствует всем допустимым функциям бенчмаркинга. Во-вторых, если опустить параметр `-bench`, то ни одна функция бенчмаркинга не будет выполнена.

Итак, о чем говорят эти результаты? Прежде всего, `-8` в конце каждой тестовой функции (`Benchmark10fib1-8`) означает количество горутин, использованных во время ее выполнения, что, по сути, является значением переменной среды `GOMAXPROCS`. Напомню, что мы обсуждали переменную среды `GOMAXPROCS` еще в главе 10. Аналогично вы можете увидеть здесь значения `GOOS` и `GOARCH`, которые соответствуют операционной системе и архитектуре вашей машины.

Во втором столбце результатов показано, сколько раз была выполнена соответствующая функция. Более быстрые функции выполняются чаще, чем медленные. Например, функция `Benchmark30fib3()` была выполнена 500 000 раз, а функция `Benchmark50fib2()` — только один раз! В третьем столбце отображается среднее время каждого выполнения функции.

Как видим, функции `fib1()` и `fib2()` действительно медленнее, чем функция `fib3()`. Если вы хотите включить в результаты бенчмаркинга статистику выделения памяти, то выполните следующую команду:

```
$ go test -benchmem -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fib1-8 300      4413791 ns/op      0 B/op      0 allocs/op
Benchmark30fib2-8 300      4430097 ns/op      0 B/op      0 allocs/op
Benchmark30fib3-8 500000    2774 ns/op        2236 B/op    6 allocs/op
Benchmark50fib1-8 1      71534648696 ns/op  0 B/op      0 allocs/op
Benchmark50fib2-8 1      72551120174 ns/op  0 B/op      0 allocs/op
Benchmark50fib3-8 300000    4612 ns/op        2481 B/op   10 allocs/op
PASS
ok   command-line-arguments  150.500s
```

Этот результат аналогичен результату бенчмаркинга без параметра командной строки `-benchmem`, но здесь появились еще два столбца. В четвертом столбце показано среднее количество памяти, которое выделялось при каждом выполнении функции тестирования. В пятом — количество операций выделения памяти, применяемых для выделения количества памяти, указанного в четвертом столбце. Так, для каждого вызова функции `Benchmark50fiboz()` выделялся в среднем 2481 байт за десять операций выделения.

Как видим, функции `fiboz1()` и `fiboz2()` не нуждаются в каком-либо особом виде памяти, кроме ожидаемого, потому что ни одна из них не использует какую-либо структуру данных, — в отличие от функции `fiboz3()`, в которой используется хеш-таблица. Отсюда и отличные от нуля значения в четвертом и пятом столбцах для `Benchmark10fiboz3-8`.

Неправильно определенные функции бенчмаркинга

Рассмотрим код Go следующей тестовой функции:

```
func BenchmarkFibozI(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fiboz1(i)
    }
}
```

Функция `BenchmarkFiboz()` имеет правильное имя и сигнатуру. Однако, к сожалению, эта функция бенчмаркинга некорректна, и после выполнения команды `go test` вы не получите от нее никакого результата.

Причина в том, что из-за использования цикла `for`, при увеличении значения `b.N` описанным выше способом, время выполнения этой функции бенчмаркинга также увеличивается. Это не позволяет функции `BenchmarkFibozI()` сойтись к какому-либо определенному числу, так что функция не завершается и, следовательно, не возвращает результат.

По аналогичной причине следующая функция бенчмаркинга также реализована неправильно:

```
func BenchmarkfibozII(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fiboz2(b.N)
    }
}
```

А в реализации следующих двух функций бенчмаркинга нет ничего плохого:

```
func BenchmarkFibozIV(b *testing.B) {
    for i := 0; i < b.N; i++ {
```

```

        _ = fibo3(10)
    }
}

func BenchmarkFiboIII(b *testing.B) {
    _ = fibo3(b.N)
}

```

Бенчмаркинг буферизованной записи

В этом разделе вы узнаете, как размер буфера записи влияет на производительность всей операции записи. Для этого мы рассмотрим код Go из файла `writingBU.go`. Разделим его на пять частей.

Программа `writingBU.go` генерирует фиктивные файлы со случайно сгенерированными данными. Аргументами программы являются размер буфера и размер выходного файла.

Первая часть `writingBU.go` выглядит так:

```

package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)

var BUFFERSIZE int
var FILESIZE int

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

```

Вторая часть файла `writingBU.go` содержит следующий код Go:

```

func createBuffer(buf *[]byte, count int) {
    *buf = make([]byte, count)
    if count == 0 {
        return
    }
    for i := 0; i < count; i++ {
        intByte := byte(random(0, 100))
        if len(*buf) > count {
            return
        }
        *buf = append(*buf, intByte)
    }
}

```

Третья часть `writingBU.go` содержит такой код Go:

```
func Create(dst string, b, f int) error {
    _, err := os.Stat(dst)
    if err == nil {
        return fmt.Errorf("File %s already exists.", dst)
    }
    destination, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer destination.Close()
    if err != nil {
        panic(err)
    }
    buf := make([]byte, 0)
    for {
        createBuffer(&buf, b)
        buf = buf[:b]
        if _, err := destination.Write(buf); err != nil {
            return err
        }
        if f < 0 {
            break
        }
        f = f - len(buf)
    }
    return err
}
```

Основную работу в программе выполняет функция `Create()`, и именно для нее необходимо выполнить бенчмаркинг.

Обратите внимание, что если бы размер буфера и размер файла не были частью сигнатуры функции `Create()`, то у нас возникла бы проблема при написании функции бенчмаркинга для `Create()`. Это потому, что было бы необходимо использовать глобальные переменные `BUFFERSIZE` и `FILESIZE`, которые инициализируются в функции `main()` программы `writingBU.go`. Сделать это в файле `writingBU_test.go` затруднительно. Следовательно, нужно думать о бенчмаркинге функции еще при написании кода самой этой функции.

Четвертый фрагмент кода `writingBU.go` выглядит так:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need BUFFERSIZE FILESIZE!")
        return
    }
    output := "/tmp/randomFile"
    BUFFERSIZE, _ = strconv.Atoi(os.Args[1])
    FILESIZE, _ = strconv.Atoi(os.Args[2])
}
```

```
err := Create(output, BUFFERSIZE, FILESIZE)
if err != nil {
    fmt.Println(err)
}
```

Остальной код Go программы `writingBU.go` выглядит так:

```
err = os.Remove(output)
if err != nil {
    fmt.Println(err)
}
}
```

Несмотря на то что вызов `os.Remove()`, который удаляет временный файл, находится внутри функции `main()`, которая не вызывается функциями бенчмаркинга, `os.Remove()` легко можно вызвать из функций бенчмаркинга, так что в этом нет проблемы.

Двойной запуск `writeBU.go` на компьютере с macOS Mojave и жестким диском SSD с использованием утилиты `time(1)` для проверки скорости выполнения программы привел к следующим результатам:

```
$ time go run writingBU.go 1 100000
real    0m1.193s
user    0m0.349s
sys     0m0.809s
$ time go run writingBU.go 10 100000
real    0m0.283s
user    0m0.195s
sys     0m0.228s
```

Итак, несмотря на очевидный факт, что размер буфера записи играет ключевую роль в производительности программы, мы должны быть более конкретными и точными. Поэтому мы напишем функции бенчмаркинга, которые сохраним в файле `writingBU_test.go`.

В первой части `writingBU_test.go` содержится следующий код Go:

```
package main

import (
    "fmt"
    "os"
    "testing"
)

var ERR error

func benchmarkCreate(b *testing.B, buffer, filesize int) {
    var err error
    for i := 0; i < b.N; i++ {
```

```

    err = Create("/tmp/random", buffer, filesize)
}
ERR = err
err = os.Remove("/tmp/random")
if err != nil {
    fmt.Println(err)
}
}

```

Как вы, вероятно, помните, такая функция бенчмаркинга не является допустимой.

Второй фрагмент кода `writeBU_test.go` выглядит так:

```

func Benchmark1Create(b *testing.B) {
    benchmarkCreate(b, 1, 1000000)
}

func Benchmark2Create(b *testing.B) {
    benchmarkCreate(b, 2, 1000000)
}

```

Остальной код `writingBU_test.go` следующий:

```

func Benchmark4Create(b *testing.B) {
    benchmarkCreate(b, 4, 1000000)
}

func Benchmark10Create(b *testing.B) {
    benchmarkCreate(b, 10, 1000000)
}

func Benchmark1000Create(b *testing.B) {
    benchmarkCreate(b, 1000, 1000000)
}

```

Здесь мы создали пять тестовых функций, которые будут проверять производительность функции `benchmarkCreate()`. Она, в свою очередь, проверяет производительность `Create()` для разного размера буфера записи.

Выполнение команды `go test` для файлов `writingBU.go` и `writingBU_test.go` приведет к результатам следующего вида:

```

$ go test -bench=. writingBU.go writingBU_test.go
goos: darwin
goarch: amd64
Benchmark1Create-8      1      6001864841 ns/op
Benchmark2Create-8      1     3063250578 ns/op
Benchmark4Create-8      1     1557464132 ns/op
Benchmark10Create-8    100000      11136 ns/op
Benchmark1000Create-8  200000       5532 ns/op
PASS
ok      command-line-arguments  21.847s

```

Следующая команда проверяет также выделение памяти для функций бенчмаркинга:

```
$ go test -bench=. writingBU.go writingBU_test.go -benchmem
goos: darwin
goarch: amd64
Benchmark1Create-8 1      6209493161 ns/op  16000840 B/op  2000017 allocs/op
Benchmark2Create-8 1      3177139645 ns/op   8000584 B/op  1000013 allocs/op
Benchmark4Create  1      1632772604 ns/op   4000424 B/op   500011 allocs/op
Benchmark10Create-8 100000  11238 ns/op       336 B/op        7 allocs/op
Benchmark1000Create-8 200000  5122 ns/op        303 B/op        5 allocs/op
PASS
ok   command-line-arguments  24.031s
```

Теперь пора интерпретировать результаты выполнения двух команд бенчмаркинга.

Очевидно, что использование буфера записи размером 1 байт неэффективно и только замедляет работу. Кроме того, такой размер буфера требует слишком много операций с памятью, что еще больше замедляет программу.

Использование буфера записи размером 2 байта ускоряет программу в два раза, и это хорошо. Однако это еще очень медленно. То же самое относится и к буферу записи размером 4 байта. Все сильно ускоряется, когда мы увеличиваем буфер записи до 10 байт. Наконец, результаты бенчмаркинга показывают, что использование буфера записи размером 1000 байт не ускоряет работу в 100 раз по сравнению с использованием буфера размером 10 байт. Следовательно, золотая середина между скоростью и размером буфера записи находится где-то между этими двумя значениями размера буфера.

Обнаружение недоступного кода Go

Код Go, который не может быть выполнен, является логической ошибкой, и поэтому разработчикам или компилятору Go в обычном режиме его довольно сложно найти. Проще говоря, в недоступном коде нет ничего плохого, кроме того, что этот код не может быть выполнен.

Рассмотрим следующий код Go, который хранится в файле `cannotReach.go`:

```
package main

import (
    "fmt"
)
```

```
func f1() int {
    fmt.Println("Entering f1()")
    return -10
    fmt.Println("Exiting f1()")
    return -1
}

func f2() int {
    if true {
        return 10
    }
    fmt.Println("Exiting f2()")
    return 0
}

func main() {
    fmt.Println(f1())
    fmt.Println("Exiting program...")
}
```

В коде Go программы `cannotReach.go` нет ничего синтаксически некорректного. Мы можем выполнить `cannotReach.go`, не получив от компилятора вывода об ошибках:

```
$ go run cannotReach.go
Entering f1()
-1
Exiting program...
```

Однако обратите внимание, что функция `f2()` в программе никогда не используется. Очевидно, что следующий код Go функции `f2()` никогда не выполняется, потому что условие в предыдущем операторе `if` всегда истинно:

```
fmt.Println("Exiting f2()")
return 0
```

Что же можно сделать? Мы можем выполнить `go vet`:

```
$ go vet cannotReach.go
# command-line-arguments
./cannotReach.go:10:2: unreachable code
```

Результат говорит о том, что в строке 10 программы есть недоступный код. Теперь удалим из функции `f1()` инструкцию `return -10` и снова запустим `go vet`:

```
$ go vet cannotReach.go
```

Теперь сообщений об ошибках нет, несмотря на то что в функции `f2()` все еще остается недоступный код. Это означает, что `go vet` не фиксирует все возможные типы логических ошибок.

Кросс-компиляция

Кросс-компиляция — это процесс создания двоичного исполняемого файла для архитектуры, отличной от той, на которой вы работаете.

Главное преимущество, которое дает кросс-компиляция, заключается в том, что нам не требуется второй или третий компьютер, чтобы создавать исполняемые файлы для разных архитектур. В целом достаточно всего одной машины для разработки. К счастью, в Go есть встроенная поддержка кросс-компиляции.

В этом разделе для иллюстрации процесса кросс-компиляции мы рассмотрим пример кода Go, который находится в файле `xCompile.go`. Этот код выглядит так:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("with Go version", runtime.Version())
}
```

Выполнение `xCompile.go` на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.12.4
```

Для кросс-компиляции исходного файла Go необходимо присвоить переменным среды `GOOS` и `GOARCH` значения, соответствующие той операционной системе и архитектуре, для которых компилируется программа. Это не так сложно, как кажется.

Итак, процесс кросс-компиляции выглядит так:

```
$ env GOOS=linux GOARCH=arm go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
$ ./xCompile
-bash: ./xCompile: cannot execute binary file
```

Первая команда генерирует двоичный файл, который работает на машинах с Linux и архитектурой ARM. Результат выполнения команды `file(1)` подтверждает, что сгенерированный двоичный файл действительно предназначен для другой архитектуры.

Поскольку компьютер с Debian Linux, который используется в этом примере, имеет процессор Intel, нам придется еще раз выполнить команду `go build` со значением `GOARCH`:

```
$ env GOOS=linux GOARCH=386 go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, with debug_info, not stripped
```

Выполнение сгенерированного двоичного исполняемого файла на компьютере с Linux даст следующий ожидаемый результат:

```
$ ./xCompile
You are using gc on a 386 machine
with Go version go1.12.4
$ go version
go version go1.3.3 linux/amd64
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.3.3
```

Следует отметить, что двоичный файл, полученный в результате кросс-компиляции `xCompile.go`, выводит версию Go, примененную для его компиляции. Второе, на что следует обратить внимание, — архитектура машины с Linux — это на самом деле `amd64`, а не `386`, которая использовалась при кросс-компиляции.



Список доступных значений переменных среды `GOOS` и `GOARCH` размещен по адресу <https://golang.org/doc/install/source>. Однако имейте в виду, что не все пары `GOOS` и `GOARCH` актуальны.

Создание примеров функций

Частью процесса документирования является создание примеров кода, которые демонстрируют использование некоторых или всех функций и типов, существующих в пакете. Примеры функций имеют много преимуществ, в том числе они являются готовыми тестами, которые можно выполнить с помощью команды `go test`. Таким образом, если функция примера содержит строку `// Output:`, то инструмент `go test` проверит, соответствует ли вычисленный результат значениям, найденным после строки `// Output:`.

Кроме того, примеры, размещенные в документации к пакету, действительно полезны. Подробнее документация рассмотрена в следующем разделе. Наконец, функции примеров, представленные на сервере документации Go (https://golang.org/pkg/io/#example_Copy), позволяют экспериментировать с примерами кода. Интерактивная среда Go на сайте <https://play.golang.org/> также поддерживает эту функцию.

Поскольку за примеры программы отвечает подкоманда `go test`, необходимо импортировать стандартный Go-пакет `testing` и включить примеры функций в файлы Go, которые заканчиваются на `_test.go`. Кроме того, имя каждой функции-примера должно начинаться со слова `Example`. Наконец, функции примеров не принимают входных аргументов и не возвращают результатов.

Теперь создадим несколько функций примеров для следующего пакета, который сохраним в файле `ex.go`:

```
package ex

func F1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 || n == 2 {
        return 1
    }
    return F1(n-1) + F1(n-2)
}

func S1(s string) int {
    return len(s)
}
```

Исходный файл `ex.go` содержит реализацию двух функций с именами `F1()` и `S1()`.

Обратите внимание, что в программу `ex.go` не нужно импортировать пакет `fmt`.

Как вы уже знаете, функции примеров должны быть включены в файл `ex_test.go`. Разделим его на три части.

Первая часть `ex_test.go` выглядит так:

```
package ex

import (
    "fmt"
)
```

Вторая часть кода `ex_test.go` содержит следующий код Go:

```
func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 55
    // 1
}
```

Остальной код Go файла `ex_test.go`:

```
func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 8
    // 0
}
```

Выполнение команды `go test` для пакета `ex.go` приведет к результатам следующего вида:

```
$ go test ex.go ex_test.go -v
=== RUN ExampleF1
--- PASS: ExampleF1 (0.00s)
=== RUN ExampleS1
--- FAIL: ExampleS1 (0.00s)
got:
9
0
want:
8
0
FAIL
FAIL    command-line-arguments    0.006s
```

Как видим, судя по данным, которые выводятся после комментария `// Output:`, здесь что-то не так с функцией `S1()`.

От кода Go до машинного кода

В этом разделе подробно показано, как код Go преобразуется в машинный код. В качестве примера мы рассмотрим программу Go, которая называется `machineCode.go`. Разделим ее на две части.

Первая часть `machineCode.go` выглядит так:

```
package main

import (
    "fmt"
)

func hello() {
    fmt.Println("Hello!")
}
```

Вторая и последняя часть `machineCode.go`:

```
func main() {
    hello()
}
```

Теперь посмотрим, как выглядит программа `machineCode.go`, переведенная в машинный код:

```
$ GOSSAFUNC=main GOOS=linux GOARCH=amd64 go build -gcflags "-S"
machineCode.go
# runtime
dumped SSA to /usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html
# command-line-arguments
dumped SSA to ./ssa.html
os.(*File).close STEXT dupok nosplit size=26 args=0x18 locals=0x0
    0x0000 00000 (<autogenerated>:1) TEXT os.(*File).close(SB),
DUPOK|NOSPLIT|ABIInternal, $0-24
    0x0000 00000 (<autogenerated>:1) FUNCDATA $0, gcllocals
e6397a44f8e1b6e77d0f200b4fba5269(SB)
    0x0000 00000 (<autogenerated>:1) FUNCDATA $1, gcllocals
69c1753bd5f81501d95132d08af04464(SB)
    0x0000 00000 (<autogenerated>:1) FUNCDATA $3, gcllocals
9fb7f0986f647f17cb5
...
```

Судя по первым строкам, есть два файла, которые содержат все сгенерированные выходные данные: `/usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html` и `./ssa.html`. Стоит отметить, что если у вас установлена другая версия Go или другой комплект установки Go, то первый файл будет расположен в другом месте. Форма *статического одиночного присваивания* (Static Single Assignment, SSA) — это метод описания низкоуровневых операций, который очень близок к машинным инструкциям. Однако заметьте, что SSA, в отличие от машинного кода, действует так, как будто у него бесконечное количество регистров.

Открыв этот файл в любом браузере, вы увидите много полезной, но низкоуровневой информации о программе. Наконец, обратите внимание, что значением параметра `GOSSAFUNC` является функция Go, которую вы хотите *дизассемблировать*.

К сожалению, более подробное обсуждение SSA выходит за рамки данной книги.

Использование ассемблера в Go

В этом подразделе речь пойдет об *ассемблере* и Go, а также о том, как использовать ассемблер для реализации функций Go. Для начала мы рассмотрим следующую программу Go, которая хранится в файле `add_me.go`:

```
package main

import (
```

```

    "fmt"
)

func add(x, y int64) int64 {
    return x + y
}

func main() {
    fmt.Println(add(1, 2))
}

```

Выполнив следующую команду, мы увидим ассемблерную реализацию функции `add()`:

```

$ GOOS=darwin GOARCH=amd64 go tool compile -S add_me.go
"".add STEXT nosplit size=19 args=0x18 locals=0x0
0x0000 00000 (add_me.go:7) TEXT      "".add(SB), NOSPLIT|ABIInternal, $0-24
0x0000 00000 (add_me.go:7) FUNCDATA  $0, gcllocals
                                33cdeccccebe80329f1fdbee7f5874cb(SB)
0x0000 00000 (add_me.go:7) FUNCDATA  $1, gcllocals
                                33cdeccccebe80329f1fdbee7f5874cb(SB)
0x0000 00000 (add_me.go:7) FUNCDATA  $3, gcllocals
                                33cdeccccebe80329f1fdbee7f5874cb(SB)
0x0000 00000 (add_me.go:8) PCDATA    $2, $0
0x0000 00000 (add_me.go:8) PCDATA    $0, $0
0x0000 00000 (add_me.go:8) MOVQ      "".y+16(SP), AX
0x0005 00005 (add_me.go:8) MOVQ      "".x+8(SP), CX
0x000a 00010 (add_me.go:8) ADDQ       CX, AX
0x000d 00013 (add_me.go:8) MOVQ      AX, ""..~r2+24(SP)
0x0012 00018 (add_me.go:8) RET
0x0000 48 8b 44 24 10 48 8b 4c 24 08 48 01 c8 48 89 44
                                H.D$.H.L$.H..H.D
0x0010 24 18 c3                  $..
"".main STEXT size=150 args=0x0 locals=0x58

```

Последняя строка не является частью ассемблерной реализации функции `add()`. Кроме того, строки `FUNCDATA` также не имеют ничего общего с ассемблерной реализацией функции, они добавляются компилятором Go.

Теперь внесем некоторые изменения в представленный выше ассемблерный код, чтобы он выглядел следующим образом:

```

TEXT    add(SB), $0
    MOVQ x+0(FP), BX
    MOVQ y+8(FP), BP
    ADDQ BP, BX
    MOVQ BX, ret+16(FP)
    RET

```

Этот ассемблерный код мы сохраним в файле `add_amd64.s`, чтобы использовать в качестве реализации функции `add()`.

Новая версия `add_me.go` будет выглядеть так:

```
package main

import (
    "fmt"
)

func add(x, y int64) int64

func main() {
    fmt.Println(add(1, 2))
}
```

Это означает, что `add_me.go` будет использовать ассемблерную реализацию функции `add()`. Применять ассемблерную реализацию функции `add()` очень просто:

```
$ go build
$ ls -l
total 4136
-rw-r--r--@ 1 mtsouk staff 93 Apr 18 22:49 add_amd64.s
-rw-r--r--@ 1 mtsouk staff 101 Apr 18 22:59 add_me.go
-rwxr-xr-x 1 mtsouk staff 2108072 Apr 18 23:00 asm
$ ./asm
3
$ file asm
asm: Mach-O 64-bit executable x86_64
```

Единственная сложность состоит в том, что ассемблерный код не является переносимым. К сожалению, более подробное обсуждение использования ассемблера в Go выходит за рамки данной книги — советую изучить ассемблер и архитектуру вашего процессора.

Генерация документации

Go предоставляет инструмент `godoc`, который позволяет просматривать документацию пакетов — при условии, что в файлы пакетов включена дополнительная информация.



Общая рекомендация: следует стремиться документировать все, за исключением очевидного. Другими словами, не говорите: «Здесь я создаю новую переменную типа `int`». Лучше сообщите о назначении этой переменной! Однако по-настоящему хороший код обычно не нуждается в документации.

Правило написания документации в Go довольно простое и понятное: чтобы задокументировать какой-либо объект, нужно поместить непосредственно перед

его объявлением одну или несколько обычных строк комментариев, которые начинаются с символов `//`. Это соглашение применяется для документирования функций, переменных, констант и даже пакетов.

Кроме того, вы скоро заметите, что первая строка документации пакета, независимо от его размера, появляется в списке пакетов `godoc`, как это сделано на сайте <https://golang.org/pkg/>. Поэтому описание должно быть довольно емким и полным.

Помните, что комментарии, начинающиеся со слова `BUG` (что-то там), появятся в разделе `Bugs` документации пакета, даже если они не предшествуют объявлению объекта.

Если вам нужен пример, посмотрите на исходный код и страницу документации пакета `bytes`, которые вы найдете по адресам <https://golang.org/src/bytes/bytes.go> и <https://golang.org/pkg/bytes/> соответственно. Наконец, все комментарии, не связанные с объявлением верхнего уровня, в выводе, генерируемом утилитой `godoc`, опускаются.

Рассмотрим следующий код Go, который хранится в файле `documentMe.go`:

```
// This package is for showcasing the documentation capabilities of Go
// It is a naive package!
package documentMe
// Pie is a global variable
// This is a silly comment!
const Pie = 3.1415912
// The S1() function finds the length of a string
// It iterates over the string using range
func S1(s string) int {
    if s == "" {
        return 0
    }
    n := 0
    for range s {
        n++
    }
    return n
}
// The F1() function returns the double value of its input integer
// A better function name would have been Double()!
func F1(n int) int {
    return 2 * n
}
```

Как сказано в предыдущем разделе, для создания примеров к этим функциям нам нужно создать файл `documentMe_test.go`. Содержимое `documentMe_test.go` выглядит так:

```
package documentMe

import (
```



```

    "fmt"
)

func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 9
    // 0
}

func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 1
    // 55
}

```

Чтобы просмотреть документацию к файлу `documentMe.go`, как говорилось в главе 6, нам необходимо установить этот пакет на свой компьютер. Для этого нужно выполнить следующие команды из оболочки UNIX:

```

$ mkdir ~/go/src/documentMe
$ cp documentMe* ~/go/src/documentMe/
$ ls -l ~/go/src/documentMe/
total 16
-rw-r--r--@ 1 mtsouk staff 542 Mar 6 21:11 documentMe.go
-rw-r--r--@ 1 mtsouk staff 223 Mar 6 21:11 documentMe_test.go
$ go install documentMe
$ cd ~/go/pkg/darwin_amd64
$ ls -l documentMe.a
-rw-r--r-- 1 mtsouk staff 1626 Mar 6 21:11 documentMe.a

```

А затем следует выполнить утилиту `godoc`:

```
$ godoc -http=":8080"
```

Обратите внимание, что если этот порт уже используется и пользователь имеет права `root`, то вы получите следующее сообщение об ошибке:

```
$ godoc -http=":22"
2019/08/19 15:18:21 ListenAndServe :22: listen tcp :22: bind: address already in use
```

Однако если у пользователя нет привилегий `root`, то сообщение об ошибке будет следующим (даже если данный порт уже используется):

```
$ godoc -http=":22"
2019/03/06 21:03:05 ListenAndServe :22: listen tcp :22: bind: permission denied
```

Когда все будет готово, вы сможете просматривать созданную HTML-документацию в любом браузере. Для этого нужно открыть URL `http://localhost:8080/pkg/`.

На рис. 11.6 показан корневой каталог сервера `godoc`, о котором мы только что говорили. Здесь, наряду с другими пакетами Go, присутствует пакет `documentMe`, который мы создали в файле `documentMe.go`.

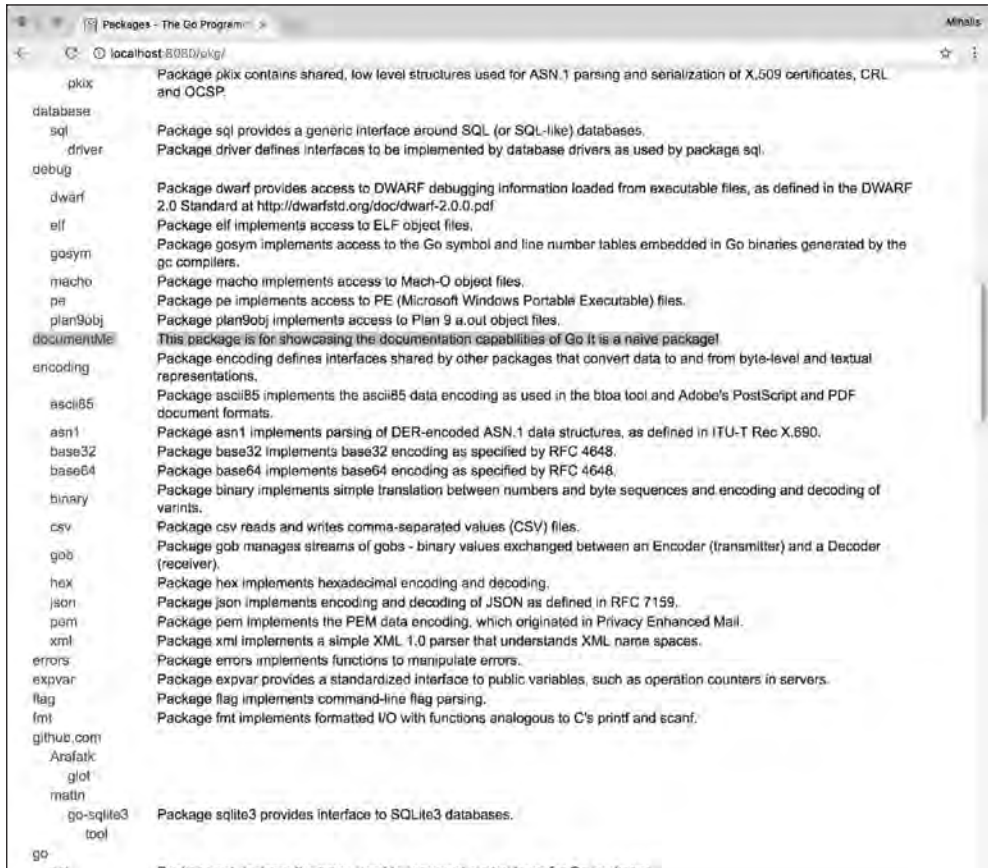


Рис. 11.6. Корневой каталог сервера `godoc`

На рис. 11.7 показан корневой каталог документации пакета `documentMe`, реализованного в исходном файле `documentMe.go`.

Аналогично на рис. 11.8 более подробно показана документация функции `S1()` пакета `documentMe.go`, которая также включает в себя пример кода.

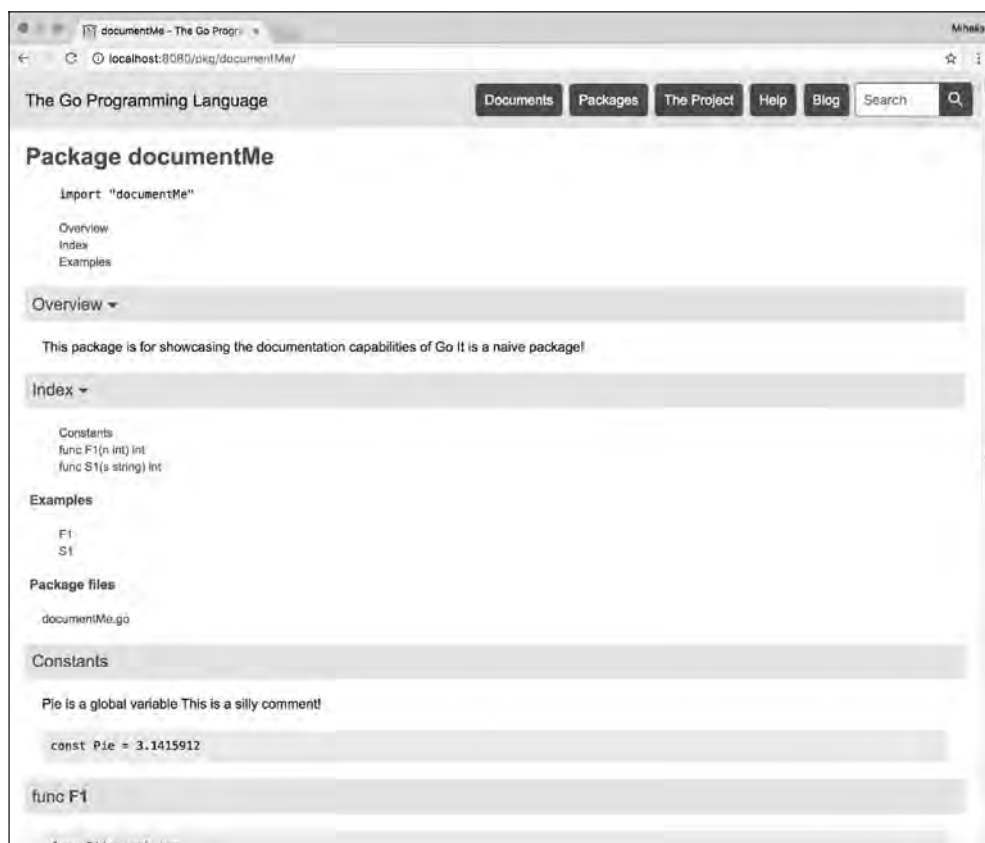


Рис. 11.7. Начальная страница файла documentMe.go

Выполнение команды `go test` приведет к следующим результатам, которые позволяют выявить потенциальные проблемы и ошибки в коде:

```
$ go test -v documentMe*
=== RUN ExampleS1
--- PASS: ExampleS1 (0.00s)
=== RUN ExampleF1
--- FAIL: ExampleF1 (0.00s)
got:
20
4
want:
1
55
FAIL
FAIL    command-line-arguments  0.005s
```



Рис. 11.8. Страница документации и пример функции S1()

Использование образов Docker

В этом разделе вы узнаете, как создать образ Docker, содержащий базу данных PostgreSQL и код Go. Этот образ не будет идеальным, но на его примере вы увидите, как это делается. Как вы уже знаете, все должно начинаться с Dockerfile, который в данном случае будет следующим:

```
FROM ubuntu:18.04
```

```
RUN apt-get update && apt-get install -y gnupg
```

```
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-  
keys B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
```

```
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" >  
/etc/apt/sources.list.d/pgdg.list
```

```

RUN apt-get update && apt-get install -y software-properties-common
postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3
RUN apt-get update && apt-get install -y git golang vim

USER postgres
RUN /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';" &&\
    createdb -O docker docker

RUN echo "host all all 0.0.0.0/0 md5" >>
/etc/postgresql/9.3/main/pg_hba.conf
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

USER root
RUN mkdir files
COPY webServer.go files
WORKDIR files
RUN go get github.com/lib/pq
RUN go build webServer.go
RUN ls -l
USER postgres
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D",
"/var/lib/postgresql/9.3/main", "-c",
"config_file=/etc/postgresql/9.3/main/postgresql.conf"]

```

Это довольно сложный Dockerfile. Он начинается с базового образа Docker из Docker Hub и загружает дополнительное программное обеспечение. Этого более или менее достаточно, чтобы создать настраиваемый образ Docker, который соответствует вашим потребностям.

Выполнение Dockerfile сгенерирует много выходных данных, включая следующие:

```

$ docker build -t go_postgres .
Sending build context to Docker daemon 9.216kB
Step 1/19 : FROM ubuntu:18.04
---> 94e814e2efa8
Step 2/19 : RUN apt-get update && apt-get install -y gnupg
...
Step 15/19 : RUN go get github.com/lib/pq
---> Running in 17aede1c97d8
Removing intermediate container 17aede1c97d8
---> 1878408c6e06
Step 16/19 : RUN go build webServer.go
---> Running in 39cfe8af63d5
Removing intermediate container 39cfe8af63d5
---> 1b4d638242ae
...

```

```
Removing intermediate container 9af6a391c1e4
--> c24937079367
Successfully built c24937079367
Successfully tagged go_postgres:latest
```

В последней части Dockerfile мы также соберем исполняемый файл из исходной программы webServer.go, используя операционную систему образа Docker, в данном случае Ubuntu Linux.

Выполнение команды `docker images` приведет к результатам следующего вида:

```
$ docker images
REPOSITORY      TAG          IMAGE ID          CREATED          SIZE
go_postgres     latest      c24937079367     34 seconds ago  831MB
```

Теперь мы можем выполнить этот образ Docker и подключиться к его оболочке `bash(1)`:

```
$ docker run --name=my_go -dt go_postgres:latest
23de1ab0d3f5517d5dbf8c599a68075574a8ed9217aa3cb4899ea2f92412a833
$ docker exec -it my_go bash
postgres@23de1ab0d3f5:/files$
```

Дополнительные ресурсы

Советую посетить следующие веб-ссылки:

- ☐ посетите веб-сайт Graphviz, расположенный по адресу <http://graphviz.org>;
- ☐ посетите страницу документации пакета `testing`, расположенную по адресу <https://golang.org/pkg/testing/>;
- ☐ если вы не знаете о Дональде Кнуте (Donald Knuth) и его работах, советую узнать о нем больше на странице https://en.wikipedia.org/wiki/Donald_Knuth;
- ☐ чтобы ближе познакомиться с утилитой `godoc`, посетите ее страницу документации <https://godoc.org/golang.org/x/tools/cmd/godoc>;
- ☐ посетите страницу документации стандартного Go-пакета `runtime/pprof` по адресу <https://golang.org/pkg/runtime/pprof/>;
- ☐ просмотрите код Go пакета `net/http/pprof`, посетив его страницу <https://golang.org/src/net/http/pprof/pprof.go>;
- ☐ страницу документации пакета `net/http/pprof` вы найдете по адресу <https://golang.org/pkg/net/http/pprof/>;
- ☐ чтобы больше узнать об инструменте `pprof`, посетите страницу разработки <https://github.com/google/pprof>;

- ❑ посмотрите видеоролик *Advanced Testing with Go*, размещенный на GopherCon 2017 Митчеллом Хашимото (Mitchell Hashimoto): <https://www.youtube.com/watch?v=8hQG7QlcLBk>;
- ❑ почитайте исходный код пакета `testing` по адресу <https://golang.org/src/testing/testing.go>;
- ❑ дополнительную информацию о пакете `testing/quick` вы найдете на странице <https://golang.org/pkg/testing/quick/>;
- ❑ чтобы больше узнать о пакете `profile`, посетите веб-страницу <https://github.com/pkg/profile>;
- ❑ *Manual for the Plan 9 assembler*: <https://9p.io/sys/doc/asm.html>;
- ❑ страницу документации пакета `arm64` вы найдете по адресу <https://golang.org/pkg/cmd/internal/obj/arm64/>;
- ❑ чтобы больше узнать о Go и SSA, посетите страницу <https://golang.org/src/cmd/compile/internal/ssa/>;
- ❑ библиография SSA: <http://www.dcs.gla.ac.uk/~jsinger/ssa.html>;
- ❑ чтобы ближе познакомиться с инструментом `go fix`, посетите его веб-страницу <https://golang.org/cmd/fix/>.

Упражнения

- ❑ Напишите тестовые функции для программы `byWord.go`, которую мы разработали в главе 8.
- ❑ Напишите функции бенчмаркинга для программы `readSize.go`, которую мы разработали в главе 8.
- ❑ Попробуйте исправить проблемы в Go-программах `documentMe.go` и `documentMe_test.go`.
- ❑ Используя текстовый интерфейс утилиты `go tool pprof`, просмотрите файл `memoryProfile.out`, созданный программой `profileMe.go`.
- ❑ Измените `webServer.go` и `webServer_test.go` так, чтобы они позволяли работать с другими базами данных, такими как MySQL и SQLite3.
- ❑ Измените файл `Dockerfile` из последнего раздела таким образом, чтобы включить в него `webServer_test.go`.
- ❑ Измените `Dockerfile` таким образом, чтобы можно было выполнить `go test -v` для веб-сервера. Изменения должны позволять корректно создавать пользователей и базы данных PostgreSQL.
- ❑ Используя веб-интерфейс утилиты `go tool pprof`, проверьте файл `memoryProfile.out`, созданный с помощью `profileMe.go`.

Резюме

В этой главе мы рассмотрели тестирование, оптимизацию и профилирование кода. В конце главы вы узнали, как найти в программе недоступный код и как кросс-компилировать код Go, как использовать команду `go test` для тестирования и бенчмаркинга кода Go, а также для составления дополнительной документации с использованием примеров функций.

Несмотря на то что здесь представлено далеко не полное описание профилировщика Go и команды `go tool trace`, следует понимать, что при изучении таких тем, как профилирование и трассировка кода, ничто не заменит самостоятельные эксперименты и испытания новых методов!

В следующей главе рассмотрено сетевое программирование на Go, то есть программирование приложений, работающих в компьютерных сетях TCP/IP, в том числе в Интернете. Следующая глава посвящена таким темам, как пакет `net/http`, создание веб-клиентов и веб-серверов на Go, структуры `http.Response` и `http.Request`, профилирование HTTP-серверов, gRPC и превышение времени ожидания сетевых подключений.

Кроме того, в главе 12 рассмотрены протоколы *IPv4* и *IPv6*, а также инструменты Wireshark и tshark, которые применяются для перехвата и анализа сетевого трафика.

12 Основы сетевого программирования на Go

В предыдущей главе речь шла о бенчмаркинге кода Go с использованием функций бенчмаркинга, о тестировании в Go, функциях примеров, покрытии кода, кросс-компиляции и профилировании кода Go, а также о создании документации в Go и образов Docker, содержащих требуемое программное обеспечение.

Эта глава посвящена сетевому и веб-программированию. Вы узнаете, как создавать веб-приложения, работающие в компьютерных сетях и Интернете. Однако, чтобы узнать, как разрабатывать приложения для TCP и UDP, вам придется подождать до главы 13.

Обратите внимание, что для того, чтобы успешно усвоить содержание этой и следующей главы, вам понадобятся некие базовые знания о протоколе HTTP, работе с сетями и функционировании компьютерных сетей.

В этой главе рассмотрены следующие темы:

- ❑ что такое протокол TCP/IP и почему он так важен;
- ❑ протоколы IPv4 и IPv6;
- ❑ утилита командной строки *netcat*;
- ❑ *DNS*-поиск в Go;
- ❑ пакет *net/http*;
- ❑ структуры *http.Response*, *http.Request* и *http.Transport*;
- ❑ создание веб-серверов на Go;
- ❑ программирование веб-клиентов на Go;
- ❑ создание веб-сайтов на Go;
- ❑ *gRPC* и Go;
- ❑ тип *http.NewServeMux*;
- ❑ *Wireshark* и *tshark*;
- ❑ разрыв HTTP-соединений, ожидание которых занимает слишком много времени, на стороне сервера и на стороне клиента.

Что такое net/http, net и http.RoundTripper

Центральная тема этой главы — пакет `net/http`, который предоставляет функции, позволяющие разрабатывать мощные веб-серверы и веб-клиенты. Методы `http.Set()` и `http.Get()` применяются для выполнения запросов HTTP и HTTPS, а функция `http.ListenAndServe()` — для создания веб-серверов, передавая ей IP-адрес и порт TCP, который будет прослушивать сервер. Также в состав этого пакета входят функции, которые обрабатывают входящие запросы.

Кроме пакета `net/http`, в некоторых программах, представленных в этой главе, мы будем использовать пакет `net`. Однако функциональность `net` более подробно раскрыта в главе 13.

Наконец, вам будет полезно узнать, что `http.RoundTripper` — это *интерфейс*, который позволяет создавать элементы Go, способные выполнять HTTP-транзакции. Проще говоря, это означает, что такие элементы Go способны получать отклики `http.Response` для заданных запросов `http.Request`. О том, что такое `http.Response` и `http.Request`, вы скоро узнаете.

Тип http.Response

Определение структуры `http.Response`, которое находится в файле <https://golang.org/src/net/http/response.go>, выглядит так:

```
type Response struct {
    Status      string    // e.g. "200 OK"
    StatusCode  int       // e.g. 200
    Proto       string    // e.g. "HTTP/1.0"
    ProtoMajor  int       // e.g. 1
    ProtoMinor  int       // e.g. 0
    Header      Header
    Body         io.ReadCloser
    ContentLength int64
    TransferEncoding []string
    Close        bool
    Uncompressed    bool
    Trailer         Header
    Request          *Request
    TLS              *tls.ConnectionState
}
```

Цель этого довольно сложного типа `http.Response` — ответ на HTTP-запрос. В исходном файле <https://golang.org/src/net/http/response.go> содержится подробное описание назначения каждого из полей этой структуры, как для большинства структурных типов, входящих в состав стандартной библиотеки Go.

Тип `http.Request`

Назначение типа `http.Request` — представление для HTTP-запроса, уже полученного сервером или готового к отправке от HTTP-клиента на сервер.

Структурный тип `http.Request`, определенный в файле <https://golang.org/src/net/http/request.go>, выглядит так:

```
type Request struct {
    Method string
    URL *url.URL
    Proto   string    // "HTTP/1.0"
    ProtoMajor int      // 1
    ProtoMinor int      // 0
    Header Header
    Body io.ReadCloser
    GetBody func() (io.ReadCloser, error)
    ContentLength int64
    TransferEncoding []string
    Close bool
    Host string
    Form url.Values
    PostForm url.Values
    MultipartForm *multipart.Form
    Trailer Header
    RemoteAddr string
    RequestURI string
    TLS *tls.ConnectionState
    Cancel <-chan struct{}
    Response *Response
    ctx context.Context
}
```

Тип `http.Transport`

Определение структуры `http.Transport`, которое находится по адресу <https://golang.org/src/net/http/transport.go>, выглядит так:

```
type Transport struct {
    idleMu      sync.Mutex
    wantIdle    bool
    idleConn    map[connectMethodKey][]*persistConn
    idleConnCh  map[connectMethodKey]chan *persistConn
    idleLRU     connLRU
    reqMu       sync.Mutex
    reqCanceler map[*Request]func(error)
    altMu       sync.Mutex
    altProto    atomic.Value
    Proxy func(*Request) (*url.URL, error)
```

```

DialContext func(ctx context.Context, network, addr string) (net.Conn, error)
Dial func(network, addr string) (net.Conn, error)
DialTLS func(network, addr string) (net.Conn, error)
TLSClientConfig *tls.Config
TLSHandshakeTimeout time.Duration
DisableKeepAlives bool
DisableCompression bool
MaxIdleConns int
MaxIdleConnsPerHost int
IdleConnTimeout time.Duration
ResponseHeaderTimeout time.Duration
ExpectContinueTimeout time.Duration
TLSNextProto map[string]func(authority string, c *tls.Conn) RoundTripper
ProxyConnectHeader Header
MaxResponseHeaderBytes int64
nextProtoOnce sync.Once
h2transport *http2Transport
}

```

Как видим, структура `http.Transport` довольно сложна и содержит очень большое количество полей. К счастью, нам не придется применять структуру `http.Transport` во всех программах, а когда мы все же будем ее использовать, не нужно будет каждый раз иметь дело со всеми ее полями.

Структура `http.Transport` реализует интерфейс `http.RoundTripper`, поддерживает протоколы HTTP, HTTPS и HTTP-прокси. Обратите внимание, что `http.Transport` — довольно низкоуровневый тип данных, в то время как структура `http.Client`, которая также используется в этой главе, реализует HTTP-клиент высокого уровня.

Что такое TCP/IP

TCP/IP — это семейство протоколов, на которых работает Интернет. Их название происходит от двух самых известных протоколов: *TCP* и *IP*.

TCP расшифровывается как Transmission Control Protocol — протокол управления передачей. Программное обеспечение TCP передает данные между машинами, используя сегменты, которые называются *TCP-пакетами*. Главное преимущество TCP состоит в том, что это надежный протокол, который гарантирует, что пакет был доставлен, и программисту не требуется писать для этого дополнительный код. Если доставка пакета не подтверждается, то TCP отправляет этот пакет повторно. Пакет TCP также может использоваться для установления соединений, передачи данных, отправки подтверждений и закрытия соединений.

Когда между двумя машинами устанавливается TCP-соединение, создается виртуальный полнодуплексный канал, похожий на телефонный звонок. Эти машины постоянно связываются, чтобы убедиться, что данные отправлены и получены

правильно. Если по какой-либо причине соединение не удастся, две машины попытаются найти проблему и сообщить об этом соответствующему приложению.

IP расшифровывается как Internet Protocol. Главное достоинство IP в том, что он по своей природе не является надежным протоколом. IP инкапсулирует данные, которые передаются по сети TCP/IP, так как отвечает за доставку пакетов с исходного хоста на хост назначения в соответствии с IP-адресами. IP обеспечивает способ адресации для эффективной доставки пакета в точку назначения. Несмотря на то что существуют специальные выделенные устройства — маршрутизаторы, осуществляющие IP-маршрутизацию, каждое TCP/IP-устройство выполняет некоторую базовую маршрутизацию самостоятельно.

Протокол *UDP* (User Datagram Protocol, протокол передачи датаграмм) основан на IP, следовательно, также ненадежен. Вообще, протокол UDP проще, чем TCP, потому, что UDP по своей природе не является надежным. В результате UDP-сообщения могут быть потеряны, продублированы или доставлены не в должной последовательности. Кроме того, пакеты могут прибывать быстрее, чем получатель способен их обрабатывать. Поэтому протокол UDP используется тогда, когда скорость важнее надежности.

Что такое IPv4 и IPv6

Первая версия протокола IP сейчас называется *IPv4*, чтобы отличать ее от последней версии этого протокола — *IPv6*.

Главная проблема IPv4 состоит в том, что у него заканчиваются IP-адреса. Именно поэтому создан протокол IPv6. Это произошло из-за того, что на адрес IPv4 выделяется всего 32 бита, что позволяет использовать в общей сложности 2^{32} (4 294 967 296) IP-адресов. В IPv6, напротив, представление адреса занимает 128 бит. Адреса IPv4 имеют формат 10.20.32.245 (четыре части, разделенные точками), а формат адреса IPv6 — 3fce:1706:4523:3:150:f8ff:fe21:56cf (восемь частей, разделенных двоеточиями).

Утилита командной строки nc(1)

Утилита *nc(1)*, которая также называется *netcat(1)*, очень удобна для тестирования серверов и клиентов TCP/IP. В этом разделе рассмотрены самые распространенные варианты этой утилиты.

Например, утилиту *nc(1)* можно применять в качестве клиента для сервиса TCP, который работает на компьютере с IP-адресом 10.10.1.123 и прослушивает номер порта 1234:

```
$ nc 10.10.1.123 1234
```

По умолчанию `nc(1)` использует протокол TCP. Но если выполнить `nc(1)` с флагом `-u`, то `nc(1)` будет использовать протокол UDP.

Если установить параметр `-l`, то `netcat(1)` будет действовать как сервер, то есть начнет прослушивать соединения с заданным номером порта.

Наконец, `netcat(1)` с параметрами `-v` и `-vv` будет генерировать подробный отчет, который пригодится при устранении неполадок в сетевых подключениях.

Утилита `netcat(1)` также помогает тестировать HTTP-приложения, однако подробно эта тема рассмотрена в главе 13 на примере разработки собственных клиентов и серверов TCP и UDP. Поэтому в данной главе к утилите `netcat(1)` обратимся только один раз.

Чтение конфигурации сетевых интерфейсов

Конфигурация сети состоит из четырех основных элементов, таких как IP-адрес интерфейса, сетевая маска интерфейса, DNS-серверы, обеспечивающие маршрутизацию к данному компьютеру, а также шлюз или маршрутизатор, используемый по умолчанию для данного компьютера. Но есть проблема: мы не можем получить все эти данные, используя «родной», переносимый код Go. Другими словами, не существует переносимого кода, позволяющего получить DNS-конфигурацию и информацию о шлюзе по умолчанию на компьютере с UNIX.

Поэтому в данном разделе показано, как считывать конфигурацию сетевых интерфейсов UNIX-машины на Go. Для этого мы рассмотрим переносимые утилиты, которые позволяют получить информацию о сетевых интерфейсах.

Исходный код первой утилиты, которая называется `netConfig.go`, мы разделим на три части. Первая часть `netConfig.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Функция `net.Interfaces()` возвращает все интерфейсы текущего компьютера в виде среза, содержащего элементы типа `net.Interface`. Этот срез мы будем использовать для получения желаемой информации.

Во второй части кода `netConfig.go` содержится следующий код Go:

```
for _, i := range interfaces {
    fmt.Printf("Interface: %v\n", i.Name)
    byName, err := net.InterfaceByName(i.Name)
    if err != nil {
        fmt.Println(err)
    }
}
```

В этом коде мы перебираем все элементы среза типа `net.Interface`, чтобы получить нужную информацию.

Остальной код Go из файла `netConfig.go` выглядит так:

```
addresses, err := byName.Addrs()
for k, v := range addresses {
    fmt.Printf("Interface Address #%v: %v\n", k, v.String())
}
fmt.Println()
}
```

Выполнение `netConfig.go` на компьютере с macOS Mojave и с Go версии 1.12.4 приводит к следующим результатам:

```
$ go run netConfig.go
Interface: lo0
Interface Address #0 : 127.0.0.1/8
Interface Address #1 : ::1/128
Interface Address #2 : fe80::1/64
Interface: gif0
Interface: stf0
Interface: xhc20
Interface: en0
Interface Address #0 : fe80::1435:19cd:ece8:f532/64
Interface Address #1 : ece8:f532/64
Interface: p2p0
Interface: awdl0
Interface Address #0 : fe80::888:68ff:fe01:99c/64
Interface: en1
Interface: en2
Interface: bridge0
Interface: utun0
Interface Address #0 : fe80::7fd3:e1ba:a4b1:fe22/64
```

Как видим, утилита `netConfig.go` возвращает довольно много данных, поскольку современные компьютеры обычно имеют множество сетевых интерфейсов и программа поддерживает протоколы IPv4 и IPv6.

Выполнение `netConfig.go` на компьютере с Debian Linux и с версией Go 1.7.4 приводит к следующим результатам:

```
$ go run netConfig.go
Interface: lo
```

```

Interface Address #0: 127.0.0.1/8
Interface Address #1: ::1/128
Interface: dummy0
Interface: eth0
Interface Address #0: 10.74.193.253/24
Interface Address #1: 2a01:7e00::f03c:91ff:fe69:1381/64
Interface Address #2: fe80::f03c:91ff:fe69:1381/64
Interface: teql0
Interface: tunl0
Interface: gre0
Interface: gretap0
Interface: erspan0
Interface: ip_vti0
Interface: ip6_vti0
Interface: sit0
Interface: ip6tnl0
Interface: ip6gre0

```

Обратите внимание, что главная причина, по которой сетевой интерфейс может не иметь сетевого адреса, заключается в том, что этот интерфейс не работает. Это, по сути, означает, что он в данный момент не настроен.



Не ко всем перечисленным интерфейсам подключено реальное аппаратное сетевое устройство. Более репрезентативный пример — интерфейс `lo0`, который представляет собой устройство обратной связи. Устройство обратной связи — это специальный виртуальный сетевой интерфейс, который используется компьютером для связи по сети с самим собой.

Код Go для второй утилиты, которая называется `netCapabilities.go`, мы также разделим на три части. Назначение `netCapabilities.go` — раскрыть возможности всех сетевых интерфейсов, обнаруженных в системе UNIX.

В утилите `netCapabilities.go` используются поля структуры `net.Interface`, которая определяется следующим образом:

```

type Interface struct {
    Index      int
    MTU        int
    Name       string
    HardwareAddr HardwareAddr
    Flags      Flags
}

```

Первая часть кода Go утилиты `netCapabilities.go` выглядит так:

```

package main

import (
    "fmt"
    "net"
)

```


Вторая часть кода `netCapabilities.go` содержит следующий код Go:

```
func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Print(err)
        return
    }
}
```

Последняя часть `netCapabilities.go` содержит такой код Go:

```
for _, i := range interfaces {
    fmt.Printf("Name: %v\n", i.Name)
    fmt.Println("Interface Flags:", i.Flags.String())
    fmt.Println("Interface MTU:", i.MTU)
    fmt.Println("Interface Hardware Address:", i.HardwareAddr)
    fmt.Println()
}
}
```

Запуск `netCapabilities.go` на компьютере с macOS Mojave приведет к следующим результатам:

```
$ go run netCapabilities.go
Name : lo0
Interface Flags: up|loopback|multicast
Interface MTU: 16384
Interface Hardware Address:
Name : gif0
Interface Flags: pointtopoint|multicast
Interface MTU: 1280
Interface Hardware Address:
Name : stf0
Interface Flags: 0
Interface MTU: 1280
Interface Hardware Address:
Name : XHC20
Interface Flags: 0
Interface MTU: 0
Interface Hardware Address:
Name : en0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: b8:e8:56:34:a1:c8
Name : p2p0
Interface Flags: up|broadcast|multicast
Interface MTU: 2304
Interface Hardware Address: 0a:e8:56:34:a1:c8
Name : awdl0
Interface Flags: up|broadcast|multicast
Interface MTU: 1484
```

```

Interface Hardware Address: 0a:88:68:01:09:9c
Name : en1
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0
Name : en2
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b1
Name : bridge0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0
Name : utun0
Interface Flags: up|pointtopoint|multicast
Interface MTU: 2000
Interface Hardware Address:

```

Результаты запуска `netCapabilities.go` на компьютере с Debian Linux будут аналогичными.

Наконец, если вы действительно хотите узнать параметры шлюза, используемого на компьютере по умолчанию, вы можете выполнить команду `netstat -nr` либо из оболочки, либо с помощью функции `exec.Command()`, получив результат через канал или из функции `exec.CombinedOutput()` и обработав его как текст средствами Go. Правда, такое решение нельзя назвать ни элегантным, ни идеальным!

Выполнение DNS-поиска

Аббревиатура DNS расшифровывается как Domain Name System — система доменных имен — и означает способ преобразования IP-адреса в имя, такое как `packt.com`, и обратно из имени в IP-адрес. Логика утилиты `DNS.go`, которая рассмотрена в этом разделе, довольно проста: если заданный аргумент командной строки является корректным IP-адресом, то программа обработает его как IP-адрес. В противном случае программа будет считать, что ей передано имя хоста, которое необходимо преобразовать в один или несколько IP-адресов.

Мы разделим код утилиты `DNS.go` на три части. Первая часть программы содержит следующий код Go:

```

package main

import (
    "fmt"
    "net"
    "os"
)

```

```

func lookIP(address string) ([]string, error) {
    hosts, err := net.LookupAddr(address)
    if err != nil {
        return nil, err
    }
    return hosts, nil
}

func lookHostname(hostname string) ([]string, error) {
    IPs, err := net.LookupHost(hostname)
    if err != nil {
        return nil, err
    }
    return IPs, nil
}

```

Функция `lookIP()` получает в качестве входного аргумента IP-адрес и возвращает список имен, которые соответствуют этому IP-адресу, полученный с помощью функции `net.LookupAddr()`.

Функция `lookHostname()`, наоборот, получает в качестве входного аргумента имя хоста и возвращает список соответствующих IP-адресов с помощью `net.LookupHost()`.

Во второй части `DNS.go` содержится следующий код Go:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide an argument!")
        return
    }

    input := arguments[1]
    IPaddress := net.ParseIP(input)

```

Функция `net.ParseIP()` выполняет синтаксический анализ строки как адреса IPv4 или IPv6. Если IP-адрес оказывается недействительным, то `net.ParseIP()` возвращает `nil`.

Остальной код Go утилиты `DNS.go` выглядит так:

```

    if IPaddress == nil {
        IPs, err := lookHostname(input)
        if err == nil {
            for _, singleIP := range IPs {
                fmt.Println(singleIP)
            }
        }
    } else {
        hosts, err := lookIP(input)
        if err == nil {

```

```

        for _, hostname := range hosts {
            fmt.Println(hostname)
        }
    }
}

```

Выполнение `DNS.go` с различными вариантами входных данных приводит к следующим результатам:

```

$ go run DNS.go 127.0.0.1
localhost
$ go run DNS.go 192.168.1.1
cisco
$ go run DNS.go packtpub.com
83.166.169.231
$ go run DNS.go google.com
2a00:1450:4001:816::200e
216.58.210.14
$ go run DNS.go www.google.com
2a00:1450:4001:816::2004
216.58.214.36
$ go run DNS.go cnn.com
2a04:4e42::323
2a04:4e42:600::323
2a04:4e42:400::323
2a04:4e42:200::323
151.101.193.67
151.101.1.67
151.101.129.67
151.101.65.67

```

Обратите внимание, что результаты выполнения команды `go run DNS.go 192.168.1.1` взяты из моего файла `/etc/hosts`, поскольку в моем файле `/etc/hosts` имя хоста `cisco` является псевдонимом для IP-адреса `192.168.1.1`.

Как видно из результатов выполнения последней команды, иногда у одного имени хоста (`cnn.com`) может быть несколько публичных IP-адресов. Пожалуйста, обратите особое внимание на слово «*публичных*», потому что, несмотря на то что у `www.google.com` несколько IP-адресов, у этого хоста только один публичный IP-адрес (`216.58.214.36`).

Получение NS-записей домена

Один из самых популярных DNS-запросов связан с поиском *серверов доменных имен* (name servers), данные о которых хранятся в *NS-записях* этого домена. Рассмотрим эту функциональность на примере кода из файла `Nsrecords.go`. Разделим его на две части.

Первая часть `NSrecords.go` выглядит так:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
```

Здесь мы проверяем, получила ли программа хотя бы один аргумент командной строки, чтобы у нее было с чем работать.

Остальной код Go программы `NSrecords.go` выглядит так:

```
    domain := arguments[1]
    NSs, err := net.LookupNS(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, NS := range NSs {
        fmt.Println(NS.Host)
    }
}
```

Основную работу делает функция `net.LookupNS()`. Она возвращает NS-записи домена в виде переменной среза типа `net.NS`. Именно поэтому программа выводит поле `Host` для каждого элемента среза `net.NS`. Выполнение `NSrecords.go` приводит к результатам следующего вида:

```
$ go run NSrecords.go mtsoukalos.eu
ns5.linode.com.
ns4.linode.com.
ns1.linode.com.
ns2.linode.com.
ns3.linode.com.
$ go run NSrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

Чтобы проверить правильность этих результатов, можно воспользоваться утилитой `host(1)`:

```
$ host -t ns www.mtsoukalos.eu
www.mtsoukalos.eu has no NS record
$ host -t ns mtsoukalos.eu
mtsoukalos.eu name server ns3.linode.com.
mtsoukalos.eu name server ns1.linode.com.
mtsoukalos.eu name server ns4.linode.com.
mtsoukalos.eu name server ns2.linode.com.
mtsoukalos.eu name server ns5.linode.com.
```

Получение MX-записей домена

Другой очень популярный DNS-запрос связан с получением *MX-записей* домена. MX-записи указывают на *почтовые серверы* (mail servers) домена. Чтобы показать, как выполнить эту задачу на Go, мы рассмотрим код утилиты `MXrecords.go`. Первая часть `MXrecords.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain!")
        return
    }
}
```

Во второй части `MXrecords.go` содержится такой код Go:

```
    domain := arguments[1]
    MXs, err := net.LookupMX(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, MX := range MXs {
        fmt.Println(MX.Host)
    }
}
```

Код `MXrecords.go` работает аналогично коду утилиты `NXrecords.go`, представленному в предыдущем разделе. Единственное отличие состоит в том, что в `MXrecords.go` вместо функции `net.LookupNS()` используется `net.LookupMX()`.

Выполнение MXrecords.go приведет к результатам следующего вида:

```
$ go run MXrecords.go golang.com
aspmx.l.google.com.
alt3.aspmx.l.google.com.
alt1.aspmx.l.google.com.
alt2.aspmx.l.google.com.
$ go run MXrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

Как и в прошлый раз, мы можем убедиться в правильности этих результатов с помощью утилиты `host(1)`:

```
$ host -t mx golang.com
golang.com mail is handled by 2 alt3.aspmx.l.google.com.
golang.com mail is handled by 1 aspmx.l.google.com.
golang.com mail is handled by 2 alt1.aspmx.l.google.com.
golang.com mail is handled by 2 alt2.aspmx.l.google.com.
$ host -t mx www.mtsoukalos.eu
www.mtsoukalos.eu has no MX record
```

Создание веб-сервера на Go

Используя некоторые функции стандартной библиотеки Go, можно создавать собственные веб-серверы.



Написанный на Go веб-сервер позволяет выполнять многие задачи эффективно и безопасно. Однако если вам нужен действительно мощный веб-сервер, который будет поддерживать модульность, несколько веб-сайтов и виртуальные хосты, то лучше использовать такой веб-сервер, как Apache, Nginx или Candy, который также написан на Go.

В этом разделе в качестве примера рассмотрим программу Go, которая называется `www.go`. Разделим ее на пять частей. Первая часть `www.go` содержит ожидаемые объявления импорта:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)
```

Пакет `time` для работы веб-сервера необязателен. Однако в данном случае он необходим, поскольку сервер будет отправлять своим клиентам время и дату.

Второй фрагмент кода `www.go` содержит следующий код Go:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

Это реализация первой функции-обработчика в данной программе. Функция-обработчик обслуживает один или несколько URL-адресов, в зависимости от конфигурации, указанной в коде Go, — мы можем создать столько функций-обработчиков, сколько пожелаем.

В третьей части `www.go` содержится следующий код Go:

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

В этом коде Go представлена реализация второй функции-обработчика в этой программе. Эта функция генерирует динамическое содержимое.

Четвертый фрагмент кода нашего веб-сервера посвящен аргументам командной строки и определению поддерживаемых URL-адресов:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/", myHandler)
```

Функция `http.HandleFunc()` связывает URL-адрес с функцией-обработчиком. Самое важное, что все URL-адреса, кроме `/time`, обслуживаются функцией `myHandler()`, поскольку ее первый аргумент, равный `/`, соответствует любому URL-адресу, который не подходит никакому другому обработчику.

Последняя часть программы `www.go` выглядит так:

```
err := http.ListenAndServe(PORT, nil)
if err != nil {
    fmt.Println(err)
    return
}
}
```


Чтобы запустить веб-сервер, необходимо воспользоваться функцией `http.ListenAndServe()`, указав соответствующий номер порта.

Выполнение `www.go` и подключение к созданному веб-серверу приведет к следующим результатам:

```
$ go run www.go
Using default port number: :8001
Served: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served: localhost:8001
Served: localhost:8001
```

Несмотря на то что результаты программы содержат интересные данные, я полагаю, что вы предпочли бы увидеть реальный результат работы программы в окне вашего любимого браузера. На рис. 12.1 показан результат работы функции `myHandler()` веб-сервера в Google Chrome.

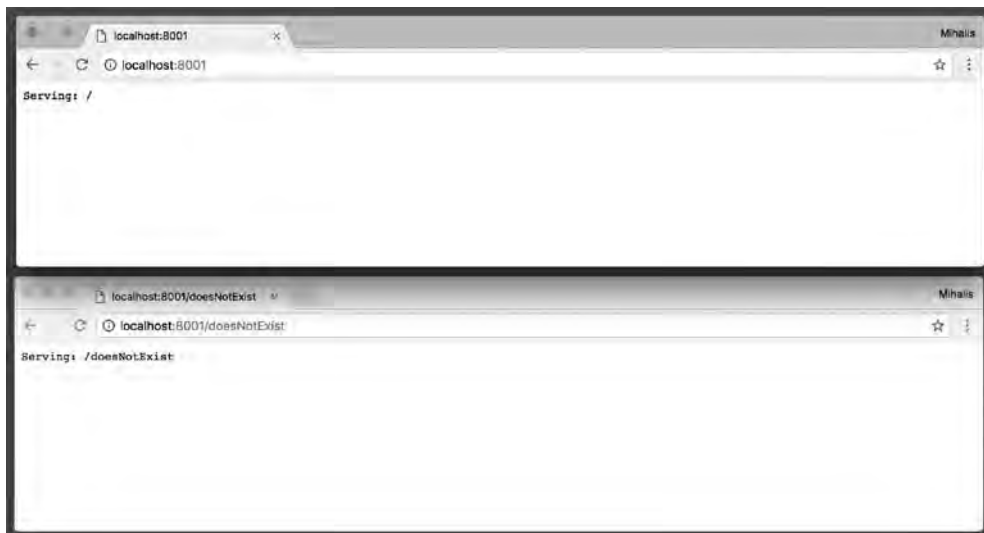


Рис 12.1. Начальная страница веб-сервера `www.go`

На рис. 12.2 видно, что `www.go` позволяет также генерировать динамические страницы. В данном случае это веб-страница, зарегистрированная по адресу `/time`, которая отображает текущую дату и время.

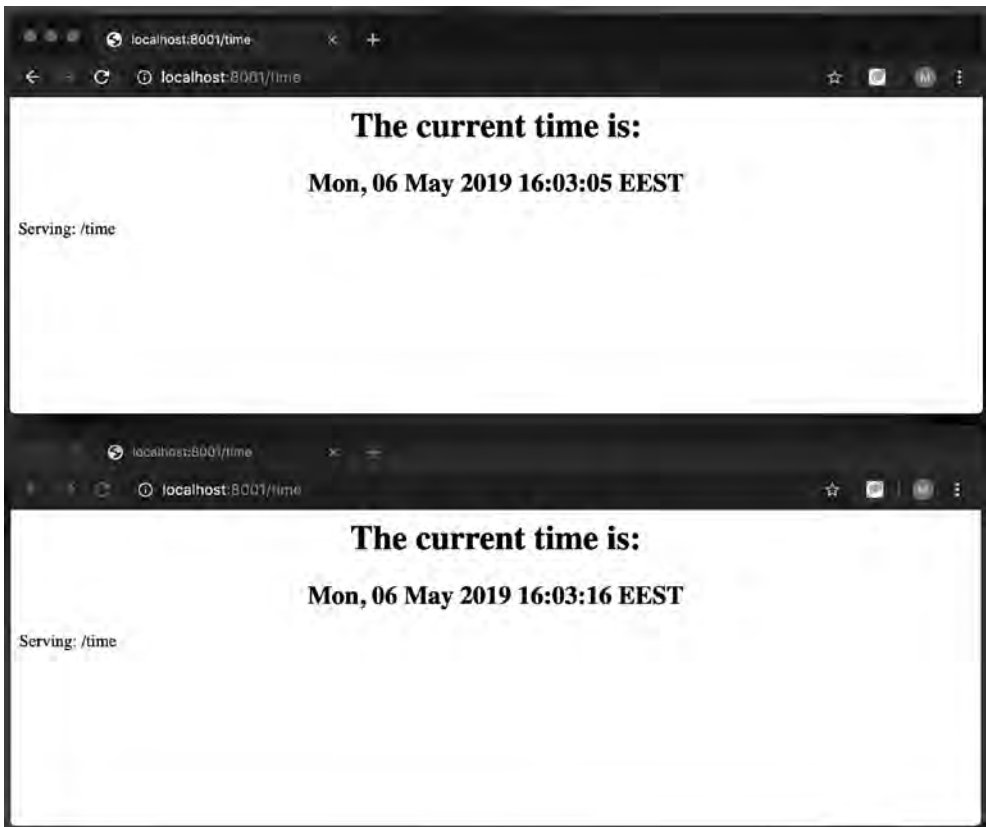


Рис. 12.2. Получение текущей даты и времени от веб-сервера `www.go`

Использование пакета `atomic`

В этом разделе вы узнаете, как использовать пакет `atomic` в среде HTTP-сервера. Для этого рассмотрим программу `atomWWW.go`, которую разделим на три части.

Первая часть `atomWWW.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
    "runtime"
    "sync/atomic"
)

var count int32
```

Переменную, которая используется пакетом `atomic`, мы сделали глобальной, чтобы она была доступной из любой точки кода.

Во второй части `atomWWW.go` содержится следующий код Go:

```
func handleAll(w http.ResponseWriter, r *http.Request) {
    atomic.AddInt32(&count, 1)
}

func getCounter(w http.ResponseWriter, r *http.Request) {
    temp := atomic.LoadInt32(&count)
    fmt.Println("Count:", temp)
    fmt.Fprintf(w, "<h1 align=\"center\">%d</h1>", count)
}
```

Счетчик `atomic`, применяемый в этой программе, связан с глобальной переменной `count` и помогает подсчитать общее количество клиентов, которые обслужил веб-сервер.

Последняя часть `atomWWW.go` выглядит так:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU() - 1)
    http.HandleFunc("/getCounter", getCounter)
    http.HandleFunc("/", handleAll)
    http.ListenAndServe(":8080", nil)
}
```

Тестирование `atomWWW.go` с помощью утилиты `ab(1)` помогает проверить, насколько эффективен пакет `atomic`, поскольку оно показывает, что переменная `count` доступна всем клиентам без каких-либо ограничений. Прежде всего запустим `atomWWW.go`:

```
$ go run atomWWW.go
Count: 1500
```

Затем выполним `ab(1)`:

```
$ ab -n 1500 -c 100 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1826891 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking localhost (be patient)
Completed 150 requests
Completed 300 requests
Completed 450 requests
Completed 600 requests
Completed 750 requests
Completed 900 requests
Completed 1050 requests
Completed 1200 requests
Completed 1350 requests
Completed 1500 requests
```

```

Finished 1500 requests
Server Software:
Server Hostname:      localhost
Server Port:          8080
Document Path:        /
Document Length:       0 bytes
Concurrency Level:     100
Time taken for tests:  0.098 seconds
Complete requests:     1500
Failed requests:        0
Total transferred:     112500 bytes
HTML transferred:      0 bytes
Requests per second:   15238.64 [#/sec] (mean)
Time per request:       6.562 [ms] (mean)
Time per request:       0.066 [ms] (mean, across all concurrent requests)
Transfer rate:          1116.11 [Kbytes/sec] received

Connection Times (ms)
              min         mean[+/-sd]       median        max
Connect:        0           3    0.5           3           5
Processing:      2           3    0.6           3           6
Waiting:         0           3    0.6           3           5
Total:          4           6    0.6           6           9

Percentage of the requests served within a certain time (ms)
 50%          6
 66%          6
 75%          7
 80%          7
 90%          7
 95%          7
 98%          8
 99%          8
100%         9 (longest request)

```

Показанная выше команда `ab(1)` отправляет 1500 запросов, при этом количество одновременных запросов равно 100.

После выполнения `ab(1)` посетите адрес `/getCounter`, чтобы получить текущее значение переменной `count`:

```
$ wget -qO- http://localhost:8080/getCounter
<h1 align="center">1500</h1>%

```

Профилирование HTTP-сервера

Как вы уже знаете из главы 11, для того чтобы профилировать приложение Go с собственным HTTP-сервером, следует использовать стандартный пакет Go `net/http/pprof`. Для этого при импорте `net/http/pprof` в папку `/debug/pprof/URL` устанавливаются различные обработчики. Вскоре вы ближе познакомитесь с этой технологией. Пока достаточно вспомнить, что для профилирования веб-приложений с помощью HTTP-сервера необходимо использовать пакет `net/http/pprof`, тогда

как для профилирования остальных видов приложений применяется стандартный Go-пакет `runtime/pprof`.

Обратите внимание, что если ваш профилировщик работает с адресом `http://localhost:8080`, то вы автоматически получите поддержку следующих веб-ссылок:

- ☐ `http://localhost:8080/debug/pprof/goroutine;`
- ☐ `http://localhost:8080/debug/pprof/heap;`
- ☐ `http://localhost:8080/debug/pprof/threadcreate;`
- ☐ `http://localhost:8080/debug/pprof/block;`
- ☐ `http://localhost:8080/debug/pprof/mutex;`
- ☐ `http://localhost:8080/debug/pprof/profile;`
- ☐ `http://localhost:8080/debug/pprof/trace?seconds=5.`

Следующая программа, которую мы рассмотрим, применяет `www.go` в качестве отправной точки и добавляет необходимый код Go, чтобы ее можно было профилировать.

Эта программа называется `wwwProfile.go`. Разделим ее на четыре части.

Заметьте, что для регистрации поддерживаемых программой путей в `wwwProfile.go` используется переменная `http.NewServeMux`. Главная причина этого заключается в том, что для того, чтобы использовать `http.NewServeMux`, нужно задать конечные точки HTTP вручную. Также обратите внимание, что мы можем определять подмножество поддерживаемых конечных точек HTTP. Если вы решите не использовать `http.NewServeMux`, то конечные точки HTTP будут зарегистрированы автоматически. Для этого вам также придется импортировать пакет `net/http/pprof`, поставив перед ним символ `_`.

Первая часть `wwwProfile.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/pprof"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
```

```

Body := "The current time is:"
fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
fmt.Printf("Served time for: %s\n", r.Host)
}

```

Реализации этих двух функций-обработчиков точно такие же, как и раньше.

Второй фрагмент кода `wwwProfile.go` выглядит так:

```

func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Println("Using port number: ", PORT)
    }

    r := http.NewServeMux()
    r.HandleFunc("/time", timeHandler)
    r.HandleFunc("/", myHandler)
}

```

В этом коде Go мы определяем URL-адреса, которые будут поддерживаться веб-сервером с помощью функций `http.NewServeMux()` и `HandleFunc()`.

Третья часть `wwwProfile.go` содержит следующий код Go:

```

r.HandleFunc("/debug/pprof/", pprof.Index)
r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
r.HandleFunc("/debug/pprof/profile", pprof.Profile)
r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
r.HandleFunc("/debug/pprof/trace", pprof.Trace)

```

В этом коде Go определены конечные точки HTTP, связанные с профилированием. Без них мы не сможем профилировать веб-приложение.

Остальной код Go выглядит так:

```

err := http.ListenAndServe(PORT, r)
if err != nil {
    fmt.Println(err)
    return
}
}

```

Этот код запускает веб-сервер Go и позволяет ему обслуживать соединения с HTTP-клиентами. Вы, вероятно, заметили, что второй параметр функции `http.ListenAndServe()` больше не равен нулю.

Как видим, в `wwwProfile.go` не определена конечная точка HTTP `/debug/pprof/goroutine`. Это оправданно, поскольку в `wwwProfile.go` отсутствуют горютины.

Выполнение `wwwProfile.go` приведет к результатам следующего вида:

```
$ go run wwwProfile.go 1234
Using port number: :1234
Served time for: localhost:1234
```

Использование профилировщика Go для получения данных — довольно простая задача. Для ее выполнения нужно запустить следующую команду, которая автоматически откроет оболочку профилировщика Go:

```
$ go tool pprof http://localhost:1234/debug/pprof/profile
Fetching profile over HTTP from http://localhost:1234/debug/pprof/profile
Saved profile in /Users/mtsouk/pprof/pprof.samples.cpu.003.pb.gz
Type: cpu
Time: Mar 27, 2018 at 10:04pm (EEST)
Duration: 30s, Total samples = 21.04s (70.13%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 19.94s, 94.77% of 21.04s total
Dropped 159 nodes (cum <= 0.11s)
Showing top 10 nodes out of 75
```

flat	flat%	sum%	cum	cum%	
13.73s	65.26%	65.26%	13.73s	65.26%	syscall.Syscall
1.58s	7.51%	72.77%	1.58s	7.51%	runtime.kevent
1.36s	6.46%	79.23%	1.36s	6.46%	runtime.mach_semaphore_signal
1.02s	4.85%	84.08%	1.02s	4.85%	runtime.usleep
0.80s	3.80%	87.88%	0.80s	3.80%	runtime.mach_semaphore_wait
0.53s	2.52%	90.40%	2.11s	10.03%	runtime.netpoll
0.44s	2.09%	92.49%	0.44s	2.09%	internal/poll.convertErr
0.26s	1.24%	93.73%	0.26s	1.24%	net.(*TCPConn).Read
0.18s	0.86%	94.58%	0.18s	0.86%	runtime.freedefer
0.04s	0.19%	94.77%	1.05s	4.99%	runtime.runqsteal

```
(pprof)
```

Теперь мы можем применять данные профилирования и анализировать их с помощью инструмента `go pprof`, как вы научились в главе 11.



Чтобы узнать результаты профилирования, посетите страницу `http://ИМЯ_ХОСТА:НОМЕР_ПОРТА/debug/pprof/`. Если `ИМЯ_ХОСТА` — `localhost`, а `НОМЕР_ПОРТА` — `1234`, получим страницу `http://localhost:1234/debug/pprof/`.

Протестировать производительность веб-серверного приложения можно с помощью утилиты `ab(1)`, которая более известна как инструмент бенчмаркинга HTTP-сервера Apache для создания трафика для сервера и выполнения бенчмаркинга `wwwProfile.go`. Это также позволит инструменту `go pprof` собрать более точные данные — для этого нужно выполнить `ab(1)` следующим образом:

```
$ ab -k -c 10 -n 100000 "http://127.0.0.1:1234/time"
This is ApacheBench, Version 2.3 <$Revision: 1807734 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
```

Licensed to The Apache Software Foundation, <http://www.apache.org/>
 Benchmarking 127.0.0.1 (be patient)

Completed 10000 requests
 Completed 20000 requests
 Completed 30000 requests
 Completed 40000 requests
 Completed 50000 requests
 Completed 60000 requests
 Completed 70000 requests
 Completed 80000 requests
 Completed 90000 requests
 Completed 100000 requests
 Finished 100000 requests

Server Software:

Server Hostname: 127.0.0.1
 Server Port: 1234
 Document Path: /time
 Document Length: 114 bytes
 Concurrency Level: 10
 Time taken for tests: 2.114 seconds
 Complete requests: 100000
 Failed requests: 0
 Keep-Alive requests: 100000
 Total transferred: 25500000 bytes
 HTML transferred: 11400000 bytes
 Requests per second: 47295.75 [#/sec] (mean)
 Time per request: 0.211 [ms] (mean)
 Time per request: 0.021 [ms] (mean, across all concurrent requests)
 Transfer rate: 11777.75 [Kbytes/sec] received

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	0 0.0	0	0
Processing:	0	0 0.7	0	13
Waiting:	0	0 0.7	0	13
Total:	0	0 0.7	0	13

Percentage of the requests served within a certain time (ms)

50%	0
66%	0
75%	0
80%	0
90%	0
95%	0
98%	0
99%	0
100%	13 (longest request)



Можно ли использовать пакет `net/http/pprof` для профилирования приложений командной строки? Да, можно! Однако пакет `net/http/pprof` особенно полезен для профилирования работающего веб-приложения и сбора оперативных данных. Именно поэтому он представлен в данной главе.

Создание веб-сайта на Go

Помните ли вы приложение `keyValue.go` из главы 4 и `kvSaveLoad.go` из главы 8? В этом подразделе вы узнаете, как создать веб-интерфейс для приложения `keyValue.go`, используя возможности стандартной библиотеки Go. Для этого мы рассмотрим файл `kvWeb.go`, который разделим на шесть частей.

Главное различие между программой `kvWeb.go` и `www.go`, разработанной ранее в этой главе, заключается в том, что в `kvWeb.go` для обработки HTTP-запросов используется тип `http.NewServeMux`, поскольку этот тип гораздо более универсальный и подходит для нетривиальных веб-приложений.

Первая часть `kvWeb.go` выглядит так:

```
package main

import (
    "encoding/gob"
    "fmt"
    "html/template"
    "net/http"
    "os"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

Этот код нам уже встречался в программе `kvSaveLoad.go` в главе 8.

Вторая часть `kvWeb.go` содержит следующий код Go:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()
```

```

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

```

```

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

```

```

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

```

```

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

```

```

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    }
}

```

```

    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}

```

Этот код Go вам также должен быть знаком, так как вы уже встречали его в программе `kvSaveLoad.go` в главе 8.

Третья часть кода `kvWeb.go` выглядит так:

```

func homePage(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Serving", r.Host, "for", r.URL.Path)
    myT := template.Must(template.ParseGlob("home.gohtml"))
    myT.ExecuteTemplate(w, "home.gohtml", nil)
}

func listAll(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Listing the contents of the KV store!")

    fmt.Fprintf(w, "<a href=\"/\" style=\"margin-right: 20px;\">Home sweet home!</a>")
    fmt.Fprintf(w, "<a href=\"/list\" style=\"margin-right: 20px;\">")
        List all elements!</a>")
    fmt.Fprintf(w, "<a href=\"/change\" style=\"margin-right: 20px;\">")
        Change an element!</a>")
    fmt.Fprintf(w, "<a href=\"/insert\" style=\"margin-right: 20px;\">")
        Insert new element!</a>")

    fmt.Fprintf(w, "<h1>The contents of the KV store are:</h1>")
    fmt.Fprintf(w, "<ul>")
    for k, v := range DATA {
        fmt.Fprintf(w, "<li>")
        fmt.Fprintf(w, "<strong>%s</strong> with value: %v\n", k, v)
        fmt.Fprintf(w, "</li>")
    }

    fmt.Fprintf(w, "</ul>")
}

```

Функция `listAll()` не использует шаблоны Go для генерации динамического вывода. Вместо этого вывод функции генерируется по ходу выполнения программы с помощью Go. Это можно считать исключением из правил, поскольку обычно

веб-приложения лучше работают с HTML-шаблонами и стандартным Go-пакетом `html/template`.

Четвертая часть `kvWeb.go` содержит следующий код Go:

```
func changeElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Changing an element of the KV store!")
    tmpl := template.Must(template.ParseFiles("update.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:      r.FormValue("name"),
        Surname:    r.FormValue("surname"),
        Id:         r.FormValue("id"),
    }

    if !CHANGE(key, n) {
        fmt.Println("Update operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}
```

В этом коде Go показано, как можно читать значения из полей HTML-формы с помощью `FormValue()`. Функция `template.Must()` — это вспомогательная функция, которая гарантирует, что предоставленный файл шаблона не содержит ошибок.

Пятый фрагмент в `kvWeb.go` содержит следующий код Go:

```
func insertElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Inserting an element to the KV store!")
    tmpl := template.Must(template.ParseFiles("insert.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:      r.FormValue("name"),
        Surname:    r.FormValue("surname"),
        Id:         r.FormValue("id"),
    }
}
```

```

    if !ADD(key, n) {
        fmt.Println("Add operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}

```

Остальной код Go выглядит так:

```

func main() {
    err := load()
    if err != nil {
        fmt.Println(err)
    }

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", homePage)
    http.HandleFunc("/change", changeElement)
    http.HandleFunc("/list", listAll)
    http.HandleFunc("/insert", insertElement)
    err = http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
    }
}

```

Функция `main()` в `kvWeb.go` намного проще, чем `main()` в `kvSaveLoad.go` из главы 8, потому что эти две программы имеют совершенно разную структуру.

Теперь пора взглянуть на файлы `gohtml`, необходимые для этого проекта, начиная с `home.gohtml`:

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>

```

```

<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

<h2>Welcome to the Go KV store!</h2>

</body>
</html>

```

Файл `home.gohtml` является статическим, то есть его содержимое не изменяется. Однако остальные файлы `gohtml` отображают информацию динамически.

Содержимое `update.gohtml` выглядит так:

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
  <h1>Element updated!</h1>
{{else}}
  <h1>Please fill in the fields:</h1>

  <form method="POST">
    <label>Key:</label><br />
    <input type="text" name="key"><br />
    <label>Name:</label><br />
    <input type="text" name="name"><br />
    <label>Surname:</label><br />
    <input type="text" name="surname"><br />
    <label>Id:</label><br />
    <input type="text" name="id"><br />
    <input type="submit">
  </form>
{{end}}

</body>
</html>

```

В основном этот код представляет собой HTML. Самая интересная его часть — оператор `if`, который указывает, что следует обновлять: форму или сообщение `Element updated!`.

Наконец, содержимое файла `insert.gohtml`:

```

<!doctype html>
<html lang="en">

```

```

<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element inserted!</h1>
{{else}}
    <h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br />
        <input type="text" name="key"><br />
        <label>Name:</label><br />
        <input type="text" name="name"><br />
        <label>Surname:</label><br />
        <input type="text" name="surname"><br />
        <label>Id:</label><br />
        <input type="text" name="id"><br />
        <input type="submit">
    </form>
{{end}}

</body>
</html>

```

Как видим, `insert.gohtml` и `update.gohtml` идентичны, кроме текста внутри тега `<title>`.

Выполнение `kvWeb.go` в оболочке UNIX выведет следующее:

```

$ go run kvWeb.go
Loading /tmp/dataFile.gob
Using default port number: :8001
Serving localhost:8001 for /
Serving localhost:8001 for /favicon.ico
Listing the contents of the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Add operation failed!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Saving /tmp/dataFile.gob
Serving localhost:8001 for /favicon.ico

```

```
Inserting an element to the KV store!  
Serving localhost:8001 for /favicon.ico  
Changing an element of the KV store!  
Serving localhost:8001 for /favicon.ico
```

Кроме того, здесь особенно интересно то, как мы можем взаимодействовать с `kvWeb.go` из браузера.

На рис. 12.3 показана начальная страница веб-сайта, определенная в `home.gohtml`.

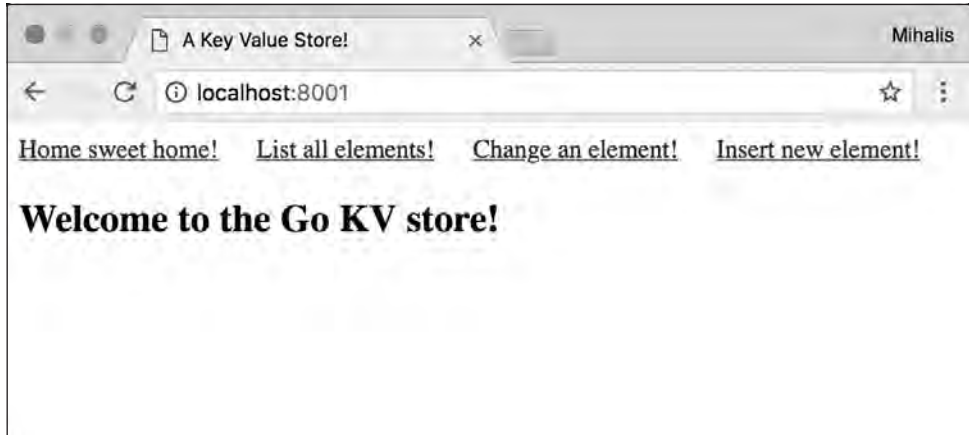


Рис. 12.3. Статическая начальная страница нашего веб-приложения

Содержимое хранилища «ключ — значение» видно на рис. 12.4.

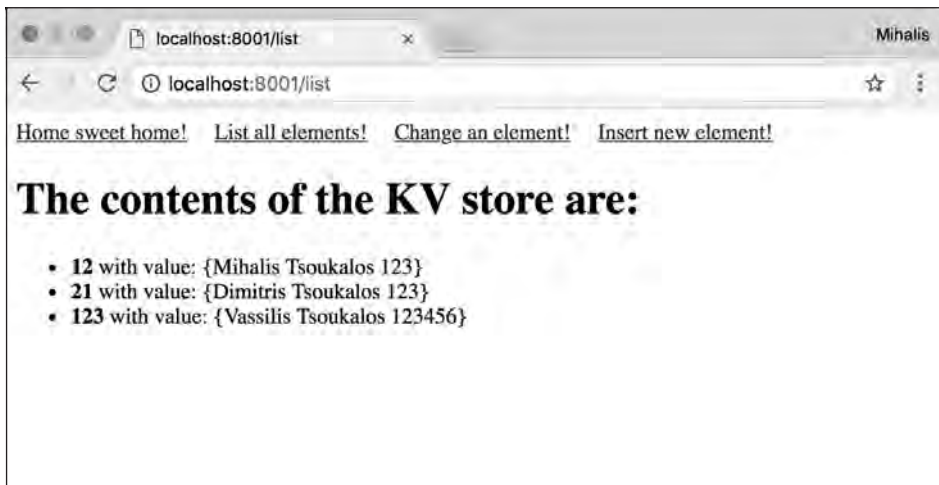


Рис. 12.4. Содержимое хранилища «ключ — значение»

На рис. 12.5 — внешний вид веб-страницы, которая позволяет добавлять новые данные в хранилище «ключ — значение» с помощью веб-интерфейса веб-приложения kvWeb.go.



The screenshot shows a web browser window titled "A Key Value Store!". The address bar displays "localhost:8001/insert". The page has a navigation bar with links: "Home sweet home!", "List all elements!", "Change an element!", and "Insert new element!". Below the navigation bar, the text "Please fill in the fields:" is followed by a form with the following fields: "Key:" (containing "89"), "Name:" (containing "aName"), "Surname:" (containing "aSurname"), "Id:" (containing "32"), and a "Submit" button.

Рис. 12.5. Добавление новой записи в хранилище «ключ — значение»

Как можно изменить значение существующего ключа с помощью веб-интерфейса веб-приложения kvWeb.go, вы видите на рис. 12.6.



The screenshot shows a web browser window titled "A Key Value Store!". The address bar displays "localhost:8001/change". The page has a navigation bar with links: "Home sweet home!", "List all elements!", "Change an element!", and "Insert new element!". Below the navigation bar, the text "Please fill in the fields:" is followed by a form with the following fields: "Key:" (containing "21"), "Name:" (containing "myName"), "Surname:" (containing "mySurname"), "Id:" (containing "1223"), and a "Submit" button.

Рис. 12.6. Изменение значения ключа в хранилище «ключ — значение»

Веб-приложение `kvWeb.go` далеко от совершенства, поэтому не стесняйтесь улучшать его — это хорошее упражнение!



В этом подразделе показано, как можно разрабатывать на Go целые веб-сайты и веб-приложения. Хотя требования к вашим приложениям наверняка будут другими, методы останутся теми же, что и в `kvWeb.go`. Обратите внимание, что сайты, написанные вручную, более безопасны, чем сайты, созданные с использованием популярных систем управления контентом.

HTTP-трассировка

Go поддерживает HTTP-трассировку с помощью стандартного пакета `net/http/httptrace`. Он позволяет поэтапно отслеживать прохождение HTTP-запроса. Рассмотрим использование стандартного Go-пакета `net/http/httptrace` на примере программы `httpTrace.go`. Разделим ее на пять частей.

Первая часть `httpTrace.go` выглядит так:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptrace"
    "os"
)
```

Как и следовало ожидать, чтобы сделать возможной HTTP-трассировку, нам нужно импортировать пакет `net/http/httptrace`.

Вторая часть `httpTrace.go` содержит следующий код Go:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: URL\n")
        return
    }

    URL := os.Args[1]
    client := http.Client{}
```

Здесь мы читаем аргументы командной строки и создаем новую переменную `http.Client`.

Объект `http.Client` заслуживает отдельного обсуждения. Он позволяет отправить запрос на сервер и получить ответ.

Его поле `Transport` позволяет задавать различные параметры HTTP-запроса вместо установленных по умолчанию.

Обратите внимание, что в реальном программном обеспечении никогда не следует использовать для объекта `http.Client` значения по умолчанию, поскольку эти значения не определяют задержки запросов, которые могли бы значительно снизить производительность программ и горутин. Кроме того, структура объектов `http.Client` позволяет безопасно применять их в конкурентных программах.

Третий фрагмент `httpTrace.go` содержит следующий код Go:

```
req, _ := http.NewRequest("GET", URL, nil)
trace := &httptrace.ClientTrace{
    GotFirstResponseByte: func() {
        fmt.Println("First response byte!")
    },
    GotConn: func(connInfo httptrace.GotConnInfo) {
        fmt.Printf("Got Conn: %+v\n", connInfo)
    },
    DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
        fmt.Printf("DNS Info: %+v\n", dnsInfo)
    },
    ConnectStart: func(network, addr string) {
        fmt.Println("Dial start")
    },
    ConnectDone: func(network, addr string, err error) {
        fmt.Println("Dial done")
    },
    WroteHeaders: func() {
        fmt.Println("Wrote headers")
    },
}
```

Этот код отвечает за отслеживание HTTP-запросов. Объект `httptrace.ClientTrace` определяет события, которые нас интересуют. Когда происходит одно из таких событий, выполняется соответствующий код. Более подробную информацию о поддерживаемых событиях и их назначении вы найдете в документации пакета net/http/httptrace.

Четвертая часть утилиты `httpTrace.go` выглядит так:

```
req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
fmt.Println("Requesting data from server!")
_, err := http.DefaultTransport.RoundTrip(req)
if err != nil {
    fmt.Println(err)
    return
}
```

Функция `httptrace.WithClientTrace()` возвращает новый контекст, полученный на основе заданного родительского контекста; метод `http.DefaultTransport.RoundTrip()` служит оберткой для объекта `http.DefaultTransport.RoundTrip`, чтобы он отслеживал текущий запрос.

Обратите внимание, что HTTP-трассировка в Go разработана только для отслеживания событий `http.Transport.RoundTrip`. Однако, поскольку при обработке одного HTTP-запроса возможно несколько перенаправлений URL-адресов, вам необходимо определить текущий запрос.

Остальной код Go программы `httpTrace.go` выглядит так:

```
response, err := client.Do(req)
if err != nil {
    fmt.Println(err)
    return
}

io.Copy(os.Stdout, response.Body)
}
```

Последняя часть программы отвечает за фактическое выполнение запроса к веб-серверу с помощью функции `Do()`, а также за получение HTTP-данных и вывод их на экран.

Выполнение `httpTrace.go` позволяет получить следующую очень важную информацию:

```
$ go run httpTrace.go http://localhost:8001/
Requesting data from server!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil>
Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142000 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil>
Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142008 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
Serving: /
```

Учтите, что, поскольку `httpTrace.go` выводит полный HTML-отклик HTTP-сервера, при тестировании реального веб-сервера мы можем получить много выходных данных. Именно поэтому мы воспользовались веб-сервером, разработанным ранее с помощью `www.go`.



Если у вас есть время, взгляните на исходный код пакета `net/http/httptest` по адресу <https://golang.org/src/net/http/httptest/trace.go>. Тогда вы сразу поймете, что `net/http/httptest` — низкоуровневый пакет, для реализации функций которого используются пакеты `context`, `reflect` и `internal/nettrace`. Помните, что вы можете сделать это для любого кода стандартной библиотеки, потому что Go — это проект с полностью открытым исходным кодом.

Тестирование HTTP-обработчиков

В этом подразделе вы научитесь тестировать HTTP-обработчики в Go. Мы начнем с кода Go из файла `www.go` и будем менять его по мере необходимости.

Новую версию `www.go` мы назовем `testWWW.go`. Рассмотрим ее, разделив на три части.

Первая часть кода `testWWW.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func CheckStatusOK(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `Fine!`)
}
```

Вторая часть `testWWW.go` содержит следующий код Go:

```
func StatusNotFound(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusNotFound)
}

func MyHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

Остальной код Go программы `testWWW.go`:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }
}
```

```

http.HandleFunc("/CheckStatusOK", CheckStatusOK)
http.HandleFunc("/StatusNotFound", StatusNotFound)
http.HandleFunc("/", MyHandler)

err := http.ListenAndServe(PORT, nil)
if err != nil {
    fmt.Println(err)
    return
}
}

```

Теперь нам нужно начать тестирование `testWWW.go`, для чего следует создать файл `testWWW_test.go`. Содержимое этого файла представлено в четырех частях.

Первая часть `testWWW_test.go` содержит следующий код Go:

```

package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
    "testing"
)

```

Обратите внимание, что для тестирования веб-приложений в Go необходимо импортировать стандартный Go-пакет `net/http/httptest`.

Вторая часть `testWWW_test.go` содержит следующий код:

```

func TestCheckStatusOK(t *testing.T) {
    req, err := http.NewRequest("GET", "/CheckStatusOK", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(CheckStatusOK)
    handler.ServeHTTP(rr, req)
}

```

Функция `httptest.NewRecorder()` возвращает объект `httptest.ResponseRecorder` и используется для записи HTTP-ответа.

Третья часть `testWWW_test.go` выглядит так:

```

status := rr.Code
if status != http.StatusOK {
    t.Errorf("handler returned %v", status)
}

expect := `Fine!`
if rr.Body.String() != expect {
    t.Errorf("handler returned %v", rr.Body.String())
}
}

```

Сначала мы убеждаемся, что код ответа соответствует ожидаемому, а затем — что тело ответа также совпадает с ожидаемым.

Остальной код Go программы `testWWW_test.go` выглядит так:

```
func TestStatusNotFound(t *testing.T) {
    req, err := http.NewRequest("GET", "/StatusNotFound", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(StatusNotFound)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusNotFound {
        t.Errorf("handler returned %v", status)
    }
}
```

Эта тестовая функция проверяет, что `StatusNotFound()` основного пакета работает должным образом.

Выполнение двух тестовых функций `testWWW_test.go` приведет к следующим результатам:

```
$ go test testWWW.go testWWW_test.go -v --count=1
=== RUN   TestCheckStatusOK
--- PASS: TestCheckStatusOK (0.00s)
=== RUN   TestStatusNotFound
--- PASS: TestStatusNotFound (0.00s)
PASS
ok   command-line-arguments   (cached)
```

Создание веб-клиента на Go

В этом разделе вы узнаете больше о разработке веб-клиентов на Go. Утилита веб-клиента, которую мы рассмотрим, называется `webClient.go`. Разделим ее на четыре части.

Первая часть `webClient.go` содержит следующий код Go:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)
```

Во второй части `webClient.go` мы читаем URL-адрес, заданный в качестве аргумента командной строки:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL := os.Args[1]
```

Главное происходит в третьей части кода `webClient.go`:

```
data, err := http.Get(URL)
```

```
if err != nil {
    fmt.Println(err)
    return
}
```

Всю работу выполняет функция `http.Get()`. Это очень удобно, если мы не хотим иметь дело с параметрами и опциями. Однако этот вариант вызова не обеспечивает гибкости. Обратите внимание, что `http.Get()` возвращает переменную `http.Response`.

Остальной код Go выглядит так:

```
    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}
```

Здесь содержимое поля `Body` структуры `http.Response` копируется в стандартный поток вывода.

Выполнение `webClient.go` выведет следующий результат (обратите внимание, что здесь представлена лишь небольшая часть информации):

```
$ go run webClient.go http://www.mtsoukalos.eu/ | head -20
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RdFa 1.0//EN"
    "http://www.w3.org/MarkUp/DTD/xhtml1-rdfa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
version="XHTML+RdFa 1.0" dir="ltr"
    xmlns:content="http://purl.org/rss/1.0/modules/content/"
    xmlns:dc="http://purl.org/dc/terms/"
    xmlns:foaf="http://xmlns.com/foaf/0.1/"
    xmlns:og="http://ogp.me/ns#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:sioc="http://rdfs.org/sioc/ns#"
    xmlns:sioc="http://rdfs.org/sioc/types#">
```



```

xmlns:skos="http://www.w3.org/2004/02/skos/core#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<head profile="http://www.w3.org/1999/xhtml/vocab">
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link rel="shortcut icon" href="http://www.mtsoukalos.eu/misc/favicon.ico"
type="image/vnd.microsoft.icon" />
<meta name="HandheldFriendly" content="true" />
<meta name="MobileOptimized" content="width" />
<meta name="Generator" content="Drupal 7 (http://drupal.org)" />

```

Основная проблема `webClient.go` заключается в том, что процесс практически не контролируется — вы либо получаете весь вывод HTML, либо вовсе ничего!

Как усовершенствовать наш веб-клиент Go

Поскольку веб-клиент, созданный ранее в разделе, довольно примитивен и совершенно не гибок, в этом подразделе показан более элегантный способ чтения URL-адресов без помощи функции `http.Get()` и с дополнительными параметрами. Утилита, которую мы рассмотрим, называется `advancedWebClient.go`. Разделим ее на пять частей.

Первый фрагмент файла `advancedWebClient.go` содержит следующий код Go:

```
package main
```

```
import (
    "fmt"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
    "path/filepath"
    "strings"
    "time"
)
```

Вторая часть файла `advancedWebClient.go` выглядит так:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL, err := url.Parse(os.Args[1])
    if err != nil {
        fmt.Println("Error in parsing:", err)
        return
    }
}
```

Третья часть файла `advancedWebClient.go` содержит следующий код Go:

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
request, err := http.NewRequest("GET", URL.String(), nil)
if err != nil {
    fmt.Println("Get:", err)
    return
}
httpData, err := c.Do(request)
if err != nil {
    fmt.Println("Error in Do():", err)
    return
}
```

Функция `http.NewRequest()` возвращает объект `http.Request` с указанием метода, URL-адреса и, возможно, тела запроса. Функция `http.Do()` отправляет HTTP-запрос (`http.Request`) посредством `http.Client` и получает HTTP-ответ (`http.Response`). Таким образом, `http.Do()` делает то же самое, что `http.Get()`, но более подробно.

Строку "GET" в вызове `http.NewRequest()` можно заменить на `http.MethodGet`.

Четвертая часть файла `advancedWebClient.go` содержит следующий код Go:

```
fmt.Println("Status code:", httpData.Status)
header, _ := httputil.DumpResponse(httpData, false)
fmt.Print(string(header))

contentType := httpData.Header.Get("Content-Type")
characterSet := strings.SplitAfter(contentType, "charset=")
if len(characterSet) > 1 {
    fmt.Println("Character Set:", characterSet[1])
}

if httpData.ContentLength == -1 {
    fmt.Println("ContentLength is unknown!")
} else {
    fmt.Println("ContentLength:", httpData.ContentLength)
}
```

В этом коде показано, как начать поиск ответа сервера, чтобы найти то, что мы хотим.

Последняя часть утилиты `advancedWebClient.go` выглядит так:

```
length := 0
var buffer [1024]byte
r := httpData.Body
for {
    n, err := r.Read(buffer[0:])
    if err != nil {
```

```

        fmt.Println(err)
        break
    }
    length = length + n
}
fmt.Println("Calculated response data length:", length)
}

```

В этом коде показано, как можно определить размер HTTP-ответа сервера. Если мы хотим вывести на экран HTML-код, то можем вывести содержимое буферной переменной `r`.

Выполнение `advancedWebClient.go` для посещения веб-страницы приведет к таким результатам — намного более подробным, чем раньше:

```

$ go run advancedWebClient.go http://www.mtsoukalos.eu
Status code: 200 OK
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 0
Cache-Control: no-cache, must-revalidate
Connection: keep-alive
Content-Language: en
Content-Type: text/html; charset=utf-8
Date: Sat, 24 Mar 2018 18:52:17 GMT
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Server: Apache/2.4.25 (Debian) PHP/5.6.33-0+deb8u1 mod_wsgi/4.5.11
Python/2.7
Vary: Accept-Encoding
Via: 1.1 varnish (Varnish/5.0)
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Generator: Drupal 7 (http://drupal.org)
X-Powered-By: PHP/5.6.33-0+deb8u1
X-Varnish: 886025
Character Set: utf-8
ContentLength is unknown!
EOF
Calculated response data length: 50176

```

Выполнение `advancedWebClient.go` для посещения другого URL-адреса выдаст немного другой результат:

```

$ go run advancedWebClient.go http://www.google.com
Status code: 200 OK
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-7
Date: Sat, 24 Mar 2018 18:52:38 GMT
Expires: -1
P3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
Set-Cookie: 1P_JAR=2018-03-24-18; expires=Mon, 23-Apr-2018 18:52:38 GMT;

```

```

path=/; domain=.google.gr
Set-Cookie:
NID=126=csX1_koD30SjC_ljAfcM2V8kTfRkppmAamLjINLfcIracMxuk6JGe4glc0Pjs8uD00
bqGaxkSW-J-ZNDJexG2ZX9pNB9E_dRc2y1KZ05V7pk0b0czE2FtS1zb50Uofl; expires=Sun,
23-Sep-2018 18:52:38 GMT; path=/; domain=.google.gr; HttpOnly
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
Character Set: ISO-8859-7
ContentLength in unknown!
EOF
Calculated response data length: 10240

```

Если попробовать выполнить `advancedWebClient.go` для некорректного URL-адреса, то получим следующий результат:

```

$ go run advancedWebClient.go http://www.google
Error in Do(): Get http://www.google: dial tcp: lookup www.google: no such host
$ go run advancedWebClient.go www.google.com
Error in Do(): Get www.google.com: unsupported protocol scheme ""

```

Не стесняйтесь вносить в `advancedWebClient.go` любые изменения, чтобы результат соответствовал вашим требованиям!

Задержки HTTP-соединений

Из этого раздела вы узнаете, как разрывать сетевые соединения, выполнение которых занимает слишком много времени. Напомню, что подобный способ рассмотрен в главе 10 на примере стандартного Go-пакета `context`. Этот метод был представлен в файле `useContext.go`.

Способ, описанный в данном разделе, реализован гораздо проще. Соответствующий код хранится в файле `clientTimeout.go`. Разделим файл на четыре части. Эта утилита принимает два аргумента командной строки: URL-адрес и время ожидания в секундах. Обратите внимание, что второй параметр является необязательным.

Первая часть кода `clientTimeout.go` выглядит так:

```

package main

import (
    "fmt"
    "io"
    "net"
    "net/http"
    "os"
    "path/filepath"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)

```

Второй фрагмент `clientTimeout.go` содержит следующий код Go:

```
func Timeout(network, host string) (net.Conn, error) {
    conn, err := net.DialTimeout(network, host, timeout)
    if err != nil {
        return nil, err
    }
    conn.SetDeadline(time.Now().Add(timeout))
    return conn, nil
}
```

Мы рассмотрим `SetDeadline()` более подробно в следующем подразделе. Функция `Timeout()` будет использоваться полем `Dial` переменной `http.Transport`.

В третьей части `clientTimeout.go` содержится следующий код:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Printf("Usage: %s URL TIMEOUT\n", filepath.Base(os.Args[0]))
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(temp) * time.Second
        }
    }

    URL := os.Args[1]
    t := http.Transport{ Dial: Timeout, }
```

Остальной код Go утилиты `clientTimeout.go` выглядит так:

```
client := http.Client{
    Transport: &t,
}
data, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
} else {
    defer data.Body.Close()
    _, err := io.Copy(os.Stdout, data.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
}
}
```

Мы протестируем веб-клиент `clientTimeout.go` с помощью веб-сервера `slowWWW.go`, разработанного в главе 10.

Если дважды запустить `clientTimeout.go`, то получим следующее:

```
$ go run clientTimeout.go http://localhost:8001
Serving: /
Delay: 0
$ go run clientTimeout.go http://localhost:8001
Get http://localhost:8001: read tcp [::1]:57397->[::1]:8001: i/o timeout
```

Как видно из полученных данных, первый запрос без проблем подключился к нужному веб-серверу. Однако второй запрос `http.Get()` занял больше времени, чем планировалось, его время ожидания истекло, и он был прерван.

Подробнее о SetDeadline

Функция `SetDeadline()` используется в пакете `net` для того, чтобы задать максимальную длительность операций чтения и записи для данного сетевого соединения. Специфика работы функции `SetDeadline()` такова, что ее необходимо вызвать перед любой операцией чтения или записи. Следует помнить, что максимальная длительность операций необходима в Go для реализации принудительного прерывания операций. Поэтому не нужно переопределять период ожидания каждый раз, когда приложение получает или отправляет данные.

Установка периода ожидания на стороне сервера

В этом подразделе вы узнаете, как устанавливать период ожидания соединения на стороне сервера. Это необходимо сделать потому, что в некоторых ситуациях клиентам для завершения HTTP-соединения требуется намного больше времени, чем ожидалось. Обычно это происходит по одной из двух причин: из-за ошибок в клиентском программном обеспечении или когда серверный процесс подвергается атаке!

Технология, которую мы рассмотрим, реализована в программе `serverTimeout.go`, которую мы разделим на четыре части. Первая часть `serverTimeout.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)
```

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

Во второй части `serverTimeout.go` содержится следующий код Go:

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

Третий фрагмент `serverTimeout.go` содержит следующий код Go:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    }

    m := http.NewServeMux()
    srv := &http.Server{
        Addr:      PORT,
        Handler:    m,
        ReadTimeout: 3 * time.Second,
        WriteTimeout: 3 * time.Second,
    }

    m.HandleFunc("/time", timeHandler)
    m.HandleFunc("/", myHandler)

    err := srv.ListenAndServe()
    if err != nil {
```

В данном случае мы использовали структуру `http.Server`, поля которой поддерживают два вида задержек. Первый из них называется `ReadTimeout`, а второй — `WriteTimeout`. Значение поля `ReadTimeout` определяет максимальную продолжительность, разрешенную для операций чтения всего запроса, включая его тело.

Значение поля `WriteTimeout` устанавливает максимально допустимое время ожидания ответа. Проще говоря, это время от конца прочитанного заголовка запроса до конца записи ответа.

Остальной код Go программы `serverTimeout.go` выглядит так:

```
    m.HandleFunc("/time", timeHandler)
    m.HandleFunc("/", myHandler)

    err := srv.ListenAndServe()
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
}

```

Теперь мы выполним программу `serverTimeout.go` и будем взаимодействовать с ней с помощью `nc(1)`:

```

$ go run serverTimeout.go
Listening on http://0.0.0.0:8001

```

Команду `nc(1)`, которая в данном случае играет роль фиктивного HTTP-клиента, который подключается к `serverTimeout.go`, необходимо выполнить в следующем формате:

```

$ time nc localhost 8001
real    0m3.012s
user    0m0.001s
sys     0m0.002s

```

Поскольку мы не давали никаких команд, HTTP-сервер завершил соединение. Утилита `time(1)` позволяет узнать время, которое потребовалось серверу для того, чтобы закрыть соединение.

Еще один способ определить период ожидания

В этом подразделе представлен еще один способ задать максимально допустимый период ожидания HTTP-соединения, на этот раз на стороне клиента. Как вы увидите, это самая простая форма определения периода ожидания, потому что здесь достаточно просто использовать объект `http.Client` и присвоить его полю `Timeout` желаемое значение периода ожидания.

Утилита, на примере которой мы рассмотрим последний способ определения периода ожидания, называется `anotherTimeout.go`. Разделим ее на четыре части.

Первая часть `anotherTimeout.go` выглядит так:

```

package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)

```


Вторая часть `anotherTimeout.go` содержит следующий код Go:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please provide a URL")
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(temp) * time.Second
        }
    }

    URL := os.Args[1]
```

Третий фрагмент `anotherTimeout.go` содержит такой код Go:

```
client := http.Client{
    Timeout: timeout,
}
client.Get(URL)
```

Именно здесь в поле `Timeout` переменной `http.Client` определяется период ожидания.

Последняя часть кода `anotherTimeout.go` выглядит так:

```
data, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
} else {
    defer data.Body.Close()
    _, err := io.Copy(os.Stdout, data.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
}
}
```

Выполнение `anotherTimeout.go` и взаимодействие с веб-сервером `slowWWW.go`, разработанным в главе 10, выведет такой результат:

```
$ go run anotherTimeout.go http://localhost:8001
Get http://localhost:8001: net/http: request canceled (Client.Timeout
exceeded while awaiting headers)
$ go run anotherTimeout.go http://localhost:8001 15
Serving: /
Delay: 8
```

Инструменты Wireshark и tshark

В этом разделе кратко представлены мощные утилиты *Wireshark* и *tshark*. Wireshark — это графическое приложение, которое является главным инструментом для анализа сетевого трафика практически любого типа. Несмотря на то что Wireshark — очень мощный инструмент, в некоторых случаях нужно что-то более легкое, без графического интерфейса, с возможностью выполнить удаленно. В таких ситуациях можно использовать tshark — версию Wireshark с интерфейсом командной строки.

К сожалению, более подробная информация о Wireshark и tshark не рассматривается в данной главе.

Go и gRPC

gRPC — это протокол, основанный на HTTP/2, который позволяет легко создавать сервисы. gRPC может использовать буферы протокола для определения *языка описания интерфейса*, а также для указания формата передаваемых сообщений. Клиенты и серверы gRPC могут быть написаны на любом языке программирования, при этом клиенты не обязательно должны быть созданы на том же языке, что и их серверы.

Процесс, который здесь описан, состоит из трех этапов. Первый — создание файла описания интерфейса, второй — разработка клиента gRPC, а третий — разработка сервера gRPC, который будет работать с клиентом gRPC.

Определение файла описания интерфейса

Как вы уже знаете из предыдущего раздела, прежде чем начать разрабатывать клиент и сервер gRPC для нашего сервиса, необходимо определить некоторые структуры данных и протоколы, которые они будут использовать.

Для сериализации структурированных данных мы воспользуемся методом *Protocol Buffers (protobuf)*. Поскольку в protobuf используется двоичный формат, он занимает меньше места, чем текстовые форматы сериализации, такие как JSON и XML. Однако данные в формате protobuf необходимо закодировать, чтобы их мог использовать компьютер, и декодировать, чтобы их мог прочесть человек.

Таким образом, чтобы использовать protobuf в приложениях, нам нужно загрузить необходимые инструменты, которые позволят работать с этим форматом, — забавно, что большинство инструментов для protobuf написаны на Go, потому что он отлично подходит для создания инструментов командной строки!

На компьютере с macOS необходимые инструменты можно загрузить с помощью *Homebrew*:

```
$ brew install protobuf
```

Чтобы обеспечить поддержку protobuf в Go, необходимо выполнить еще одну операцию, поскольку protobuf не поддерживает Go по умолчанию. Для этого нужно выполнить следующую команду:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Кроме прочего, эта команда загружает исполняемый файл `protoc-gen-go` и помещает его в каталог `~/go/bin`, который на моем компьютере является значением переменной `$GOPATH/bin`. Однако, чтобы компилятор `protoc` нашел данный файл, необходимо добавить этот каталог в переменную окружения `PATH`, что в оболочках `bash(1)` и `zsh(1)` можно сделать так:

```
$ export PATH=$PATH:~/go/bin
```

После того как все необходимые инструменты будут настроены, нужно определить структуры и функции, которые будут использоваться клиентом и сервером gRPC. В нашем случае файл описания интерфейса, который хранится под именем `api.proto`, выглядит так:

```
syntax = "proto3";

package message_service;

message Request {
    string text    = 1;
    string subtext = 2;
}

message Response {
    string text    = 1;
    string subtext = 2;
}

service MessageService {
    rpc SayIt (Request) returns (Response);
}
```

Сервер и клиент gRPC, которые мы разрабатываем, будут поддерживать этот протокол, который более или менее определяет простейшую службу обмена сообщениями с двумя основными типами, `Request` и `Response`, и одной функцией `SayIt()`.

Однако мы еще не закончили, поскольку `api.proto` должен обрабатываться и компилироваться инструментом `protobuf`. В данном случае компилятор `protobuf`, который находится в `/usr/local/bin/protoc`. Это можно сделать так:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
```

После выполнения этих команд на вашем компьютере появится дополнительный файл с именем `api.pb.go`:

```
$ ls -l api.pb.go/
-rw-r--r-- 1 mtsouk staff 7320 May 4 18:31 api.pb.go
```

Итак, для Go компилятор `protobuf` генерирует файлы `.pb.go`, которые, кроме прочего, содержат тип для каждого типа сообщений в файле описания интерфейса и интерфейс Go, который должен быть реализован на сервере gRPC. Для других языков программирования вместо расширения `.pb.go` применяются иные файловые расширения.

Файл `api.pb.go` начинается со следующих строк:

```
// Code generated by protoc-gen-go. DO NOT EDIT.
// source: api.proto

package message_service

import (
    context "context"
    fmt "fmt"
    proto "github.com/golang/protobuf/proto"
    grpc "google.golang.org/grpc"
    codes "google.golang.org/grpc/codes"
    status "google.golang.org/grpc/status"
    math "math"
)
```

Здесь говорится о том, что не следует редактировать файл `api.pb.go` самостоятельно и что пакет называется `message_service`. Чтобы ваши программы Go могли найти все файлы, связанные с `protobuf`, желательно поместить их в ваш собственный репозиторий GitHub. В нашем примере это будет репозиторий <https://github.com/mactsouk/protobuf>. Это означает, что вам нужно получить этот репозиторий с помощью следующей команды:

```
$ go get github.com/mactsouk/protobuf
```



Если вы однажды обновите `api.proto` или другой подобный файл на своем компьютере, то не забудьте совершить два действия: во-первых, обновить репозиторий GitHub, а во-вторых, выполнить команду `go get -u -v`, после которой указать адрес удаленного репозитория на GitHub, чтобы загрузить оттуда обновления на ваш локальный компьютер.

Теперь мы готовы продолжить разработку кода Go для клиента и сервера gRPC.

Обратите внимание, что если на вашем компьютере не окажется необходимого пакета Go, то при попытке скомпилировать файл описания интерфейса выведется сообщение об ошибке:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
protoc-gen-go: program not found or is not executable
--go_out: protoc-gen-go: Plugin failed with status code 1.
```

gRPC-клиент

В этом подразделе мы разработаем gRPC-клиент на Go, который будет храниться в `gClient.go`. Разделим файл на три части.

Первая часть `gClient.go` выглядит так:

```
package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
)

var port = ":8080"
```

Обратите внимание: вам не нужно загружать внешние пакеты Go, поскольку вы уже сделали это, когда выполнили показанную ниже команду, чтобы скомпилировать файл языка описания интерфейса и создать выходные файлы для Go:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Учтите, что, когда задан параметр `-u`, `go get` обновляет именованные пакеты вместе с их зависимостями. Если использовать параметр `-v`, то мы будем лучше понимать, что на самом деле происходит, так как при наличии этого параметра команда выводит дополнительную отладочную информацию.

Во второй части `gClient.go` содержится следующий код Go:

```
func AboutToSayIt(ctx context.Context, m p.MessageServiceClient,
    text string) (*p.Response, error) {
    request := &p.Request{
        Text:    text,
        Subtext: "New Message!",
    }
    r, err := m.SayIt(ctx, request)
    if err != nil {
        return nil, err
    }
    return r, nil
}
```

Вы можете назвать `AboutToSayIt()` как угодно, но сигнатура функции должна содержать параметр `context.Context`, а также параметр `MessageServiceClient`, чтобы впоследствии можно было вызывать функцию `SayIt()`. Обратите внимание, что для клиента не нужно реализовывать функции языка описания интерфейса — достаточно просто вызвать их.

Последняя часть `gClient.go` выглядит так:

```
func main() {
    conn, err := grpc.Dial(port, grpc.WithInsecure())
```

```

if err != nil {
    fmt.Println("Dial:", err)
    return
}

client := p.NewMessageServiceClient(conn)
r, err := AboutToSayIt(context.Background(), client, "My Message!")
if err != nil {
    fmt.Println(err)
}

fmt.Println("Response Text:", r.Text)
fmt.Println("Response SubText:", r.Subtext)
}

```

Чтобы подключиться к gRPC-серверу и создать новый клиент с помощью функции `NewMessageServiceClient()`, нужно вызвать `grpc.Dial()`. Имя последней функции зависит от значения оператора `package` в файле `api.proto`. Сообщение, отправляемое на gRPC-сервер, жестко закодировано в файле `gClient.go`.

Нет смысла запускать gRPC-клиент без gRPC-сервера, поэтому для запуска клиента вам нужно дождаться следующего раздела. Но если вы хотите увидеть, как программа выдаст сообщение об ошибке, то можете попробовать:

```

$ go run gClient.go
rpc error: code = Unavailable desc = all SubConns are in TransientFailure,
latest connection error: connection error: desc = "transport: Error while
dialing dial tcp :8080: connect: connection refused"
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x13d8afe]
goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch12/gRPC/gClient.go:41 +0x22e
exit status 2

```

gRPC-сервер

В этом подразделе вы узнаете, как разработать gRPC-сервер на Go. Программа, которую мы рассмотрим, называется `gServer.go`. Разделим ее на четыре части.

Первая часть `gServer.go` выглядит так:

```

package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"
    "google.golang.org/grpc"
    "net"
)

```

Поскольку Go-пакет с языком описания интерфейса носит довольно длинное имя `message_service`, я ввел для него более короткий и удобный псевдоним `p`.

Пакеты `golang.org/x/net/context` и `google.golang.org/grpc` были загружены ранее, вместе с другими зависимостями пакета `github.com/golang/protobuf/protoc-gen-go`, следовательно, вам не нужно их скачивать.



Начиная с версии Go 1.7, пакет `golang.org/x/net/context` входит в стандартную библиотеку под именем `context`. Таким образом, если хотите, можете заменить этот пакет на `context`.

Во второй части `gServer.go` содержится следующий код Go:

```
type MessageServer struct {
}
```

```
var port = ":8080"
```

Эта пустая структура нужна для того, чтобы впоследствии можно было создать gRPC-сервер в коде Go.

В третьей части `gServer.go` реализован интерфейс:

```
func (MessageServer) SayIt(ctx context.Context, r *p.Request) (*p.Response, error)
{
    fmt.Println("Request Text:", r.Text)
    fmt.Println("Request SubText:", r.Subtext)

    response := &p.Response{
        Text:      r.Text,
        Subtext:    "Got it!",
    }

    return response, nil
}
```

Сигнатура функции `SayIt()` зависит от данных в файле языка описания интерфейса и находится в файле `api.pb.go`.

Функция `SayIt()` возвращает содержимое поля `Text` клиенту, если изменяется содержимое поля `Subtext`.

Последняя часть `gServer.go` выглядит так:

```
func main() {
    server := grpc.NewServer()
    var messageServer MessageServer
    p.RegisterMessageServiceServer(server, messageServer)
    listen, err := net.Listen("tcp", port)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

```

    }
    fmt.Println("Serving requests...")
    server.Serve(listen)
}

```

Чтобы проверить соединение, вначале нужно запустить `gServer.go`:

```

$ go run gServer.go
Serving requests...

```

Запустив `gClient.go` в то время, пока работает `gServer.go`, получим такой результат:

```

$ go run gClient.go
Response Text: My Message!
Response SubText: Got it!

```

Во время работы `gClient.go` мы увидим следующий вывод от `gServer.go`:

```

Request Text: My Message!
Request SubText: New Message!

```

Программа `gClient.go` завершится автоматически, однако `gServer.go` нужно завершить вручную.

Дополнительные ресурсы

Следующие ресурсы определенно расширят ваш кругозор, поэтому, пожалуйста, найдите время, чтобы с ними ознакомиться:

- ❑ официальная веб-страница веб-сервера Apache: <http://httpd.apache.org/>;
- ❑ официальная веб-страница веб-сервера Nginx: <http://nginx.org/>;
- ❑ если вы хотите больше узнать об Интернете и протоколе TCP/IP, его многочисленных сервисах, начните с чтения *RFC*-документов. Один из источников этих документов находится по адресу <https://www.rfc-archive.org/>;
- ❑ посетите сайт Wireshark и tshark по адресу <https://www.wireshark.org/>, чтобы больше узнать об этих инструментах;
- ❑ посетите страницу документации стандартного Go-пакета `net`, которую вы найдете по адресу <https://golang.org/pkg/net/>. Это один из самых больших документов в официальной документации Go;
- ❑ посетите страницу документации Go-пакета `net/http` по адресу <https://golang.org/pkg/net/http/>;
- ❑ если вы хотите создать сайт без написания кода Go, то попробуйте использовать написанную на Go утилиту Hugo. Подробнее о фреймворке Hugo вы узнаете на сайте <https://gohugo.io/>. Однако любому программисту на Go будет интересно

взглянуть на код Go этого фреймворка, который вы найдете по адресу <https://github.com/gohugoio/hugo>;

- ❑ посетите страницу документации пакета `net/http/httptrace`: <https://golang.org/pkg/net/http/httptrace/>;
- ❑ страницу документации пакета `net/http/pprof` вы найдете по адресу <https://golang.org/pkg/net/http/pprof/>;
- ❑ посетите страницу руководства утилиты командной строки `nc(1)`, чтобы больше узнать о ее возможностях и параметрах командной строки;
- ❑ утилита `httpstat`, разработанная Дейвом Чейни, размещена по адресу <https://github.com/davecheney/httpstat>. Это хороший пример использования Go-пакета `net/http/httptrace` для HTTP-трассировки;
- ❑ более подробную информацию о веб-сервере `Caddy` вы найдете по адресу <https://github.com/caddyserver/caddy>;
- ❑ чтобы больше узнать о `protobuf`, посетите страницы <https://opensource.google.com/projects/protobuf> и <https://developers.google.com/protocol-buffers/>;
- ❑ документацию `Protocol Buffers Language Guide` вы найдете по адресу <https://developers.google.com/protocol-buffers/docs/proto3>;
- ❑ страницу документации Go-пакета `protobuf` вы найдете по адресу <https://github.com/golang/protobuf>;
- ❑ чтобы получить больше информации об утилите `ab(1)`, посетите ее страницу руководства по адресу <https://httpd.apache.org/docs/2.4/programs/ab.html>.

Упражнения

- ❑ Напишите на Go свой веб-клиент, не заглядывая в код этой главы.
- ❑ Объедините код программ `MXrecords.go` и `NSrecords.go`, чтобы получить общую утилиту, которая бы выполняла обе задачи на основе аргументов командной строки.
- ❑ Измените код утилит `MXrecords.go` и `NSrecords.go` так, чтобы они принимали в качестве входных данных в том числе и IP-адреса.
- ❑ Создайте версию `MXrecords.go` и `NSrecords.go` с помощью пакетов `cobra` и `viper`.
- ❑ Создайте свое gRPC-приложение на основе собственного языка описания интерфейса.
- ❑ Измените код `gServer.go` таким образом, чтобы использовать горутины и сохранять количество обслуженных клиентов.
- ❑ Измените программу `advancedWebClient.go` таким образом, чтобы сохранять HTML-вывод во внешнем файле.

- ❑ Попробуйте самостоятельно реализовать на Go простую версию `ab(1)`, применяя горутины и каналы.
- ❑ Измените программу `kvWeb.go` таким образом, чтобы она поддерживала операции `DELETE` и `LOOKUP`, присутствующие в исходной версии хранилища «ключ — значение».
- ❑ Измените программу `httpTrace.go` таким образом, чтобы она поддерживала флаг, запрещающий выполнение оператора `io.Copy(os.Stdout, response.Body)`.

Резюме

В этой главе вы узнали о коде Go для программирования веб-клиентов и веб-серверов, а также для создания веб-сайта на Go. Вы также познакомились со структурами `http.Response`, `http.Request` и `http.Transport`, которые позволяют определять параметры HTTP-соединения.

Кроме того, вы научились разрабатывать gRPC-приложения и научились с помощью кода Go получать сетевую конфигурацию машины UNIX и выполнять в программе Go DNS-поиск, включая получение NS- и MX-записей домена.

Наконец, я рассказал о Wireshark и tshark, двух очень популярных утилитах, которые позволяют захватывать и, что самое главное, анализировать сетевой трафик. В начале этой главы я также упомянул утилиту `nc(1)`.

В следующей главе этой книги тема сетевого программирования на Go продолжится. Однако на этот раз представлен код Go более низкого уровня, который позволит вам разрабатывать клиенты и серверы TCP, а также процессы клиентов и серверов UDP. Кроме того, вы узнаете о создании клиентов и серверов RCP.

13

Сетевое программирование: создание серверов и клиентов

В предыдущей главе обсуждались темы, связанные с сетевым программированием, такие как разработка HTTP-клиентов, HTTP-серверов и веб-приложений, выполнение DNS-поиска и период ожидания HTTP-соединений.

Эта глава выведет вас на следующий уровень: вы узнаете, как работать с протоколом HTTPS, как программировать собственные TCP-клиенты и TCP-серверы, а также собственные клиенты и серверы UDP.

Кроме того, в этой главе приводятся два примера программирования конкурентного TCP-сервера. Первый пример — относительно простой, так как конкурентный TCP-сервер будет просто вычислять и возвращать числа последовательности Фибоначчи. Однако во втором примере мы возьмем за основу код приложения `keyValue.go` из главы 4 и преобразуем хранилище «ключ — значение» в конкурентное TCP-приложение, которое способно работать без веб-браузера. В этой главе рассмотрены следующие темы:

- ❑ работа с HTTPS-трафиком;
- ❑ стандартный Go-пакет `net`;
- ❑ разработка TCP-клиентов и TCP-серверов;
- ❑ разработка конкурентного TCP-сервера;
- ❑ разработка UDP-клиентов и UDP-серверов;
- ❑ модификация программы `kvSaveLoad.go` из главы 8, чтобы она могла обслуживать запросы через TCP-соединения;
- ❑ запуск сервера TCP/IP в образе Docker;
- ❑ создание RCP-клиента и RCP-сервера.

Работа с HTTPS-трафиком

Прежде чем мы создадим сервер TCP/IP на Go, вы узнаете, как работать в Go с протоколом HTTPS, который является безопасной версией протокола HTTP. Обратите внимание, что по умолчанию для HTTPS номер TCP-порта — 443, но при желании можно использовать любой другой номер порта, если указать его в URL-адресе.

Создание сертификатов

Чтобы повторить и выполнить примеры кода из этого подраздела, вам сначала необходимо создать сертификаты, которые требует HTTPS. На компьютере с macOS Mojave для этого нужно выполнить следующие команды:

```
$ openssl genrsa -out server.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
$ openssl ecparam -genkey -name secp384r1 -out server.key
$ openssl req -new -x509 -sha256 -key server.key -out server.crt -days 3650
```

Последняя команда задаст вам несколько вопросов, которые здесь не отображаются. Что вводить в ответ — неважно, можно просто оставить большинство ответов пустыми. После выполнения этих команд будут созданы следующие два файла:

```
$ ls -l server.crt
-rw-r--r-- 1 mtsouk staff 501 May 16 09:42 server.cr
$ ls -l server.key
-rw-r--r-- 1 mtsouk staff 359 May 16 09:42 server.key
```

Обратите внимание: если сертификат является самозаверяющим, как тот, который мы только что создали, для работы HTTPS-клиента нужно использовать параметр `InsecureSkipVerify: true` в структуре `http.Transport` — скоро вы узнаете, как это сделать.

Теперь нужно создать сертификат для клиента — необходимо выполнить следующую команду:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout client.key -out
client.crt -days 3650 -subj "/"
Generating a 2048 bit RSA private key
.....+++
.....+++
writing new private key to 'client.key'
-----
```

Эта команда создаст еще два файла:

```
$ ls -l client.*
-rw-r--r-- 1 mtsouk staff 924 May 16 22:17 client.crt
-rw-r--r-- 1 mtsouk staff 1704 May 16 22:17 client.key
```

Теперь мы готовы продолжить. Поговорим о создании HTTPS-клиента.

HTTPS-клиент

В настоящее время большинство веб-сайтов работают не с HTTP, а с HTTPS. Поэтому в данном подразделе вы узнаете, как создать HTTPS-клиент. Программа, которую мы рассмотрим, называется `httpsClient.go`. Разделим ее на три части.



В зависимости от архитектуры развертывания программы Go могут просто передавать HTTP-данные, а некоторые другие сервисы (например, веб-сервер Nginx или облачные сервисы) могут предоставлять данные протокола защиты информации Secure Sockets Layer (SSL).

В первой части `httpsClient.go` содержится следующий код:

```
package main

import (
    "crypto/tls"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
    "strings"
)
```

Самым важным из этих пакетов является `crypto/tls`, который в соответствии со своей документацией частично реализует протокол безопасности на *транспортном уровне* (Transport Layer Security, TLS) версии 1.2 согласно RFC 5246 и TLS 1.3 согласно RFC 8446.

Вторая часть `httpsClient.go` выглядит так:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
}
```

```

}
URL := os.Args[1]

tr := &http.Transport{
    TLSClientConfig: &tls.Config{},
}
client := &http.Client{Transport: tr}
response, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
}
defer response.Body.Close()

```

Структура `http.Transport` настроена на использование TLS посредством параметра `TLSClientConfig`, в котором содержится другая структура с именем `tls.Config`, которая в данный момент использует значения по умолчанию.

В последней части HTTPS-клиента `httpsClient.go` содержится следующий код, отвечающий за чтение ответа от HTTPS-сервера и вывод его на экран:

```

content, _ := ioutil.ReadAll(response.Body)
s := strings.TrimSpace(string(content))

fmt.Println(s)
}

```

Выполнение `httpsClient.go` для чтения защищенного веб-сайта приведет к результатам такого вида:

```

$ go run httpsClient.go https://www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
lang="el"><head><meta content="text/html; charset=UTF-8" http-
equiv="Content-Type"><meta
content="/images/branding/googleg/1x/googleg_standard_color_128dp.png"
.
.
.

```

Однако в некоторых случаях выполнение `httpsClient.go` может привести к сбою, в зависимости от сертификата сервера:

```

$ go run httpsClient.go https://www.mtsoukalos.eu/
Get https://www.mtsoukalos.eu/: x509: certificate signed by unknown
authority

```

Решение этой проблемы — использование параметра `InsecureSkipVerify: true` при инициализации `http.Transport`.

Если хотите, попробуйте сейчас сделать это сами или дождитесь, когда мы будем рассматривать программу `TLSclient.go`.

Простой HTTPS-сервер

В этом подразделе мы рассмотрим код Go для HTTPS-сервера. Реализация простого HTTPS-сервера хранится в файле `https.go`. Разделим его на три части.

Первая часть `https.go` выглядит так:

```
package main

import (
    "fmt"
    "net/http"
)

var PORT = ":1443"
```

Во второй части `https.go` содержится следующий код:

```
func Default(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "This is an example HTTPS server!\n")
}
```

Эта функция будет обрабатывать все входящие HTTPS-соединения.

Последняя часть `https.go` выглядит так:

```
func main() {
    http.HandleFunc("/", Default)
    fmt.Println("Listening to port number", PORT)

    err := http.ListenAndServeTLS(PORT, "server.crt", "server.key", nil)
    if err != nil {
        fmt.Println("ListenAndServeTLS: ", err)
        return
    }
}
```

Функция `ListenAndServeTLS()` аналогична `ListenAndServe()`, использованной в предыдущей главе. Основное различие между ними состоит в том, что `ListenAndServeTLS()` ожидает HTTPS-соединения, в то время как `ListenAndServe()` не обслуживает HTTPS-клиенты. Кроме того, `ListenAndServeTLS()` принимает больше аргументов, чем `ListenAndServe()`, поскольку использует файл сертификата, а также файл ключа.

Выполнение программы `https.go` и подключение к ней клиента `httpsClient.go` приведет к таким результатам для клиента `httpsClient.go`:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names,
but wanted to match localhost
```

Помните, что использование самозаверенного сертификата не позволит `httpsClient.go` подключиться к нашему HTTPS-серверу. Это является проблемой

при реализации как клиента, так и сервера. В данном случае результат `https.go` будет следующим:

```
$ go run https.go
Listening to port number :1443
2019/05/17 10:11:21 http: TLS handshake error from [::1]:56716: remote
error: tls: bad certificate
```

HTTPS-сервер, разработанный в этом разделе, использует HTTPS через SSL, а это не самый безопасный вариант. Лучше использовать TLS — в следующем подразделе рассмотрена реализация на Go HTTPS-сервера, использующего TLS.

Разработка TLS-сервера и TLS-клиента

В этом подразделе мы намерены реализовать TLS-сервер с именем `TLSserver.go`. Рассмотрим его, разделив на четыре части. Этот HTTPS-сервер будет лучше, чем сервер `https.go`, реализованный в предыдущем разделе.

Первая часть `TLSserver.go` выглядит так:

```
package main

import (
    "crypto/tls"
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
)

var PORT = ":1443"

type handler struct {
```

Во второй части `TLSserver.go` содержится следующий код:

```
func (h *handler) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    w.Write([]byte("Hello world!\n"))
}
```

Это функция-обработчик веб-сервера, которая обрабатывает все клиентские соединения.

Третья часть `TLSserver.go` выглядит так:

```
func main() {
    caCert, err := ioutil.ReadFile("client.crt")
    if err != nil {
        fmt.Println(err)
        return
    }
}
```



```

caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
cfg := &tls.Config{
    ClientAuth: tls.RequireAndVerifyClientCert,
    ClientCAs: caCertPool,
}

```

Го-пакет `x509` анализирует ключи и сертификаты в кодировке *X.509*. Более подробную информацию об этом пакете можно найти по адресу <https://golang.org/pkg/crypto/x509/>.

Последняя часть `TLSserver.go` содержит следующий код:

```

srv := &http.Server{
    Addr:      PORT,
    Handler:   &handler{},
    TLSConfig: cfg,
}
fmt.Println("Listening to port number", PORT)
fmt.Println(srv.ListenAndServeTLS("server.crt", "server.key"))
}

```

Вызов функции `ListenAndServeTLS()` запускает HTTPS-сервер — его полная конфигурация содержится в структуре `http.Server`.

Если запустить `httpsClient.go` для `TLSserver.go`, то получим на стороне клиента такой результат:

```

$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names,
but wanted to match localhost

```

В этом случае вывод сервера будет таким:

```

$ go run TLSserver.go
Listening to port number :1443
2019/05/17 10:05:11 http: TLS handshake error from [::1]:56569: remote
error: tls: bad certificate

```

Как уже отмечалось, для того чтобы HTTPS-клиент `httpsClient.go` успешно взаимодействовал с `TLSserver.go`, который использует самозаверенный сертификат, необходимо добавить в структуру `http.Transport` параметр `InsecureSkipVerify: true`. Версия `httpsClient.go`, которая работает с `TLSserver.go` и содержит `InsecureSkipVerify: true`, сохранена в файле `TLSclient.go`. Разделим его на четыре части.

Первая часть `TLSclient.go` выглядит так:

```

package main

import (
    "crypto/tls"

```

```

"crypto/x509"
"fmt"
"io/ioutil"
"net/http"
"os"
"path/filepath"
)

```

Во второй части `TLScClient.go` содержится следующий код:

```

func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
    URL := os.Args[1]

    caCert, err := ioutil.ReadFile("server.crt")
    if err != nil {
        fmt.Println(err)
        return
    }

```

Третья часть `TLScClient.go` выглядит так:

```

caCertPool := x509.NewCertPool()
caCertPool.AppendCertsFromPEM(caCert)
cert, err := tls.LoadX509KeyPair("client.crt", "client.key")

if err != nil {
    fmt.Println(err)
    return
}

client := &http.Client{
    Transport: &http.Transport{
        TLSClientConfig: &tls.Config{
            RootCAs:      caCertPool,
            InsecureSkipVerify: true,
            Certificates:  []tls.Certificate{cert},
        },
    },
}

resp, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
}

```

Здесь показано использование `InsecureSkipVerify`.

В последней части `TLScient.go` содержится следующий код Go:

```
htmlData, err := ioutil.ReadAll(resp.Body)
if err != nil {
    fmt.Println(err)
    return
}

defer resp.Body.Close()
fmt.Printf("%v\n", resp.Status)
fmt.Printf(string(htmlData))
}
```

Если подключить `TLScient.go` к `TLSserver.go`, то получим ожидаемый результат:

```
$ go run TLScient.go https://localhost:1443
200 OK
Hello world!
```

Если же подключить `TLScient.go` к `https.go`, то также получим ожидаемый результат. Это означает, что `TLScient.go` — довольно хорошая реализация HTTPS-клиента:

```
$ go run TLScient.go https://localhost:1443
200 OK
This is an example HTTPS server!
```

Стандартный Go-пакет net

На этом мы закончим разговор об HTTPS. Пора рассмотреть основные протоколы TCP/IP, такие как TCP, IP и UDP.

Чтобы создать на Go TCP- или UDP-клиент либо сервер, необходимо использовать функции, предлагаемые пакетом `net`. Для подключения к сети в качестве клиента применяется функция `net.Dial()`, а для того, чтобы программа Go принимала сетевые подключения и работала как сервер, — функция `net.Listen()`. Обе функции — и `net.Dial()`, и `net.Listen()` — возвращают значение типа `net.Conn`. Этот тип реализует интерфейсы `io.Reader` и `io.Writer`. Первым параметром обеих функций является тип сети, но на этом их сходство заканчивается.

TCP-клиент

Как вы уже знаете из предыдущей главы, основной характеристикой протокола TCP является надежность. TCP-заголовок каждого пакета включает в себя поля для *порта источника* и *порта приемника*. Эти два поля в сочетании с IP-адресами источника и приемника однозначно идентифицируют каждое TCP-соединение. В этом разделе мы разработаем TCP-клиент, который называется `TCPclient.go`.

Разделим его на четыре части. Первая часть `TCPclient.go` содержит следующий код Go:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

Второй фрагмент кода `TCPclient.go` выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide host:port.")
        return
    }

    CONNECT := arguments[1]
    c, err := net.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Функция `net.Dial()` необходима для подключения к удаленному серверу. Первый параметр функции `net.Dial()` определяет тип сети, а второй — адрес сервера, который также должен включать в себя номер порта. Допустимые значения для первого параметра — `tcp`, `tcp4` (только для IPv4), `tcp6` (только для IPv6), `udp`, `udp4` (только для IPv4), `udp6` (только для IPv6), `ip`, `ip4` (только для IPv4), `ip6` (только для IPv6), `unix` (UNIX-сокеты), `unixgram` и `unixpacket`.

В третьей части `TCPclient.go` содержится следующий код:

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(c, text+"\n")
}
```

Этот код предназначен для получения данных, вводимых пользователем, что можно проверить, прочитав файл `os.Stdin`. Игнорирование значения `error`, возвращаемого функцией `reader.ReadString()`, — идея не очень хорошая. Но здесь мы таким образом немного экономим место. Конечно, никогда не следует поступать так в реальном программном обеспечении.

Последняя часть `TCPclient.go` выглядит так:

```
message, _ := bufio.NewReader(c).ReadString('\n')
fmt.Print("->: " + message)
```

```

        if strings.TrimSpace(string(text)) == "STOP" {
            fmt.Println("TCP client exiting...")
            return
        }
    }
}

```

Если для тестирования `TCPclient.go` подключить эту программу к ТСП-серверу, реализованному с использованием `netcat(1)`, то получим такой результат:

```

$ go run TCPclient.go 8001
dial tcp: address 8001: missing port in address
$ go run TCPclient.go localhost:8001
>> Hello from TCPclient.go!
->: Hi from nc!
>> STOP
->:
TCP client exiting...

```

Результат выполнения первой команды показывает, что произойдет, если не передать имя хоста как аргумент командной строки `TCPclient.go`. Результат ТСП-сервера `netcat(1)`, который должен выводиться в начале, выглядит так:

```

$ nc -l 127.0.0.1 8001
Hello from TCPclient.go!
Hi from nc!
STOP

```



Обратите внимание, что клиент для протокола TCP и UDP может быть очень обобщенным, то есть он может общаться со многими типами серверов, которые поддерживают его протокол. Как вы скоро увидите, это не относится к серверным приложениям, которые должны реализовывать предварительно организованную функциональность, используя заранее согласованный протокол.

Другая версия ТСП-клиента

В Go есть еще одно семейство функций, которое также позволяет разрабатывать ТСП-клиенты и серверы. В этом подразделе вы узнаете, как запрограммировать ТСП-клиент с помощью этих функций.

Наш ТСП-клиент называется `otherTCPclient.go`. Разделим его на четыре части. Первый фрагмент кода `otherTCPclient.go` выглядит так:

```

package main

import (
    "bufio"

```

```

    "fmt"
    "net"
    "os"
    "strings"
)

```

Во второй части `otherTCPclient.go` содержится следующий код:

```

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a server:port string!")
        return
    }

    CONNECT := arguments[1]

    tcpAddr, err := net.ResolveTCPAddr("tcp4", CONNECT)
    if err != nil {
        fmt.Println("ResolveTCPAddr:", err.Error())
        return
    }

```

Функция `net.ResolveTCPAddr()` возвращает адрес конечной точки TCP (тип `TCPAddr`) и может использоваться только для сетей TCP.

В третьей части `otherTCPclient.go` содержится следующий код:

```

conn, err := net.DialTCP("tcp4", nil, tcpAddr)
if err != nil {
    fmt.Println("DialTCP:", err.Error())
    return
}

```

Функция `net.DialTCP()` эквивалентна функции `net.Dial()`, только для сетей TCP. Остальной код `otherTCPclient.go` выглядит так:

```

for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(conn, text+"\n")

    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("->: " + message)
    if strings.TrimSpace(string(text)) == "STOP" {
        fmt.Println("TCP client exiting...")
        conn.Close()
        return
    }
}
}

```

Выполнение `otherTCPclient.go` во взаимодействии с TCP-сервером приведет к такому результату:

```
$ go run otherTCPclient.go localhost:8001
>> Hello from otherTCPclient.go!
->: Hi from netcat!
>> STOP
->:
TCP client exiting...
```

В данном примере для TCP-сервера использована утилита `netcat(1)`, которая выполняется следующим образом:

```
$ nc -l 127.0.0.1 8001
Hello from otherTCPclient.go!
Hi from netcat!
STOP
```

TCP-сервер

В этом разделе мы разработаем TCP-сервер, который возвращает клиенту текущую дату и время в одном сетевом пакете. На практике это означает, что после установки клиентского соединения сервер получит время и дату из системы UNIX и вернет эти данные клиенту.

Утилита, которую мы рассмотрим, называется `TCPserver.go`. Разделим ее на четыре части.

Первая часть `TCPserver.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)
```

Вторая часть `TCPserver.go` содержит следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide port number")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
```

```

    fmt.Println(err)
    return
}
defer l.Close()

```

Функция `net.Listen()` прослушивает соединения. Если ее второй аргумент не содержит IP-адрес, а только номер порта, то `net.Listen()` будет прослушивать все доступные IP-адреса локального компьютера.

Третий фрагмент `TCPserver.go` выглядит так:

```

c, err := l.Accept()
if err != nil {
    fmt.Println(err)
    return
}

```

Функция `Accept()` ожидает следующего соединения и возвращает порождающую переменную `Conn`. Единственная проблема этого TCP-сервера состоит в том, что он способен обслуживать только первый TCP-клиент, который к нему подключится, потому что вызов `Accept()` находится за пределами следующего цикла `for`. Остальной код Go программы `TCPserver.go` выглядит так:

```

for {
    netData, err := bufio.NewReader(c).ReadString('\n')
    if err != nil {
        fmt.Println(err)
        return
    }
    if strings.TrimSpace(string(netData)) == "STOP" {
        fmt.Println("Exiting TCP server!")
        return
    }

    fmt.Print("-> ", string(netData))
    t := time.Now()
    myTime := t.Format(time.RFC3339) + "\n"
    c.Write([]byte(myTime))
}
}

```

Выполнение `TCPserver.go` во взаимодействии с приложением TCP-клиента приведет к результатам такого вида:

```

$ go run TCPserver.go 8001
-> HELLO
Exiting TCP server!

```

На стороне клиента результат будет таким:

```

$ nc 127.0.0.1 8001
HELLO
2019-05-18T22:50:31+03:00
STOP

```


Когда утилита `TCPserver.go` пытается использовать TCP-порт, который уже занят другим процессом UNIX, появляется такое сообщение об ошибке:

```
$ go run TCPserver.go 9000
listen tcp :9000: bind: address already in use
```

Наконец, если утилита `TCPserver.go` попытается использовать TCP-порт в диапазоне от 1 до 1024, для чего в системах UNIX требуются привилегии `root`, появится такое сообщение об ошибке:

```
$ go run TCPserver.go 80
listen tcp :80: bind: permission denied
```

Другая версия TCP-сервера

В этом подразделе вы познакомитесь с альтернативной реализацией TCP-сервера на Go. На этот раз TCP-сервер реализует *сервис Echo*, который просто возвращает клиенту те же данные, которые были отправлены им. Программа, которую мы рассмотрим, называется `otherTCPserver.go`. Разделим ее на четыре части.

Первая часть `otherTCPserver.go` выглядит так:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strings"
)
```

Во второй части `otherTCPserver.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    SERVER := "localhost" + ":" + arguments[1]

    s, err := net.ResolveTCPAddr("tcp", SERVER)
    if err != nil {
        fmt.Println(err)
        return
    }

    l, err := net.ListenTCP("tcp", s)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Функция `net.ListenTCP()` — это эквивалент функции `net.Listen()` для сетей TCP.

Третий фрагмент `otherTCPserver.go`:

```
buffer := make([]byte, 1024)
conn, err := l.Accept()
if err != nil {
    fmt.Println(err)
    return
}
```

Остальной код `otherTCPserver.go` выглядит так:

```
for {
    n, err := conn.Read(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }

    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
        fmt.Println("Exiting TCP server!")
        conn.Close()
        return
    }

    fmt.Print("> ", string(buffer[0:n-1]))
    _, err = conn.Write(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Выполнение `otherTCPserver.go` и использование клиента для взаимодействия с этим сервером приведет к следующим результатам:

```
$ go run otherTCPserver.go 8001
> 1
> 2
> Hello!
> Exiting TCP server!
```

На стороне клиента, которым в этом случае является `otherTCPclient.go`, мы получим такой результат:

```
$ go run otherTCPclient.go localhost:8001
>> 1
->: 1
>> 2
->: 2
>> Hello!
```

```
->: Hello!
>> ->:
>> STOP
->: TCP client exiting...
```

Наконец, я покажу вам, как узнать имя процесса, который прослушивает данный порт TCP или UDP на машине с UNIX. Итак, чтобы узнать, какой процесс использует TCP-порт с номером 8001, нужно выполнить следующую команду:

```
$ sudo lsof -n -i :8001
COMMAND  PID  USER FD  TYPE             DEVICE SIZE/OFF NODE NAME
TCPserver 86775 mtsouk 3u  Ipv6 0x98d55014e6c9360f      0t0  TCP *:vcom-tunnel
                                     (LISTEN)
```

UDP-клиент

Если вы уже знаете, как разрабатывать TCP-клиент, то создание UDP-клиента покажется вам намного проще из-за простоты протокола UDP.



Главное различие между протоколами UDP и TCP состоит в том, что протокол UDP ненадежен. Это также означает, что в целом UDP проще, чем TCP, поскольку UDP не должен сохранять состояние UDP-соединения. Другими словами, работа UDP похожа на самонаводящуюся ракету, что в некоторых случаях идеально подходит.

В этом разделе мы рассмотрим утилиту `UDPclient.go`, код которой разделим на четыре фрагмента. Первая часть `UDPclient.go` выглядит так:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

Второй фрагмент `UDPclient.go` выглядит следующим образом:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string")
        return
    }
    CONNECT := arguments[1]
```

```

s, err := net.ResolveUDPAddr("udp4", CONNECT)
c, err := net.DialUDP("udp4", nil, s)

if err != nil {
    fmt.Println(err)
    return
}

fmt.Printf("The UDP server is %s\n", c.RemoteAddr().String())
defer c.Close()

```

Функция `net.ResolveUDPAddr()` возвращает адрес конечной точки UDP, который определяется ее вторым параметром. Первый параметр (`udp4`) указывает на то, что программа поддерживает только протокол IPv4.

Использованная здесь функция `net.DialUDP()` представляет собой аналог `net.Dial()` для сетей UDP.

Третий фрагмент `UDPclient.go` содержит следующий код:

```

for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">>> ")
    text, _ := reader.ReadString('\n')
    data := []byte(text + "\n")
    _, err = c.Write(data)
    if strings.TrimSpace(string(data)) == "STOP" {
        fmt.Println("Exiting UDP client!")
        return
    }
}

```

Этот код требует от пользователя, чтобы он ввел текст, который затем передается на UDP-сервер. Введенный пользователем текст считывается из стандартного потока ввода UNIX с помощью вызова `bufio.NewReader(os.Stdin)`. Метод `Write(data)` отправляет данные по сетевому соединению UDP.

Остальная часть кода Go выглядит так:

```

    if err != nil {
        fmt.Println(err)
        return
    }

    buffer := make([]byte, 1024)
    n, _, err := c.ReadFromUDP(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Printf("Reply: %s\n", string(buffer[0:n]))
}
}

```

После того как данные клиента отправлены, нужно дождаться данных, переданных с UDP-сервера, которые считываются с помощью функции `ReadFromUDP()`. Выполнение `UDPclient.go` и взаимодействие с утилитой `netcat(1)`, которая действует как UDP-сервер, приведет к следующим результатам:

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: Hi there!
>> Have to leave - bye!
Reply: OK.
>> STOP
Exiting UDP client!
```

На стороне UDP-сервера результат будет таким:

```
$ nc -v -u -l 127.0.0.1 8001
Hello!
Hi there!
Have to leave - bye!
OK.
STOP
^C
```

Мы нажали `Ctrl+C`, чтобы завершить выполнение `nc(1)`, так как у `nc(1)` нет кода, по которому программа бы завершала работу после того, как она получила на входе строку `STOP`.

Разработка UDP-сервера

В этом разделе мы разработаем UDP-сервер, который будет возвращать своим UDP-клиентам случайные числа от 1 до 1000. Программа, которую мы рассмотрим, называется `UDPserver.go`, и мы разделим ее на четыре фрагмента.

Первая часть `UDPserver.go` выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

Второй фрагмент UDPserver.go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]

    s, err := net.ResolveUDPAddr("udp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

В третьем фрагменте UDPserver.go содержится следующий код:

```
connection, err := net.ListenUDP("udp4", s)
if err != nil {
    fmt.Println(err)
    return
}

defer connection.Close()
buffer := make([]byte, 1024)
rand.Seed(time.Now().Unix())
```

Функция `net.ListenUDP()` — это аналог функции `net.ListenTCP()` для UDP-сетей.

Остальной код Go программы UDPserver.go выглядит так:

```
for {
    n, addr, err := connection.ReadFromUDP(buffer)
    fmt.Print("-> ", string(buffer[0:n-1]))

    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
        fmt.Println("Exiting UDP server!")
        return
    }

    data := []byte(strconv.Itoa(random(1, 1001)))
    fmt.Printf("data: %s\n", string(data))
    _, err = connection.WriteToUDP(data, addr)
    if err != nil {
        fmt.Println(err)
        return
    }
}
}
```

Функция `ReadFromUDP()` позволяет считывать данные из UDP-соединения используя буфер, который, как и следовало ожидать, является *байтовым срезом*.

Выполнение `UDPserver.go` и подключение к нему с помощью `UDPclient.go` приводит к следующим результатам:

```
$ go run UDPserver.go 8001
-> Hello!
data: 156
-> Another random number please :)
data: 944
-> Leaving...
data: 491
-> STOP
Exiting UDP server!
```

На стороне клиента результат будет таким:

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: 156
>> Another random number please :)
Reply: 944
>> Leaving...
Reply: 491
>> STOP
Exiting UDP client!
```

Конкурентный TCP-сервер

В этом разделе вы узнаете, как разработать *конкурентный TCP-сервер* с использованием горутин. Для каждого входящего соединения с TCP-сервером программа будет запускать новую горутины, которая и будет обрабатывать этот запрос. Это позволит серверу принимать больше запросов. Следовательно, конкурентный TCP-сервер сможет обслуживать несколько клиентов одновременно.

Задача нашего конкурентного TCP-сервера — принимать положительное целое число и возвращать натуральное число из последовательности Фибоначчи. Если на входе обнаруживается ошибка, то сервер возвращает -1. Поскольку вычисление чисел последовательности Фибоначчи может быть медленным, мы воспользуемся алгоритмом, который был представлен в главе 11 и применялся в программе `benchmarkMe.go`. Кроме того, на этот раз я подробнее разьясню работу этого алгоритма.

Программа, которую мы рассмотрим, называется `fibotcp.go`. Ее код представлен в пяти частях. Поскольку считается целесообразным передавать номер порта

веб-службы в качестве аргумента командной строки, мы в `fibotcp.go` тоже будем так делать.

В первой части `fibotcp.go` содержится следующий код Go:

```
package main
```

```
import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)
```

Во второй части `fibotcp.go` содержится такой код Go:

```
func f(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

В этом коде мы видим реализацию функции `f()`, которая генерирует натуральные числа, принадлежащие последовательности Фибоначчи. Поначалу этот алгоритм трудно понять, но он очень эффективен и поэтому быстр.

Прежде всего, в функции `f()` используется хеш-таблица с именем `fn`, что довольно необычно при вычислении чисел последовательности Фибоначчи. В функции `f()` также используется цикл `for`, что также довольно необычно. Наконец, в функции `f()` отсутствует рекурсия, что является главной причиной ее высокой скорости.

Идея алгоритма, реализованного в `f()`, который использует метод *динамического программирования*, состоит в том, что всякий раз, когда вычисляется очередное число Фибоначчи, оно помещается в хеш-таблицу `fn`, чтобы впоследствии не пришлось вычислять его снова. Эта простая идея экономит много времени, особенно когда нужно вычислять большие числа Фибоначчи, потому что тогда не приходится вычислять одно и то же число Фибоначчи несколько раз.

Третий фрагмент кода `fibotcp.go` выглядит так:

```
func handleConnection(c net.Conn) {
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }

        fibo := "-1\n"
        n, err := strconv.Atoi(temp)
        if err == nil {
            fibo = strconv.Itoa(f(n)) + "\n"
        }
        c.Write([]byte(string(fibo)))
    }
    time.Sleep(5 * time.Second)
    c.Close()
}
```

Функция `handleConnection()` работает со всеми клиентами конкурентного TCP-сервера.

Четвертая часть `fibotcp.go`:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
```

Остальной код Go файла `fibotcp.go` выглядит так:

```
for {
    c, err := l.Accept()
    if err != nil {
        fmt.Println(err)
```

```

        return
    }
    go handleConnection(c)
}
}

```

Конкурентность программы реализуется посредством оператора `go handleConnection(c)`, который запускает новую горутину всякий раз, когда из Интернета или локальной сети приходит новый TCP-клиент. Горутина выполняется конкурентно, что дает серверу возможность обслуживать еще больше клиентов.

Выполнение сервера `fibotcp.go` и взаимодействие с ним посредством запуска `netcat(1)` и `TCPclient.go` на двух разных терминалах выведет такой результат:

```

$ go run fibotcp.go 9000
n: 10
fibotcp: 55
n: 0
fibotcp: 1
n: -1
fibotcp: 0
n: 100
fibotcp: 3736710778780434371
n: 12
fibotcp: 144
n: 12
fibotcp: 144

```

На стороне `TCPclient.go` результат будет следующим:

```

$ go run TCPclient.go localhost:9000
>> 12
->: 144
>> a
->: -1
>> STOP
->: TCP client exiting...

```

На стороне `netcat(1)` получим такой результат:

```

$ nc localhost 9000
10
55
0
1
-1
0
100
3736710778780434371
ads
-1
STOP

```

Когда клиент отправляет процессу сервера строку `STOP`, горютина, которая обслуживает этот ТСП-клиент, прекращает работу, что приводит к завершению соединения.

Наконец, впечатляет то, что оба клиента обслуживаются одновременно. Это можно проверить с помощью следующей команды:

```
$ netstat -anp TCP | grep 9000
tcp4    0 0 127.0.0.1.9000      127.0.0.1.57309    ESTABLISHED
tcp4    0 0 127.0.0.1.57309    127.0.0.1.9000    ESTABLISHED
tcp4    0 0 127.0.0.1.9000      127.0.0.1.57305    ESTABLISHED
tcp4    0 0 127.0.0.1.57305    127.0.0.1.9000    ESTABLISHED
tcp4    0 0 *.9000              *.*                LISTEN
```

Последняя строка результатов этой команды говорит о том, что существует процесс, который прослушивает порт `9000`, следовательно, мы все равно можем подключиться к порту `9000`. Первые две строки говорят о том, что существует клиент, который использует порт `57309` для связи с процессом сервера. Третья и четвертая строки предыдущего вывода подтверждают, что существует другой клиент, который связывается с сервером, прослушивающим порт `9000`. Этот клиент применяет ТСП-порт `57305`.

Удобный конкурентный ТСП-сервер

Конкурентный ТСП-сервер, который мы создали в предыдущем разделе, работает нормально, однако для практического применения он не подходит. Поэтому в данном подразделе вы узнаете, как преобразовать приложение `keyValue.go` из главы 4 в полнофункциональное конкурентное приложение ТСП.

Сейчас мы создадим собственный ТСП-протокол для сетевого взаимодействия с хранилищем «ключ — значение». Для каждой функции хранилища «ключ — значение» нам понадобится ключевое слово. Для простоты за каждым ключевым словом будут следовать соответствующие данные. Результатом большинства команд будет сообщение об успехе или неудаче операции.



Разработка собственных протоколов на основе ТСП или UDP — непростая задача. При разработке нового протокола необходимы исключительная точность и осторожность.

Утилита, представленная в этом разделе, называется `kvTCP.go`. Разделим ее на шесть частей.

Первая часть `kvTCP.go` выглядит так:

```
package main
```

```
import (
```

```

    "bufio"
    "encoding/gob"
    "fmt"
    "net"
    "os"
    "strings"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

const welcome = "Welcome to the Key Value store!\n"

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"

```

Во второй части kvTCP.go содержится следующий код Go:

```

func handleConnection(c net.Conn) {
    c.Write([]byte(welcome))
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }

        command := strings.TrimSpace(string(netData))
        tokens := strings.Fields(command)
        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")
        }
    }
}

```

```

switch tokens[0] {
case "STOP":
    err = save()
    if err != nil {
        fmt.Println(err)
    }
    c.Close()
    return
case "PRINT":
    PRINT(c)
case "DELETE":
    if !DELETE(tokens[1]) {
        netData := "Delete operation failed!\n"
        c.Write([]byte(netData))
    } else {
        netData := "Delete operation successful!\n"
        c.Write([]byte(netData))
    }
case "ADD":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !ADD(tokens[1], n) {
        netData := "Add operation failed!\n"
        c.Write([]byte(netData))
    } else {
        netData := "Add operation successful!\n"
        c.Write([]byte(netData))
    }
    err = save()
    if err != nil {
        fmt.Println(err)
    }
case "LOOKUP":
    n := LOOKUP(tokens[1])
    if n != nil {
        netData := fmt.Sprintf("%v\n", *n)
        c.Write([]byte(netData))
    } else {
        netData := "Did not find key!\n"
        c.Write([]byte(netData))
    }
case "CHANGE":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !CHANGE(tokens[1], n) {
        netData := "Update operation failed!\n"
        c.Write([]byte(netData))
    } else {
        netData := "Update operation successful!\n"
        c.Write([]byte(netData))
    }
    err = save()
}

```

```

        if err != nil {
            fmt.Println(err)
        }
    default:
        netData := "Unknown command - please try again!\n"
        c.Write([]byte(netData))
    }
}
}

```

Функция `handleConnection()` связывается с каждым TCP-клиентом и интерпретирует входные данные этого клиента.

Третий фрагмент `kvTCP.go` содержит следующий код Go:

```

func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

```

Четвертый фрагмент `kvTCP.go` выглядит так:

```
func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}
```

Реализация этих функций такая же, как в `keyValue.go`. Ни одна из них не обращается к TCP-клиенту напрямую.

В пятой части `kvTCP.go` содержится следующий код Go:

```
func PRINT(c net.Conn) {
    for k, d := range DATA {
        netData := fmt.Sprintf("key: %s value: %v\n", k, d)
        c.Write([]byte(netData))
    }
}
```

Функция `PRINT()` передает данные прямо TCP-клиенту, по одной строке.

Остальной код Go программы выглядит так:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    err = load()
    if err != nil {
        fmt.Println(err)
    }

    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }
        go handleConnection(c)
    }
}
```

Выполнение kvTCP.go приведет к таким результатам:

```
$ go run kvTCP.go 9000
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
Saving /tmp/dataFile.gob
```

В нашем примере роль TCP-клиента для kvTCP.go выполняет утилита netcat(1):

```
$ nc localhost 9000
Welcome to the Key Value store!
PRINT
LOOKUP 1
Did not find key!
```



```

ADD 1 2 3 4
Add operation successful!
LOOKUP 1
{2 3 4}
ADD 4 -1 -2 -3
Add operation successful!
PRINT
key: 1 value: {2 3 4}
key: 4 value: {-1 -2 -3}
STOP

```

kvTCP.go — конкурентное приложение, которое применяет горутины и способно обслуживать несколько TCP-клиентов одновременно. Однако все эти TCP-клиенты будут взаимодействовать с одними и теми же данными.

Создание образа Docker для TCP/IP-сервера на Go

В этом разделе вы узнаете, как поместить файл kvTCP.go в образ Docker и использовать его оттуда. Это очень удобный способ работы с приложением TCP/IP, поскольку образ Docker легко переносится на другие машины и развертывается в Kubernetes.

Как и следовало ожидать, все начинается с Dockerfile с таким содержимым:

```

FROM golang:latest

RUN mkdir /files
COPY kvTCP.go /files
WORKDIR /files

RUN go build -o /files/kvTCP kvTCP.go
ENTRYPOINT ["/files/kvTCP", "80"]

```

После этого нужно создать образ Docker:

```

$ docker build -t kvtcp:latest .
Sending build context to Docker daemon 6.656kB
Step 1/6 : FROM golang:latest
--> 7ced090ee82e
Step 2/6 : RUN mkdir /files
--> Running in bbbbada6271f
Removing intermediate container bbbbada6271f
--> 5b0a621eee29
Step 3/6 : COPY kvTCP.go /files
--> 4aab441b14c2
Step 4/6 : WORKDIR /files
--> Running in 718560bed2e

```

```

Removing intermediate container 7185606bed2e
---> 744e9800fdb
Step 5/6 : RUN go build -o /files/kvTCP kvTCP.go
---> Running in f44fcbc8951b
Removing intermediate container f44fcbc8951b
---> a8d00c7ead13
Step 6/6 : ENTRYPOINT ["/files/kvTCP","80"]
---> Running in ec3227170e09
Removing intermediate container ec3227170e09
---> b65ba728849a
Successfully built b65ba728849a
Successfully tagged kvtcp:latest

```

Выполнив команду `docker images`, мы убедимся, что создали желаемый образ Docker:

```

$ docker images

```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
kvtcp	latest	b65ba728849a	26 minutes ago	777MB
landoop/kafka-lenses-dev	latest	289093ceee7b	2 days ago	1.37GB
golang	latest	7ced090ee82e	9 days ago	774MB

Затем нужно выполнить этот образ Docker. Следует указать дополнительные параметры, чтобы предоставить номера портов, которые мы хотим сделать доступными извне:

```

$ docker run -d -p 5801:80 kvtcp:latest
af9939992a25cb5ccf405b1b97b9c813fb4cb3f3a2e5b13942db637709c8cee2

```

Последняя команда открывает порт `80` из образа Docker для локального компьютера с помощью порта `5801` локального компьютера.

Теперь мы сможем применить TCP-сервер, расположенный в образе Docker, так:

```

$ nc 127.0.0.1 5801
Welcome to the Key Value store!

```

Обратите внимание, что, если хранить данные вне образа Docker, по окончании работы с ним данные будут потеряны.

Эта возможность Docker позволяет использовать любой номер порта, через который мы хотим получить доступ к TCP-серверу в образе Docker. Но если вы попытаетесь снова назначить порт с номером `80`, то получите следующее сообщение об ошибке:

```

$ docker run -d -p 5801:80 kvtcp:latest
709d44be8668284b101d7dfc253938d13e6797d812821838aa5ab18ea48527ec
docker: Error response from daemon: driver failed programming external connectivity on endpoint eager_nobel (fa9d43d3c129734576e824753703d8ac3ff51bcdcdc20e6937b30f3bcfefeff7): Bind for 0.0.0.0:5801 failed: port is already allocated.

```

В этом сообщении об ошибке говорится, что данный порт уже выделен и поэтому не может быть использован снова. Однако есть возможность обойти это ограничение, которую можно проиллюстрировать с помощью следующей команды:

```
$ docker run -d -p 5801:80 -p 2000:80 kvtcp:latest
5cbb17c5bbf720eaa5ce0f1e11cc73dbe5bef3cc925b7936ae1b97e245d54ae8
```

Здесь в команде запуска Docker создается отображение номера порта 80 из образа Docker на два внешних TCP-порта (5801 и 2000). Обратите внимание, что здесь работает только один сервер kvTCP и существует только одна копия данных, несмотря на то что эти данные доступны через несколько портов.

Мы можем это проверить с помощью команды `docker ps`:

```
$ docker ps
CONTAINER ID   IMAGE          COMMAND                  CREATED        STATUS
PORTS         NAMES
5cbb17c5bbf7   kvtcp:latest   "/files/kvTCP 80"       3 seconds ago Up 2 seconds
0.0.0.0:2000->80/tcp, 0.0.0.0:5801->80/tcp   compassionate_morse
```

Дистанционный вызов процедур

Дистанционный вызов процедур (Remote Procedure Call, RPC) — это клиент-серверный механизм для межпроцессного взаимодействия, применяющий протокол TCP/IP. RPC-клиент и RPC-сервер, которые мы разработаем, основаны на пакете, который называется `sharedRPC.go`:

```
package sharedRPC

type MyFloats struct {
    A1, A2 float64
}

type MyInterface interface {
    Multiply(arguments *MyFloats, reply *float64) error
    Power(arguments *MyFloats, reply *float64) error
}
```

Пакет `sharedRPC` определяет интерфейс `MyInterface` и структуру `MyFloats`, которые будут использоваться и клиентом, и сервером. Но реализует этот интерфейс только RPC-сервер.

После этого нам нужно установить пакет `sharedRPC.go`, выполнив следующие команды:

```
$ mkdir -p ~/go/src/sharedRPC
$ cp sharedRPC.go ~/go/src/sharedRPC/
$ go install sharedRPC
```

RPC-клиент

В этом подразделе мы рассмотрим код Go RPC-клиента, который хранится в файле `RPCclient.go`. Разделим его на четыре части.

Первая часть `RPCclient.go` выглядит так:

```
package main

import (
    "fmt"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

Во втором фрагменте `RPCclient.go` содержится следующий код Go:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string!")
        return
    }

    CONNECT := arguments[1]
    c, err := rpc.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Обратите внимание, что для подключения к RPC-серверу вместо функции `net.Dial()` применяется `rpc.Dial()`, несмотря на то что RCP-сервер использует протокол TCP.

Третий фрагмент `RPCclient.go` выглядит так:

```
args := sharedRPC.MyFloats{16, -0.5}
var reply float64

err = c.Call("MyInterface.Multiply", args, &reply)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply (Multiply): %f\n", reply)
```

С помощью функции `Call()` между RPC-клиентом и RPC-сервером передаются имена функций, их аргументы и результаты вызовов функций, поскольку RPC-клиент ничего не знает о фактической реализации этих функций.

Остальной код Go программы `RPCclient.go` выглядит так:

```
err = c.Call("MyInterface.Power", args, &reply)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply (Power): %f\n", reply)
}
```

Если запустить `RPCclient.go`, не запуская RPC-сервер, то выведется следующее сообщение об ошибке:

```
$ go run RPCclient.go localhost:1234
dial tcp [::1]:1234: connect: connection refused
```

RPC-сервер

Код RPC-сервера хранится в файле `RPCserver.go`. Мы его рассмотрим, разделив на пять частей.

Первая часть `RPCserver.go` выглядит так:

```
package main

import (
    "fmt"
    "math"
    "net"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

Второй фрагмент `RPCserver.go` содержит следующий код Go:

```
type MyInterface struct{}

func Power(x, y float64) float64 {
    return math.Pow(x, y)
}

func (t *MyInterface) Multiply(arguments *sharedRPC.MyFloats, reply *float64) error {
    *reply = arguments.A1 * arguments.A2
    return nil
}

func (t *MyInterface) Power(arguments *sharedRPC.MyFloats, reply *float64) error {
    *reply = Power(arguments.A1, arguments.A2)
    return nil
}
```

В этом коде Go RPC-сервер реализует требуемый интерфейс, а также вспомогательную функцию `Power()`.

Третий фрагмент `RPCserver.go`:

```
func main() {
    PORT := ":1234"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
}
```

Четвертая часть `RPCserver.go` содержит следующий код:

```
myInterface := new(MyInterface)
rpc.Register(myInterface)
t, err := net.ResolveTCPAddr("tcp4", PORT)
if err != nil {
    fmt.Println(err)
    return
}
l, err := net.ListenTCP("tcp4", t)
if err != nil {
    fmt.Println(err)
    return
}
```

Именно применение функции `rpc.Register()` делает эту программу RPC-сервером. Однако, поскольку RPC-сервер использует протокол TCP, ему все равно необходимо вызывать функции `net.ResolveTCPAddr()` и `net.ListenTCP()`.

Остальная часть кода Go из файла `RPCclient.go` выглядит так:

```
for {
    c, err := l.Accept()
    if err != nil {
        continue
    }
    fmt.Printf("%s\n", c.RemoteAddr())
    rpc.ServeConn(c)
}
}
```

Функция `RemoteAddr()` возвращает IP-адрес и номер порта, которые необходимы для связи с RPC-клиентом. Функция `rpc.ServeConn()` обслуживает RPC-клиента.

Выполнение `RPCserver.go` с ожиданием `RPCclient.go` приведет к следующим результатам:

```
$ go run RPCserver.go
127.0.0.1:52289
```

При выполнении `RPCclient.go` получим такой результат:

```
$ go run RPCclient.go localhost:1234
Reply (Multiply): -8.000000
Reply (Power): 0.250000
```

Низкоуровневое сетевое программирование

Структура `http.Transport` дает возможность изменять различные низкоуровневые параметры сетевого соединения, однако мы можем написать код Go, который позволит читать необработанные данные сетевых пакетов.

Здесь есть два момента. Во-первых, сетевые пакеты поступают в двоичном формате, так что нам придется не просто принимать все пакеты подряд, а отслеживать определенные типы сетевых пакетов. Проще говоря, при чтении сетевых пакетов необходимо заранее указать один или несколько протоколов, которые поддерживает приложение. Во-вторых, чтобы отправить сетевой пакет, нам придется создать его самостоятельно.

Следующая утилита, которую мы рассмотрим, называется `lowLevel.go`. Разделим ее на три части. Обратите внимание, что `lowLevel.go` перехватывает пакеты интернет-протокола управления сообщениями *ICMP* (Internet Control Message Protocol), которые используют протокол IPv4, и выводит их содержимое. Также учтите, что в целях безопасности для доступа к необработанным сетевым данным необходимы привилегии пользователя `root`.

Первый фрагмент `lowLevel.go` выглядит так:

```
package main

import (
    "fmt"
    "net"
)
```

Во второй части `lowLevel.go` содержится следующий код Go:

```
func main() {
    netaddr, err := net.ResolveIPAddr("ip4", "127.0.0.1")
    if err != nil {
        fmt.Println(err)
        return
    }
    conn, err := net.ListenIP("ip4:icmp", netaddr)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Протокол ICMP указан во второй части первого аргумента (`ip4: icmp`) функции `net.ListenIP()`. Более того, часть `ip4` сообщает утилите о том, что следует перехватывать только трафик IPv4.

Остальная часть `lowLevel.go` содержит следующий код Go:

```
buffer := make([]byte, 1024)
n, _, err := conn.ReadFrom(buffer)
if err != nil {
    fmt.Println(err)
    return
}

fmt.Printf("% X\n", buffer[0:n])
}
```

Согласно этому коду `lowLevel.go` читает только один сетевой пакет, поскольку здесь нет цикла `for`.

Протокол ICMP используется в утилитах `ping(1)` и `traceroute(1)`, поэтому один из способов создания ICMP-трафика — воспользоваться одним из этих инструментов. При запущенной утилите `lowLevel.go` сетевой трафик на обеих машинах UNIX будет генерироваться с помощью следующих команд:

```
$ ping -c 5 localhost
PING localhost (127.0.0.1): 56 data
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.037 ms
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.117 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.049 ms
--- localhost ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.037/0.059/0.117/0.030 ms
$ traceroute localhost
traceroute to localhost (127.0.0.1), 64 hops max, 52 byte packets
1 localhost (127.0.0.1) 0.255 ms 0.048 ms 0.067 ms
```

Выполнение `lowLevel.go` на компьютере с macOS Mojave с привилегиями пользователя `root` приведет к таким результатам:

```
$ sudo go run lowLevel.go
03 03 CD DA 00 00 00 00 45 00 34 00 B4 0F 00 00 01 11 00 00 7F 00 00 01 7F
00 00 01 B4 0E 82 9B 00 20 00 00
$ sudo go run lowLevel.go
00 00 0B 3B 20 34 00 00 5A CB 5C 15 00 04 32 A9 08 09 0A 0B 0C 0D 0E 0F 10
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29
2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
```

Первый пример вывода генерируется командой `ping(1)`, а второй — `traceroute(1)`.

Если запустить `lowLevel.go` на компьютере с Debian Linux, то получим следующий результат:

```
$ uname -a
Linux mail 4.14.12-x86_64-linode92 #1 SMP Fri Jan 5 15:34:44 UTC 2018
x86_64 GNU/Linux
# go run lowLevel.go
08 00 61 DD 3F BA 00 01 9A 5D CB 5A 00 00 00 00 26 DC 0B 00 00 00 00 00 10
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29
2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
# go run lowLevel.go
03 03 BB B8 00 00 00 00 45 00 00 3C CD 8D 00 00 01 11 EE 21 7F 00 00 01 7F
00 00 01 CB 40 82 9A 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
```

Результатом выполнения команды `uname(1)` является полезная информация об операционной системе Linux. Обратите внимание, что на современных машинах с Linux, чтобы использовать протокол IPv4, нужно выполнить команду `ping(1)` с флагом `-4`.

Получение необработанных сетевых данных ICMP

В этом подразделе вы узнаете, как с помощью пакета `syscall` перехватывать необработанные сетевые данные ICMP и как устанавливать параметры сокета с помощью функции `syscall.SetsockoptInt()`.

Учтите, что передавать необработанные ICMP-данные намного сложнее, поскольку для этого необходимо самостоятельно создавать необработанные сетевые пакеты. Утилита, которую мы рассмотрим, называется `syscallNet.go`. Разделим ее на четыре части.

Первая часть `syscallNet.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "syscall"
)
```

Второй фрагмент `syscallNet.go` содержит следующий код Go:

```
func main() {
    fd, err := syscall.Socket(syscall.AF_INET, syscall.SOCK_RAW,
                             syscall.IPPROTO_ICMP)
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }
}
```

```
f := os.NewFile(uintptr(fd), "captureICMP")
if f == nil {
    fmt.Println("Error in os.NewFile:", err)
    return
}
```

Параметр `syscall.AF_INET` сообщает функции `syscall.Socket()`, что мы будем работать с протоколом IPv4, а параметр `syscall.SOCK_RAW` — что должен быть сгенерирован необработанный сокет. Последний параметр, `syscall.IPPROTO_ICMP`, говорит `syscall.Socket()`, что нас интересует только ICMP-трафик.

Третья часть `syscallNet.go`:

```
err = syscall.SetsockoptInt(fd, syscall.SOL_SOCKET, syscall.SO_RCVBUF, 256)
if err != nil {
    fmt.Println("Error in syscall.Socket:", err)
    return
}
```

Вызов `syscall.SetsockoptInt()` определяет размер буфера приема сокета — 256. Параметр `syscall.SOL_SOCKET` указывает на то, что мы работаем на уровне сокетов. Остальной код Go файла `syscallNet.go` выглядит так:

```
for {
    buf := make([]byte, 1024)
    numRead, err := f.Read(buf)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("% X\n", buf[:numRead])
}
}
```

Благодаря циклу `for` утилита `syscallNet.go` будет продолжать перехват сетевых пакетов ICMP, пока вы не прекратите этот процесс вручную.

Выполнение `syscallNet.go` на компьютере с macOS High Sierra приведет к следующим результатам:

```
$ sudo go run syscallNet.go
45 00 40 00 BC B6 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 3F 36 71
45 00 00 5A CB 6A 90 00 0B 9F 1A 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 62 FB 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 31 EF 71
45 00 01 5A CB 6A 91 00 0B AC 5F 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 9A 5F 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 1D D6 71
45 00 02 5A CB 6A 92 00 0B C0 76 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
```

```

45 00 40 00 6E 0D 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 09 CF 71
45 00 03 5A CB 6A 93 00 0B D4 7B 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 3A 07 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 FE 9C 71
45 00 04 5A CB 6A 94 00 0B DF AB 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 24 00 45 55 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 12 00
00 00 00 45 00 34 00 C5 73 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9B 00 20 00 00
45 00 24 00 E8 1E 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 10 00
00 00 00 45 00 34 00 C5 74 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9C 00 20 00 00
45 00 24 00 2A 4B 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 0E 00
00 00 00 45 00 34 00 C5 75 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9D 00 20 00 00

```

Выполнение `syscallNet.go` на компьютере с Debian Linux приведет к таким результатам:

```

# go run syscallNet.go
45 00 00 54 7F E9 40 00 40 01 BC BD 7F 00 00 01 7F 00 00 01 08 00 6F 07 53
E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 7F EA 00 00 40 01 FC BC 7F 00 00 01 7F 00 00 01 00 00 77 07 53
E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 54 00 00 34 01 8B 8E 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 40 4F 40 00 34 06 74 6A 6D 4A C1 FD 86 77 DC 57 B0 B8
DD 96 00 00 00 00 52 F1 AB DA 50 14 00 00 90 9E 00 00
45 00 00 54 80 4E 40 00 40 01 BC 58 7F 00 00 01 7F 00 00 01 08 00 7E 01 53
E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 4F 00 00 40 01 FC 57 7F 00 00 01 7F 00 00 01 00 00 86 01 53
E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 9B 40 00 40 01 BC 0B 7F 00 00 01 7F 00 00 01 08 00 93 EC 53
E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 9C 00 00 40 01 FC 0A 7F 00 00 01 7F 00 00 01 00 00 9B EC 53
E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 55 00 00 34 01 8B 8D 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00

```

```

00 00 00 45 00 00 28 40 D1 40 00 34 06 73 E8 6D 4A C1 FD 86 77 DC 57 8E 8E
DD 96 00 00 00 00 6C 6E D3 36 50 14 00 00 71 EF 00 00
45 00 00 54 80 F8 40 00 40 01 BB AE 7F 00 00 01 7F 00 00 01 08 00 F2 E7 53
E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 F9 00 00 40 01 FB AD 7F 00 00 01 7F 00 00 01 00 00 FA E7 53
E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 82 0D 40 00 40 01 BA 99 7F 00 00 01 7F 00 00 01 08 00 4A 82 53
E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 82 0E 00 00 40 01 FA 98 7F 00 00 01 7F 00 00 01 00 00 52 82 53
E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 56 00 00 34 01 8B 8C 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 41 74 40 00 34 06 73 45 6D 4A C1 FD 86 77 DC 57 2E 9B
DD 96 00 00 00 00 C3 D6 44 57 50 14 00 00 09 5A 00 00
45 C0 00 44 68 57 00 00 34 01 8B 8B 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 44 27 40 00 33 06 71 92 6D 4A C1 FD 86 77 DC 57 C5 C2
DD 96 00 00 00 00 CF DD DB BE 50 14 00 00 CE C3 00 00
45 C0 00 58 94 B4 00 00 40 01 E7 2E 7F 00 00 01 7F 00 00 01 03 03 F1 DA 00
00 00 00 45 00 00 3C 85 E1 00 00 01 11 35 CE 7F 00 00 01 7F 00 00 01 95 1E
82 9A 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B5 00 00 40 01 E7 2D 7F 00 00 01 7F 00 00 01 03 03 F9 EA 00
00 00 00 45 00 00 3C 85 E2 00 00 01 11 35 CD 7F 00 00 01 7F 00 00 01 8D 0D
82 9B 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B6 00 00 40 01 E7 2C 7F 00 00 01 7F 00 00 01 03 03 D2 EB 00
00 00 00 45 00 00 3C 85 E3 00 00 01 11 35 CC 7F 00 00 01 7F 00 00 01 B4 0B
82 9C 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B7 00 00 40 01 E7 2B 7F 00 00 01 7F 00 00 01 03 03 D6 AC 00
00 00 00 45 00 00 3C 85 E4 00 00 02 11 34 CB 7F 00 00 01 7F 00 00 01 B0 49
82 9D 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B8 00 00 40 01 E7 2A 7F 00 00 01 7F 00 00 01 03 03 F1 B4 00
00 00 00 45 00 00 3C 85 E5 00 00 02 11 34 CA 7F 00 00 01 7F 00 00 01 95 40
82 9E 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B9 00 00 40 01 E7 29 7F 00 00 01 7F 00 00 01 03 03 CD 43 00
00 00 00 45 00 00 3C 85 E6 00 00 02 11 34 C9 7F 00 00 01 7F 00 00 01 B9 B0
82 9F 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BA 00 00 40 01 E7 28 7F 00 00 01 7F 00 00 01 03 03 9D 8F 00
00 00 00 45 00 00 3C 85 E7 00 00 03 11 33 C8 7F 00 00 01 7F 00 00 01 E9 63

```

```

82 A0 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BB 00 00 40 01 E7 27 7F 00 00 01 7F 00 00 01 03 03 A3 13 00
00 00 00 45 00 00 3C 85 E8 00 00 03 11 33 C7 7F 00 00 01 7F 00 00 01 E3 DE
82 A1 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BC 00 00 40 01 E7 26 7F 00 00 01 7F 00 00 01 03 03 D4 66 00
00 00 00 45 00 00 3C 85 E9 00 00 03 11 33 C6 7F 00 00 01 7F 00 00 01 B2 8A
82 A2 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BD 00 00 40 01 E7 25 7F 00 00 01 7F 00 00 01 03 03 A6 8D 00
00 00 00 45 00 00 3C 85 EA 00 00 04 11 32 C5 7F 00 00 01 7F 00 00 01 E0 62
82 A3 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BE 00 00 40 01 E7 24 7F 00 00 01 7F 00 00 01 03 03 F1 C6 00
00 00 00 45 00 00 3C 85 EB 00 00 04 11 32 C4 7F 00 00 01 7F 00 00 01 95 28
82 A4 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BF 00 00 40 01 E7 23 7F 00 00 01 7F 00 00 01 03 03 A3 FE 00
00 00 00 45 00 00 3C 85 EC 00 00 04 11 32 C3 7F 00 00 01 7F 00 00 01 E2 EF
82 A5 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C0 00 00 40 01 E7 22 7F 00 00 01 7F 00 00 01 03 03 B9 AA 00
00 00 00 45 00 00 3C 85 ED 00 00 05 11 31 C2 7F 00 00 01 7F 00 00 01 CD 42
82 A6 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C1 00 00 40 01 E7 21 7F 00 00 01 7F 00 00 01 03 03 B3 B7 00
00 00 00 45 00 00 3C 85 EE 00 00 05 11 31 C1 7F 00 00 01 7F 00 00 01 D3 34
82 A7 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C2 00 00 40 01 E7 20 7F 00 00 01 7F 00 00 01 03 03 F2 62 00
00 00 00 45 00 00 3C 85 EF 00 00 05 11 31 C0 7F 00 00 01 7F 00 00 01 94 88
82 A8 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C3 00 00 40 01 E7 1F 7F 00 00 01 7F 00 00 01 03 03 DD BE 00
00 00 00 45 00 00 3C 85 F0 00 00 06 11 30 BF 7F 00 00 01 7F 00 00 01 A9 2B
82 A9 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

```

Дополнительные ресурсы

Посмотрите следующие ресурсы:

- ❑ посетите страницу документации стандартного Go-пакета `net`, расположенную по адресу <https://golang.org/pkg/net/>. Это одна из самых больших страниц документации Go;
- ❑ чтобы больше узнать о пакете `crypto/tls` из стандартной библиотеки Go, посетите страницу <https://golang.org/pkg/crypto/tls/>;

- ❑ чтобы больше узнать о пакете `crypto/x509`, посетите страницу <https://golang.org/pkg/crypto/x509/>;
- ❑ определение протокола ICMP для IPv4 содержится в RFC 792. Его описание есть на многих ресурсах, включая <https://tools.ietf.org/html/rfc792>;
- ❑ *WebSocket* — это протокол двусторонней связи между клиентом и удаленным хостом. Реализация WebSocket для Go размещена по адресу <https://github.com/gorilla/websocket>. Чтобы узнать больше о WebSocket, посетите страницу <http://www.rfc-editor.org/rfc/rfc6455.txt>;
- ❑ если вы действительно увлекаетесь сетевым программированием и хотите работать с необработанными TCP-пакетами, то вы найдете интересную и полезную информацию и инструменты в библиотеке `gopacket` по адресу <https://github.com/google/gopacket>;
- ❑ пакет `raw`, расположенный по адресу <https://github.com/mdlayher/raw>, позволяет читать и записывать данные на уровне драйвера сетевого устройства.

Упражнения

- ❑ Разработайте на Go клиент для *протокола передачи файлов* (File Transfer Protocol, FTP).
- ❑ Теперь разработайте на Go FTP-сервер. Что сложнее реализовать: FTP-клиент или FTP-сервер? Почему?
- ❑ Попробуйте реализовать Go-версию утилиты `nc` (1). Секрет программирования таких довольно сложных утилит состоит в том, чтобы начать с версии, которая реализует основные функциональные возможности утилиты, и только потом стремиться поддерживать все возможные варианты.
- ❑ Измените утилиту `TCPserver.go` таким образом, чтобы она возвращала в одном сетевом пакете дату, а в другом — время.
- ❑ Измените утилиту `TCPserver.go` таким образом, чтобы она могла последовательно обслуживать несколько клиентов. Обратите внимание, что это не то же самое, что возможность одновременного обслуживания нескольких запросов. Проще говоря, используйте цикл `for`, чтобы вызов `Accept()` выполнялся несколько раз.
- ❑ TCP-серверы, такие как `fiboTCP.go`, имеют тенденцию завершаться при получении определенного сигнала, поэтому добавьте в `fiboTCP.go` код обработки сигналов, как было показано в главе 8.
- ❑ Измените код `kvTCP.go` таким образом, чтобы защитить функцию `save()` с помощью `sync.Mutex`. Насколько это необходимо?
- ❑ Подключите `https.go` к `TLSserver.go` с помощью любого веб-браузера.

- ❑ Попробуйте поместить утилиту `https.go` в образ Docker и использовать ее оттуда.
- ❑ Напишите на Go собственный небольшой веб-сервер, применяя вместо функции `http.ListenAndServe()` простую реализацию TCP.

Резюме

В этой главе рассказано о многих интересных возможностях, включая разработку клиентов и серверов UDP и TCP, которые являются приложениями, работающими в компьютерных сетях TCP/IP.

Следующая, последняя, глава посвящена использованию Go для машинного обучения. В нее войдут такие темы, как регрессия, классификация, обнаружение аномалий и нейронные сети.

14 Машинное обучение на Go

В предыдущих двух главах рассмотрены темы, связанные с сетевым программированием, TCP/IP, HTTPS, RPC и пакетом `net`. В этой главе речь пойдет о *машинном обучении* на Go, включая такие темы, как вычисление статистических показателей, классификация, регрессия, кластеризация, обнаружение аномалий, нейронные сети, анализ выбросов и работа с Apache Kafka. Однако, поскольку темы очень объемные и заслуживают отдельной книги, в данной главе они затронуты лишь поверхностно, дано краткое представление о них, показаны некоторые удобные пакеты Go, которые помогут вам в решении этих задач.

Обратите внимание, что в основе каждой технологии машинного обучения лежит своя теория — знание теории, параметров и ограничений технологий, которые вы намерены использовать, имеет важное значение для достижения успеха в работе. Кроме того, время от времени полезно визуализировать получаемые данные, так как это позволяет быстро получить хорошее представление о них.



Если вы действительно интересуетесь машинным обучением на Go, советую начать с книги *Machine Learning with Go* Дэниела Уайтнека (Daniel Whitenack), изданной Packt Publishing в 2017 году. Если вы хотите изучить теорию, лежащую в основе машинного обучения, то обратитесь к книге *An Introduction to Statistical Learning* Гэреда Джеймса (Gareth James), Даниелы Уиттен (Daniela Witten), Тренора Хаста (Trevor Hastie) и Роберта Тибширани (Robert Tibshirani), выпущенной издательством Springer в 2013 году, и книге *The Elements of Statistical Learning*, 2nd Edition Тренора Хаста, Роберта Тибширани и Джерома Фридмана (Jerome Friedman), выпущенной Springer в 2009 году.

В этой главе рассмотрены следующие темы:

- ☐ вычисление простых статистических свойств;
- ☐ регрессия;
- ☐ классификация;

- ❑ выявление аномалий;
- ❑ кластеризация;
- ❑ нейронные сети;
- ❑ работа с TensorFlow;
- ❑ анализ выбросов;
- ❑ работа с Apache Kafka.

Вычисление простых статистических показателей

Статистика — это область математики, которая занимается сбором, анализом, интерпретацией, упорядочением и представлением данных. Делится на две основные области: описательную статистику, которая стремится описать уже существующую группу значений, и логическую статистику, которая пытается предсказать будущие значения на основе информации, присутствующей в текущем наборе значений.

Статистическое обучение — это раздел прикладной статистики, которая связана с машинным обучением. *Машинное обучение* тесно связано с вычислительной статистикой, является областью компьютерных наук, которая пытается строить обучение на основе данных и делать прогнозы без специального программирования.



Статистические модели пытаются интерпретировать данные максимально точно. Однако точность модели может зависеть от внешних факторов, влияющих на данные. Например, можно построить модель прогнозирования погоды, которая может стать совершенно неточной, если рядом случится ураган.

В этом разделе вы научитесь вычислять основные статистические показатели, такие как среднее, минимальное и максимальное значения выборки, *медианное* значение и *дисперсия* выборки. Эти значения позволяют, не вдаваясь в подробности, составить хорошее представление о данных. Однако общие показатели, которые описывают ваш набор данных, могут легко ввести в заблуждение, заставив вас поверить, что вы хорошо знаете свои данные, в то время как на самом деле это не так.

Мы вычислим все эти статистические показатели в программе `stats.go`. Рассмотрим ее, разделив на пять частей. Каждая строка входного файла содержит одно число, так что входной файл читается построчно. Ошибочный ввод будет игнорироваться без каких-либо предупреждающих сообщений.

Обратите внимание, что входные данные будут храниться в срезе, чтобы можно было использовать отдельную функцию для вычисления каждого показателя. Как вы вскоре увидите, значения среза перед обработкой отсортировываются.

Первая часть `stats.go` выглядит так:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)

func min(x []float64) float64 {
    return x[0]
}

func max(x []float64) float64 {
    return x[len(x)-1]
}
```

Поскольку срез отсортирован, в нем легко найти минимальное и максимальное значения. Но если элементы среза не являются числовыми, может потребоваться вычислить минимальное и максимальное значения другим способом, предназначенным для ваших данных.

Во второй части `stats.go` содержится следующий код Go:

```
func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

Эта функция вычисляет среднее значение числовых данных.

Третья часть `stats.go` выглядит так:

```
func medianValue(x []float64) float64 {
    length := len(x)
    if length%2 == 1 {
        // четные
        return x[(length-1)/2]
    }
```

```

    } else {
        // нечетные
        return (x[length/2] + x[(length/2)-1]) / 2
    }
    return 0
}

func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}

```

Чтобы вычислить медианное значение для набора данных, этот набор данных должен быть отсортирован.

Четвертая часть `stats.go` содержит следующий код:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: stats filename\n")
        return
    }

    data := make([]float64, 0)

    file := flag.Args()[0]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

```

И последняя часть `stats.go`:

```

r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
    line = strings.TrimRight(line, "\r\n")
    value, err := strconv.ParseFloat(line, 64)
    if err == nil {

```

```

        data = append(data, value)
    }
}
sort.Float64s(data)

fmt.Println("Min:", min(data))
fmt.Println("Max:", max(data))
fmt.Println("Mean:", meanValue(data))
fmt.Println("Median:", medianValue(data))
fmt.Println("Variance:", variance(data))
fmt.Println("Standard Deviation:", math.Sqrt(variance(data)))
}

```

Здесь мы начинаем читать входной файл. Есть много способов получить исходные данные — вы всегда можете вернуться к главе 8, чтобы освежить знания о работе с файлами в Go.

Обратите внимание, что, хотя сортировать срез с данными перед их обработкой не обязательно, при некоторых вычислениях это экономит время, поэтому такая сортировка выполняется в функции `main()`.

Последний блок операторов `fmt.Println()` выводит вычисленные статистические показатели на экран.

Выполнение `stats.go` приведет к таким результатам:

```

$ go run stats.go data.txt
Min: -2
Max: 3
Mean: 1.04
Median: 1.2
Variance: 2.8064
Standard Deviation: 1.6752313273097539

```

Содержимое файла `data.txt` будет таким:

```

$ cat data.txt
1.2
-2
1
2.0
not valid
3

```

Как видите, каждая строка содержит одно значение, следовательно, мы работаем с самым простым набором данных (из одного столбца). Манипулирование более сложными данными может немного отличаться, но общая идея остается неизменной.

В программе, представленной в этом разделе, все статистические показатели вычислялись без помощи внешних пакетов Go, предназначенных для статистических вычислений. Большая часть кода Go, который мы рассмотрим дальше, использует существующие пакеты Go, вместо того чтобы реализовывать все с нуля.

Регрессия

Регрессия — это статистический метод вычисления отношений между переменными. В этом разделе реализована *линейная регрессия* — самый популярный и простой из методов регрессии, а также очень хороший способ изучить данные. Обратите внимание, что методы регрессии не являются абсолютно точными, даже если использовать полиномы высшего порядка (нелинейные). Главное в регрессии, как и в большинстве методов машинного обучения, — выбрать не идеальные, а лишь достаточно хорошие метод и модель.

Линейная регрессия

Идея линейной регрессии проста: мы пытаемся построить модель данных, используя уравнение первой степени. Уравнение первой степени можно представить как $y = a x + b$.

Существует много способов, позволяющих построить уравнение первой степени, которое моделирует данные. Все эти методы предназначены для вычисления a и b .

Реализация линейной регрессии

Код Go, который мы рассмотрим в этом разделе, сохранен в файле `regression.go`. Разделим его на три части. Результат программы — два числа с плавающей точкой, которые соответствуют параметрам a и b в уравнении первой степени.

В первой части `regression.go` содержится следующий код:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/gonum/stat"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}
```

Структура `xy` используется для хранения данных; ее конкретный вид зависит от формата и значений исходных данных.

Вторая часть `regression.go` выглядит так:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: regression filename\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    records, err := r.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
    size := len(records)

    data := xy{
        x: make([]float64, size),
        y: make([]float64, size),
    }
}
```

Последняя часть `regression.go`:

```
for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements")
        continue
    }

    if s, err := strconv.ParseFloat(v[0], 64); err == nil {
        data.y[i] = s
    }

    if s, err := strconv.ParseFloat(v[1], 64); err == nil {
        data.x[i] = s
    }
}

b, a := stat.LinearRegression(data.x, data.y, nil, false)
fmt.Printf("%.4v x + %.4v\n", a, b)
fmt.Printf("a = %.4v b = %.4v\n", a, b)
}
```

Содержимое файла исходных данных считывается в переменную `data`. Функция, которая реализует линейную регрессию, называется `stat.LinearRegression()`. Она возвращает два числа, которые являются значениями `b` и `a`, и именно в такой последовательности.

Теперь самое время загрузить пакет `gonum`:

```
$ go get -u gonum.org/v1/gonum/stat
```

Выполнение `regression.go` с исходными данными, хранящимися в файле `reg_data.txt`, приведет к таким результатам:

```
$ go run regression.go reg_data.txt
0.9463 x + -0.3985
a = 0.9463 b = -0.3985
```

Программа возвращает два числа, которые являются значениями `a` и `b` из формулы $y = a x + b$.

Содержимое `reg_data.txt` выглядит так:

```
$ cat reg_data.txt
1,2
3,4.0
2.1,3
4,4.2
5,5.1
-5,-5.1
```

Вывод данных

Теперь настало время вывести результаты и набор данных, чтобы проверить, насколько точны результаты метода линейной регрессии. Для этого мы воспользуемся кодом Go из файла `plotLR.go`, который представлен в четырех частях. Программа `plotLR.go` принимает три аргумента командной строки: значения `a` и `b` из формулы $y = a x + b$, а также имя файла, в котором хранятся точки исходных данных. То обстоятельство, что `plotLR.go` не вычисляет `a` и `b`, позволяет экспериментировать с `a` и `b`, используя собственные значения или значения, вычисленные другой утилитой.

Первая часть `plotLR.go` выглядит так:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
```

```

    "image/color"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}

func (d xy) Len() int {
    return len(d.x)
}

func (d xy) XY(i int) (x, y float64) {
    x = d.x[i]
    y = d.y[i]
    return
}

```

Методы `Len()` и `XY()` необходимы для построения графика, а пакет `image/color` — для выбора цветов при выводе результатов.

Во второй части `plotLR.go` содержится следующий код:

```

func main() {
    flag.Parse()
    if len(flag.Args()) < 3 {
        fmt.Printf("usage: plotLR filename a b\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    a, err := strconv.ParseFloat(flag.Args()[1], 64)
    if err != nil {
        fmt.Println(a, "not a valid float!")
        return
    }

    b, err := strconv.ParseFloat(flag.Args()[2], 64)
    if err != nil {
        fmt.Println(b, "not a valid float!")
        return
    }
}

```



```

records, err := r.ReadAll()
if err != nil {
    fmt.Println(err)
    return
}

```

Эта часть программы получает аргументы командной строки и выполняет чтение данных.

Третья часть `plotLR.go`:

```

size := len(records)

data := xy{
    x: make([]float64, size),
    y: make([]float64, size),
}

for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements per line!")
        return
    }

    s, err := strconv.ParseFloat(v[0], 64)
    if err == nil {
        data.y[i] = s
    }

    s, err = strconv.ParseFloat(v[1], 64)
    if err == nil {
        data.x[i] = s
    }
}

```

Последняя часть `plotLR.go` выглядит так:

```

line := plotter.NewFunction(func(x float64) float64 { return a*x + b })
line.Color = color.RGBA{B: 255, A: 255}

p, err := plot.New()
if err != nil {
    fmt.Println(err)
    return
}

plotter.DefaultLineStyle.Width = vg.Points(1)
plotter.DefaultGlyphStyle.Radius = vg.Points(2)

scatter, err := plotter.NewScatter(data)
if err != nil {
    fmt.Println(err)
}

```

```

        return
    }
    scatter.GlyphStyle.Color = color.RGBA{R: 255, B: 128, A: 255}

    p.Add(scatter, line)

    w, err := p.WriterTo(300, 300, "svg")
    if err != nil {
        fmt.Println(err)
        return
    }

    _, err = w.WriteTo(os.Stdout)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

Функция, график которой мы построим, определяется с помощью метода `plotter.NewFunction()`.

Теперь нам нужно загрузить несколько внешних пакетов, выполнив следующие команды:

```

$ go get -u gonum.org/v1/plot
$ go get -u gonum.org/v1/plot/plotter
$ go get -u gonum.org/v1/plot/vg

```

Выполнение `plotLR.go` приведет к таким результатам:

```

$ go run plotLR.go reg_data.txt
usage: plotLR filename a b
$ go run plotLR.go reg_data.txt 0.9463 -0.3985
<?xml version="1.0"?>
<!-- Generated by SVGGo and Plotinum VG -->
<svg width="300pt" height="300pt" viewBox="0 0 300 300"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
<g transform="scale(1, -1) translate(0, -300)">
.
.
.

```

Таким образом, прежде чем выводить вычисленные данные в виде графика, нам нужно сохранить их в файле:

```

$ go run plotLR.go reg_data.txt 0.9463 -0.3985 > output.svg

```

Поскольку выходные данные представлены в формате *Scalable Vector Graphics* (SVG), то, чтобы увидеть график, необходимо загрузить их в браузер. Результаты показаны на рис. 14.1.

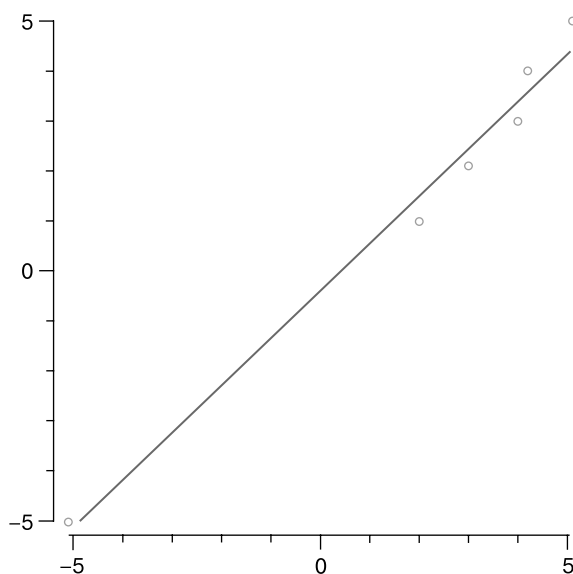


Рис. 14.1. Результат работы программы plotLR.go

На этом графике также видно, насколько точно линейное уравнение способно смоделировать исходные данные.

Классификация

В статистике и машинном обучении *классификация* — это процесс помещения элементов в predetermined наборы, которые называются категориями. В машинном обучении классификация относится к технологиям *обучения с учителем* (supervised learning), при котором перед работой с реальными данными происходит обучение программы на множестве, которое, как считается, содержит правильно классифицированные данные.

Мы рассмотрим очень популярный и простой в реализации метод классификации — *метод k ближайших соседей* (k-NN). Идея, лежащая в основе метода k-NN, заключается в том, что мы можем классифицировать элементы данных на основе их сходства с другими элементами. Буква k в записи k-NN обозначает количество соседей, которые используются при принятии решения. Таким образом, k является положительным целым числом, обычно довольно небольшим.

Входными данными для алгоритма k ближайших соседей являются обучающие примеры в пространстве признаков. Объекту присваивается категория большинством голосов его соседей, причем объект присваивается классу, который является наиболее распространенным среди его k-NN. Если значение k равно 1,

то элемент просто присваивается классу, который является ближайшим соседом, в соответствии с используемой *метрикой расстояния* (distance metric). Метрика расстояния зависит от того, с какими данными мы имеем дело. Например, для работы с комплексными числами потребуется одна метрика расстояния, а для точек в трехмерном пространстве — другая.

Код, который иллюстрирует классификацию в Go, хранится в файле `classify.go`. Рассмотрим его, разделив на три части.

Первая часть `classify.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "strconv"

    "github.com/sjwhitworth/golearn/base"
    "github.com/sjwhitworth/golearn/evaluation"
    "github.com/sjwhitworth/golearn/knn"
)
```

Во второй части `classify.go` содержится следующий код Go:

```
func main() {
    flag.Parse()
    if len(flag.Args()) < 2 {
        fmt.Printf("usage: classify filename k\n")
        return
    }

    dataset := flag.Args()[0]
    rawData, err := base.ParseCSVToInstances(dataset, false)
    if err != nil {
        fmt.Println(err)
        return
    }

    k, err := strconv.Atoi(flag.Args()[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    cls := knn.NewKnnClassifier("euclidean", "linear", k)
```

Метод `knn.NewKnnClassifier()` возвращает новый *классификатор*. Последний параметр функции — это число соседей, которое должен иметь классификатор.

Последняя часть `classify.go` выглядит так:

```
train, test := base.InstancesTrainTestSplit(rawData, 0.50)
cls.Fit(train)
```

```

    p, err := cls.Predict(test)
    if err != nil {
        fmt.Println(err)
        return
    }

    confusionMat, err := evaluation.GetConfusionMatrix(test, p)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(evaluation.GetSummary(confusionMat))
}

```

Метод `Fit()` сохраняет обучающие данные на тот случай, если вы захотите использовать их позже, а метод `Predict()` возвращает классификацию входных данных на основе обучающих данных с помощью алгоритма k-NN. Наконец, метод `evaluation.GetSummary()` возвращает таблицу точности, отклик модели, истинные положительные, ложные положительные и истинные отрицательные значения каждого класса для заданного значения `ConfusionMatrix`, которое вычислено с помощью функции `evaluation.GetConfusionMatrix()`.



Поскольку `base.InstancesTrainTestSplit()` не будет все время возвращать одни и те же значения, при каждом выполнении `classify.go` обучающие и тестовые данные будут разными. Следовательно, вы будете получать разные результаты.

Далее нам нужно установить пакет `github.com/sjwhitworth/golearn`:

```

$ go get -t -u -v github.com/sjwhitworth/golearn
github.com/sjwhitworth/golearn (download)
github.com/sjwhitworth/golearn
$ cd ~/go/src/github.com/sjwhitworth/golearn
$ go get -t -u -v ./...

```

Теперь вернемся в предыдущий каталог, чтобы запустить оттуда программу `classify.go`. Выполнение `classify.go` приведет к таким результатам:

```

$ go run classify.go class_data.txt 2
Reference  Classn True Positives False Positives True Negatives Precision Recall
F1 Score
-----
-----
Iris-versicolor      25      0      41      1.0000      0.9259      0.9615
Iris-virginica       5       2      61      0.7143      1.0000      0.8333
Iris-setosa          36      0      32      1.0000      1.0000      1.0000

```

Overall accuracy: 0.9706

```
$ go run classify.go class_data.txt 30
```

Reference	Classn	True Positives	False Positives	True Negatives	Precision	Recall
F1 Score						

Iris-versicolor	27	5	36	0.8438	1.0000	0.9153
Iris-virginica	0	0	63	NaN	0.0000	NaN
Iris-setosa	36	0	32	1.0000	1.0000	1.0000

Overall accuracy: 0.9265

Содержимое файла `class_data.txt` очень простое и имеет следующий формат:

```
$ head -4 class_data.txt
6.7,3.1,5.6,2.4,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
```

Источник — набор данных **Iris**.

Высокая точность алгоритма объясняется тем, что в файле `class_data.txt` очень мало элементов. Поскольку `golearn` включает в себя несколько примеров данных, я попробовал использовать весь набор данных Iris на своем компьютере с macOS Mojave и получил следующий результат:

```
$ go run classify.go
```

```
~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 2
```

Reference	Classn	True Positives	False Positives	True Negatives	Precision	Recall
F1 Score						

Iris-setosa	30	0	58	1.0000	1.0000	1.0000
Iris-virginica	28	3	56	0.9032	0.9655	0.9333
Iris-versicolor	26	1	58	0.9630	0.8966	0.9286

Overall accuracy: 0.9545

```
$ go run classify.go
```

```
~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 50
```

Reference	Classn	True Positives	False Positives	True Negatives	Precision	Recall
F1 Score						

Iris-setosa	0	0	58	NaN	0.0000	NaN
Iris-virginica	4	5	54	0.4444	0.1379	0.2105
Iris-versicolor	24	55	4	0.3038	0.8276	0.4444

Overall accuracy: 0.3182

Если, работая с набором данных Iris, вы пробуете выбрать другое число соседей, то увидите, что чем больше соседей, тем ниже точность результатов. К сожалению, дальнейшее обсуждение алгоритма k-NN выходит за рамки этой книги.

Кластеризация

Кластеризация — это неконтролируемая (unsupervised, без учителя) версия классификации, при которой разделение данных на категории основано на некотором показателе сходства или расстояния. В этом разделе рассмотрена *кластеризация методом k -средних* — самый известный, а также очень легко реализуемый метод кластеризации. Здесь мы снова воспользуемся внешней библиотекой, которая находится по адресу <https://github.com/mash/gokmeans>.

Мы разберем кластеризацию в Go на примере утилиты `cluster.go`. Разделим ее на три части. Эта утилита принимает один аргумент командной строки — число кластеров, которые следует создать.

Первая часть `cluster.go` выглядит так:

```
package main

import (
    "flag"
    "fmt"
    "github.com/mash/gokmeans"
    "strconv"
)

var observations []gokmeans.Node = []gokmeans.Node{
    gokmeans.Node{4},
    gokmeans.Node{5},
    gokmeans.Node{6},
    gokmeans.Node{8},
    gokmeans.Node{10},
    gokmeans.Node{12},
    gokmeans.Node{15},
    gokmeans.Node{0},
    gokmeans.Node{-1},
}
```

Здесь ради простоты данные включены прямо в программу. Однако ничто не мешает вам читать их из одного или нескольких внешних файлов.

Во второй части `cluster.go` содержится следующий код Go:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: cluster k\n")
        return
    }

    k, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
```

```

    fmt.Println(err)
    return
}

```

Последняя часть `cluster.go`:

```

if success, centroids := gokmeans.Train(observations, k, 50); success {
    fmt.Println("The centroids are the following:")
    for _, centroid := range centroids {
        fmt.Println(centroid)
    }
    fmt.Println("The clusters are the following:")
    for _, observation := range observations {
        index := gokmeans.Nearest(observation, centroids)
        fmt.Println(observation, "belongs in cluster", index+1, ".")
    }
}
}

```

Теперь нам нужно скачать внешнюю библиотеку Go, выполнив следующую команду:

```

$ go get -v -u github.com/mash/gokmeans
github.com/mash/gokmeans (download)
github.com/mash/gokmeans

```

Если мы хотим получить только один кластер, то выполнение `cluster.go` приведет к следующим результатам:

```

$ go run cluster.go 1
The centroids are the following:
[6.555555555555555]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 1 .
[10] belongs in cluster 1 .
[12] belongs in cluster 1 .
[15] belongs in cluster 1 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 1 .

```

Поскольку кластер всего один, то все элементы будут принадлежать этому кластеру, и выходные данные `cluster.go` выглядят соответствующим образом.

Если изменить значение `k`, то результат будет таким:

```

$ go run cluster.go 5
The centroids are the following:
[5]
[-0.5]

```



```

[13.5]
[10]
[8]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 5 .
[10] belongs in cluster 4 .
[12] belongs in cluster 3 .
[15] belongs in cluster 3 .
[0] belongs in cluster 2 .
[-1] belongs in cluster 2 .
$ go run cluster.go 8
The centroids are the following:
[0]
[4.5]
[-1]
[9]
[-1]
[6]
[12]
[15]
The clusters are the following:
[4] belongs in cluster 2 .
[5] belongs in cluster 2 .
[6] belongs in cluster 6 .
[8] belongs in cluster 4 .
[10] belongs in cluster 4 .
[12] belongs in cluster 7 .
[15] belongs in cluster 8 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 3 .

```

Выявление аномалий

Методы выявления аномалий стремятся определить вероятность того, что данное множество содержит элементы с аномальным поведением, которые могут иметь необычные значения или закономерности.

Утилита, которую мы разработаем в этом разделе, называется `anomaly.go`. Разделим ее на три части. В этой утилите реализовано вероятностное выявление аномалий с помощью пакета *Anomalyzer*. Вычислим вероятность того, что данный набор числовых значений содержит элементы с аномальным поведением.

Первая часть `anomaly.go` выглядит так:

```

package main

import (
    "flag"

```

```

    "fmt"
    "math/rand"
    "strconv"
    "time"
    "github.com/lytics/anomalyzer"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

```

Вторая часть `anomaly.go` выглядит так:

```

func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: anomaly MAX\n")
        return
    }

    MAX, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
        fmt.Println(err)
        return
    }

    conf := &anomalyzer.AnomalyzerConf{
        Sensitivity: 0.1,
        UpperBound: 5,
        LowerBound: anomalyzer.NA,
        ActiveSize: 1,
        Nseasons: 4,
        Methods: []string{"diff", "fence", "magnitude", "ks"},
    }
}

```

Пакет `anomalyzer` поддерживает алгоритмические тесты `cdf`, `diff`, `high rank`, `low rank`, `magnitude`, `fence` и `bootstrap ks`. Если вы решите использовать некоторые или все из них в качестве значения `Methods`, то `anomalyzer` их выполнит. Поскольку каждый из этих тестов возвращает вероятность аномального поведения, пакет `anomalyzer` будет вычислять средневзвешенное значение, которое покажет, есть ли в данном множестве элементы с аномальным поведением.

Поле `Sensitivity` предназначено для теста `magnitude` и по умолчанию равно `0.1`. Поля `UpperBound` и `LowerBound` используются в тесте `fence`. А вот поле `ActiveSize` является обязательным, и его значение должно быть не менее `1`. Наконец, поле `NSeasons`, если оно не определено, по умолчанию равно `4`.

В третьей части `anomaly.go` содержится следующий код Go:

```

data := []float64{}
SEED := time.Now().Unix()
rand.Seed(SEED)

```

```

for i := 0; i < MAX; i++ {
    data = append(data, float64(random(0, MAX)))
}
fmt.Println("data:", data)

```

Код `anomaly.go` самостоятельно генерирует случайные данные. Количество элементов задается в виде аргумента командной строки программы.

Последняя часть `anomaly.go` выглядит так:

```

anom, _ := analyzer.NewAnalyzer(conf, data)
prob := anom.Push(8.0)
fmt.Println("Anomalous Probability:", prob)
}

```

Теперь нам нужно загрузить внешний пакет Go:

```

$ go get -v -u github.com/lytics/analyzer
github.com/lytics/analyzer (download)
github.com/drewlanenga/govector (download)
github.com/drewlanenga/govector
github.com/lytics/analyzer

```

Выполнение `anomaly.go` приведет к таким результатам:

```

$ go run anomaly.go 20
data: [18 3 2 19 2 16 5 15 3 14 2 9 11 10 2 17 17 14 19 1]
Anomalous Probability: 0.8612730015082957
$ go run anomaly.go 20
data: [17 8 19 10 0 14 12 7 7 13 2 5 18 1 15 4 0 14 13 9]
Anomalous Probability: 0.7885470085470085
$ go run anomaly.go 100
data: [85 5 64 32 69 55 0 67 11 96 75 92 25 54 2 49 58 6 16 38 55 11 93 90
90 47 66 97 37 61 85 92 15 45 33 43 61 44 73 18 10 86 17 15 67 28 26 7 25
76 79 51 9 32 70 99 9 39 6 25 10 57 50 84 20 67 42 89 0 1 8 96 49 6 20 33
57 18 48 84 53 98 51 84 41 97 69 62 11 44 21 13 90 25 52 85 48 27 90 20]
Anomalous Probability: 0.8977395577395577

```

Нейронные сети

Нейронные сети стремятся работать подобно человеческому мозгу, учатся решать задачи на основе предоставленных им примеров. Нейронная сеть делится на слои. В самой маленькой нейронной сети должно быть как минимум два слоя: входной и выходной. На этапе обучения данные проходят через все слои нейронной сети. Реальные выходные значения обучающих данных используются для корректировки рассчитанных выходных значений обучающих данных, чтобы следующая итерация была более точной.

Утилита `neural.go`, которую мы разработаем в этом разделе, будет реализовывать простейшую нейронную сеть. Мы разделим ее на четыре части.

Первая часть `neural.go` выглядит так:

```
package main

import (
    "fmt"
    "math/rand"
    "time"

    "github.com/goml/gobrain"
)
```

Новая строка в списке `import` дает инструменту `gofmt` команду сортировать имена пакетов по блокам, разделенным пустыми строками. Во второй части `neural.go` содержится следующий код Go:

```
func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)
    patterns := [][]float64{
        {{0, 0, 0, 0}, {0}},
        {{0, 1, 0, 1}, {1}},
        {{1, 0, 1, 0}, {1}},
        {{1, 1, 1, 1}, {1}},
    }
}
```

Срез `patterns` содержит обучающие данные, которые будут использованы позже. Функция `rand.Seed()` создает и инициализирует генератор случайных чисел, который автоматически используется пакетом `github.com/goml/gobrain`.

Третья часть `neural.go` выглядит так:

```
ff := &gobrain.FeedForward{}
ff.Init(4, 2, 1)
ff.Train(patterns, 1000, 0.6, 0.4, false)
```

Здесь инициализируется нейронная сеть. Первый параметр метода `Init()` – это количество входов сети, второй – количество скрытых узлов, а третий – количество выходов. Измерение и данные среза `patterns` должны соответствовать значениям, использованным в `Init()`, и наоборот.

Последняя часть `neural.go` выглядит так:

```
in := []float64{1, 1, 0, 1}
out := ff.Update(in)
fmt.Println(out)

in = []float64{0, 0, 0, 0}
out = ff.Update(in)
fmt.Println(out)
}
```

Здесь выполняется два теста. Для первого теста используются входные данные $\{1, 1, 0, 1\}$, а для второго — $\{0, 0, 0, 0\}$.

Теперь вы уже знаете, какой пакет Go следует загрузить с помощью команды `go get`, поэтому скачайте его, прежде чем пытаться запустить `neural.go`.

Выполнение `neural.go` приведет к таким результатам:

```
$ go run neural.go
[0.9918648920317314]
[0.02826477691747802]
```

Первое значение приближается к 1, тогда как второе — к 0.

Поскольку здесь присутствует элемент случайности, результаты нескольких запусков `neural.go` будут незначительно различаться:

```
$ go run neural.go
[0.9920127780655835]
[0.028029429851140687]
go run neural.go
[0.9913803776914417]
[0.028875009295811015]
```

Анализ выбросов

Анализ выбросов — это поиск значений, которые выглядят так, как будто они не имеют отношения к остальным значениям. Проще говоря, выбросы — это экстремальные значения, которые сильно отличаются от остальных наблюдений. Вот хорошая книга по анализу выбросов: *Outlier Analysis, 2nd Edition*, автор Хару С. Аггарвал (Charu C. Aggarwal), издательство Springer, 2017 год.

Мы рассмотрим метод анализа выбросов, основанный на стандартном отклонении. Этот метод реализован в программе `outlier.go`, которую мы разделим на четыре части. Позже вы ближе познакомитесь с этим методом.

Первая часть `outlier.go` выглядит так:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)
```

Метод, который будет реализован, требует только пакетов стандартной библиотеки Go.

Во второй части `outlier.go` содержится следующий код Go:

```
func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}

func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

Эти две функции уже использовались ранее в программе `stats.go`. Если вы постоянно применяете одни и те же функции, удобно создать одну или несколько библиотек Go и сгруппировать в них код Go.

В третьей части `outlier.go` содержится следующий код Go:

```
func outliers(x []float64, limit float64) []float64 {
    deviation := math.Sqrt(variance(x))
    mean := meanValue(x)
    anomaly := deviation * limit
    lower_limit := mean - anomaly
    upper_limit := mean + anomaly
    fmt.Println(lower_limit, upper_limit)

    y := make([]float64, 0)
    for _, val := range x {
        if val < lower_limit || val > upper_limit {
            y = append(y, val)
        }
    }
    return y
}
```

В этой функции реализована вся логика программы. Она рассчитывает стандартное отклонение и среднее значение выборки, чтобы затем вычислить верхний и нижний пределы. Все, что выходит за эти пределы, будет считаться выбросом.

Последняя часть `outlier.go` выглядит так:

```
func main() {
    flag.Parse()
```

```

if len(flag.Args()) != 2 {
    fmt.Printf("usage: stats filename limit\n")
    return
}
file := flag.Args()[0]

f, err := os.Open(file)
if err != nil {
    fmt.Println(err)
    return
}
defer f.Close()

limit, err := strconv.ParseFloat(flag.Args()[1], 64)
if err != nil {
    fmt.Println(err)
    return
}

data := make([]float64, 0)
r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
    line = strings.TrimRight(line, "\r\n")
    value, err := strconv.ParseFloat(line, 64)
    if err == nil {
        data = append(data, value)
    }
}

sort.Float64s(data)
out := outliers(data, limit)
fmt.Println(out)
}

```

Функция `main()` принимает аргументы командной строки и считывает данные из входного файла, после чего вызывает `outliers()`.

Выполнение `outlier.go` приведет к такому результату:

```

$ go run outlier.go data.txt 2
-94.21189713007178 95.36745268562734
[-100 100]
$ go run outlier.go data.txt 5
-236.3964094918461 237.55196504740167

```

```
[ ]
$ go run outlier.go data.txt 0.02
-0.3701189713007176 1.5256745268562737
[-100 -10 -2 2 3 10 100]
```

Если уменьшить значение переменной `limit`, то получим больше выбросов из набора данных.

Однако это зависит от задачи, которую мы пытаемся решить.

В файле `data.txt` содержатся следующие данные:

```
$ cat data.txt
1.2
-2
1
2.0
not valid
3
10
100
-10
-100
```

Работа с TensorFlow

TensorFlow — довольно известная платформа с открытым исходным кодом, предназначенная для машинного обучения. Чтобы использовать TensorFlow в Go, вам сначала нужно скачать следующий пакет Go:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
```

Однако для работы этой команды необходимо предварительно установить C-интерфейс для TensorFlow. На компьютере с macOS Mojave это можно сделать так:

```
$ brew install tensorflow
```

При попытке установить пакет Go для TensorFlow, если C-интерфейс еще не установлен, выведется следующее сообщение об ошибке:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
# github.com/tensorflow/tensorflow/tensorflow/go
ld: library not found for -ltensorflow
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

Поскольку TensorFlow — довольно сложный пакет, не лишним будет проверить правильность его установки с помощью такой команды:

```
$ go test github.com/tensorflow/tensorflow/tensorflow/go
ok github.com/tensorflow/tensorflow/tensorflow/go 0.109s
```


Кроме тестов Go, также можно выполнить следующую программу Go, которая выведет версию используемого Go-пакета TensorFlow:

```
package main

import (
    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
    "fmt"
)

func main() {
    s := op.NewScope()
    c := op.Const(s, "Using TensorFlow version: " + tf.Version())
    graph, err := s.Finalize()

    if err != nil {
        fmt.Println(err)
        return
    }

    sess, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    output, err := sess.Run(nil, []tf.Output{c}, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(output[0].Value())
}
```

Если сохранить эту программу как `tfVersion.go` и выполнить ее, то получим следующий результат:

```
$ go run tfVersion.go
2019-06-10 22:30:12.880532: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports instructions
that this TensorFlow binary was not compiled to use: AVX2 FMA Using TensorFlow
version: 1.13.1
```

Первое сообщение — это предупреждение, сгенерированное Go-пакетом TensorFlow, которое в данном случае можно проигнорировать.

Теперь мы готовы продолжить, и я покажу вам настоящую программу, на примере которой вы увидите, как работает TensorFlow. Код этой программы хранится в файле `tFlow.go`. Разделим его на четыре части. Эта программа выполняет сло-

жение и умножение двух чисел, которые она принимает в качестве аргументов командной строки.

Первая часть `tFlow.go` выглядит так:

```
package main

import (
    "fmt"
    "os"
    "strconv"

    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
)
```

Во второй части `tFlow.go` содержится следующий код:

```
func Add(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("a1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("a2"), tf.Int8)
    sum_result_node := op.Add(sum_scope, input1, input2)

    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
            input1: a1,
```

```

        input2: a2,
    },
    []tf.Output{sum_result_node}, nil)

if err != nil {
    fmt.Println(err)
    return 0, err
}

return sum[0].Value(), nil
}

```

Функция `Add()` складывает два числа типа `int8`. Для простого сложения двух чисел здесь слишком много кода, что показывает, что у TensorFlow есть определенный стиль работы. Так происходит главным образом потому, что TensorFlow является расширенной средой со многими возможностями.

Третья часть `tFlow.go` выглядит так:

```

func Multiply(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("x1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("x2"), tf.Int8)

    sum_result_node := op.Mul(sum_scope, input1, input2)
    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{

```

```

        input1: x1,
        input2: x2,
    },
    []tf.Output{sum_result_node}, nil)

if err != nil {
    fmt.Println(err)
    return 0, err
}

return sum[0].Value(), nil
}

```

Функция `Multiply()` выполняет умножение двух чисел типа `int8` и возвращает результат.

Последняя часть `tFlow.go` выглядит так:

```

func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    t1, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    n1 := int8(t1)

    t2, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
    n2 := int8(t2)

    res, err := Add(n1, n2)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Add:", res)
    }

    res, err = Multiply(n1, n2)
    if err != nil {
        fmt.Println(err)
    } else {
        fmt.Println("Multiply:", res)
    }
}

```

Выполнение `tFlow.go` выдаст такой результат:

```
$ go run tFlow.go 1 20
2019-06-14 18:46:52.115676: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Add: 21
Multiply: 20
$ go run tFlow.go -2 20
2019-06-14 18:47:23.104918: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Add: 18
Multiply: -40
```

Предупреждение, которое появляется на выходе программы, говорит о том, что TensorFlow работает на вашем компьютере не так быстро, как мог бы. По сути, вам сообщают, что для решения проблемы нужно скомпилировать TensorFlow с нуля.

Поговорим о Kafka

В этом разделе вы узнаете, как писать и читать записи в формате JSON с помощью *Kafka*. Эта платформа получила название в честь писателя Франца Кафки. Платформа Kafka написана на *Scala* и *Java*. Первоначально она разрабатывалась компанией LinkedIn, а в 2011 году была передана в дар Apache Software Foundation. На структуру Kafka повлияли журналы транзакций.

Главным преимуществом Kafka является то, что ее можно использовать для быстрого сохранения большого количества данных. Это может заинтересовать тех, кому приходится работать с огромными объемами данных в реальном времени. Недостаток этой платформы — для поддержания высокой скорости обработки данные предназначены только для чтения и сохраняются примитивным способом.

На примере программы `writeKafka.go` разберем, как записывать данные в Kafka. В терминологии Kafka `writeKafka.go` называется *производителем*. Мы рассмотрим эту утилиту, разделив ее на три части.

Первая часть `writeKafka.go` выглядит так:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
```

```

    "github.com/segmentio/kafka-go"
    "math/rand"
    "os"
    "strconv"
    "time"
)

```

Go-драйвер Kafka требует использования внешнего пакета, который нужно загрузить самостоятельно.

Во второй части `writeKafka.go` содержится следующий код Go:

```

type Record struct {
    Name    string `json:"name"`
    Random  int    `json:"random"`
}

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func main() {
    MIN := 0
    MAX := 0
    TOTAL := 0
    topic := ""
    if len(os.Args) > 4 {
        MIN, _ = strconv.Atoi(os.Args[1])
        MAX, _ = strconv.Atoi(os.Args[2])
        TOTAL, _ = strconv.Atoi(os.Args[3])
        topic = os.Args[4]
    } else {
        fmt.Println("Usage:", os.Args[0], "MIN MAX TOTAL TOPIC")
        return
    }
}

```

Структура `Record` используется для хранения данных, которые будут записаны в соответствующий топик Kafka.

Третья часть `writeKafka.go`:

```

partition := 0
conn, err := kafka.DialLeader(context.Background(), "tcp",
    "localhost:9092", topic, partition)
if err != nil {
    fmt.Printf("%s\n", err)
    return
}

rand.Seed(time.Now().Unix())

```

В этой части программы мы определяем адрес сервера Kafka (`localhost: 9092`).

Последняя часть `writeKafka.go` выглядит так:

```
for i := 0; i < TOTAL; i++ {
    myrand := random(MIN, MAX)
    temp := Record{strconv.Itoa(i), myrand}
    recordJSON, _ := json.Marshal(temp)

    conn.SetWriteDeadline(time.Now().Add(1 * time.Second))
    conn.WriteMessages(
        kafka.Message{Value: []byte(recordJSON)},
    )

    if i%50 == 0 {
        fmt.Print(".")
    }
    time.Sleep(10 * time.Millisecond)
}

fmt.Println()
conn.Close()
}
```

Теперь мы рассмотрим программу Go, которая называется `readKafka.go`. На ее примере я покажу, как читать данные из Kafka. Программы, считывающие данные из Kafka, в терминологии Kafka называются *потребителями*. Мы разделим код Go программы `readKafka.go` на четыре части.

Первая часть `readKafka.go` выглядит так:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/segmentio/kafka-go"
    "os"
)
```

Утилите `readKafka.go` также нужен внешний Go-пакет.

Вторая часть `readKafka.go` выглядит так:

```
type Record struct {
    Name    string `json:"name"`
    Random  int    `json:"random"`
}

func main() {
    if len(os.Args) < 2 {
        fmt.Println("Need a Kafka topic name.")
    }
}
```

```

    return
}

partition := 0
topic := os.Args[1]
fmt.Println("Kafka topic:", topic)

```

В этой части программы определяется структура Go, в которой будут храниться записи Kafka. Также здесь обрабатываются аргументы командной строки программы.

В третьей части `readKafka.go` содержится следующий код Go:

```

r := kafka.NewReader(kafka.ReaderConfig{
    Brokers:  []string{"localhost:9092"},
    Topic:    topic,
    Partition: partition,
    MinBytes: 10e3,
    MaxBytes: 10e6,
})
r.SetOffset(0)

```

В структуре `kafka.NewReader()` содержатся данные, необходимые для подключения к процессу сервера Kafka и теме Kafka.

Последний фрагмент кода `readKafka.go` выглядит так:

```

for {
    m, err := r.ReadMessage(context.Background())
    if err != nil {
        break
    }
    fmt.Printf("message at offset %d: %s = %s\n", m.Offset,
        string(m.Key), string(m.Value))
    temp := Record{}
    err = json.Unmarshal(m.Value, &temp)
    if err != nil {
        fmt.Println(err)
    }
    fmt.Printf("%T\n", temp)
}

r.Close()
}

```

Этот цикл `for` считывает данные из заданного топика Kafka и выводит их на экран. Обратите внимание, что программа постоянно ожидает данных и никогда не завершает работу.

В данном примере мы, чтобы выполнить обе программы — `readKafka.go` и `writeKafka.go`, — воспользуемся образом Docker с Kafka. Поэтому прежде всего

нам требуется выполнить следующую команду, чтобы загрузить нужный образ Kafka, если он еще не установлен на нашем локальном компьютере:

```
$ docker pull landoop/fast-data-dev:latest
```

Результаты выполнения команды `docker images` позволят нам убедиться, что на компьютере есть нужный образ Kafka Docker. Затем следует выполнить этот образ и запустить его как *контейнер*:

```
$ docker run --rm --name=kafka-box -it -p 2181:2181 -p 3030:3030 -p
8081:8081 -p 8082:8082 -p 8083:8083 -p 9092:9092 -p 9581:9581 -p 9582:9582
-p 9583:9583 -p 9584:9584 -e ADV_HOST=127.0.0.1 landoop/fast-data-
dev:latest
Setting advertised host to 127.0.0.1.
Starting services.
This is Landoop's fast-data-dev. Kafka 2.0.1-L0 (Landoop's Kafka
Distribution).
You may visit http://127.0.0.1:3030 in about a minute.
.
.
.
```

Для того чтобы все эти утилиты работали, необходимо скачать Go-пакет, который позволяет обмениваться данными с Kafka.

Это можно сделать следующим образом:

```
$ go get -u github.com/segmentio/kafka-go
```

Теперь мы готовы использовать образ Docker Kafka и запустить обе утилиты Go.

Выполнение `writeKafka.go` приведет к таким результатам:

```
$ go run readKafka.go my_topic | head
Kafka topic: my_topic
message at offset 0: = {"name":"0","random":134}
main.Record
message at offset 1: = {"name":"1","random":27}
main.Record
message at offset 2: = {"name":"2","random":168}
main.Record
message at offset 3: = {"name":"3","random":317}
main.Record
message at offset 4: = {"name":"4","random":455}
signal: broken pipe
```

Дополнительные ресурсы

Советую посмотреть следующие ресурсы:

- ☐ чтобы больше узнать о Kafka, посетите страницу <https://kafka.apache.org/>;
- ☐ чтобы больше узнать о TensorFlow, пройдите на страницу <https://www.tensorflow.org/>;

- ❑ *Lenses* — отличный продукт для работы с платформой Kafka и ее записями. Подробнее о Lenses читайте на сайте <https://lenses.io/>;
- ❑ страницу документации Go-пакета TensorFlow вы найдете по адресу <https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go>;
- ❑ чтобы больше узнать о наборе данных Iris, посетите страницу <https://archive.ics.uci.edu/ml/datasets/iris>;
- ❑ на сайте <https://machinebox.io/> вы найдете программное обеспечение для машинного обучения, рассчитанного на тех, у кого нет специальной подготовки. Оно бесплатно предоставляется для разработчиков и функционирует из образов Docker;
- ❑ страницу документации Go-пакета Anomalyzer вы найдете по адресу <https://github.com/lytics/anomalyzer>.

Упражнения

- ❑ Разработайте собственный производитель Kafka, который бы делал записи в топик Kafka в формате JSON с тремя полями.
- ❑ Есть такое очень интересное статистическое свойство — *ковариация*. Найдите его формулу и реализуйте его в Go.
- ❑ Измените код `stats.go` таким образом, чтобы работать только с целочисленными значениями.
- ❑ Измените код `cluster.go` таким образом, чтобы получать данные из внешнего файла, имя которого будет передаваться программе в качестве аргумента командной строки.
- ❑ Измените код `outlier.go` таким образом, чтобы разделить входные данные на два среза и работать с каждым из этих срезов отдельно.
- ❑ Измените код `outlier.go` таким образом, чтобы принимать верхний и нижний пределы от пользователя, а не вычислять их.
- ❑ Если вам трудно использовать TensorFlow, попробуйте `tfgo`, который вы найдете по адресу <https://github.com/galeone/tfgo>.

Резюме

В этой главе рассмотрено несколько интересных тем, связанных с машинным обучением, включая регрессию, обнаружение аномалий, классификацию, кластеризацию и выбросы. Go — отличный, надежный язык программирования для применения в области машинного обучения. Смело используйте его в своих проектах!

Что дальше?

С философской точки зрения ни одна книга по программированию не является идеальной, так что и эта книга не исключение. Я раскрыл не все темы, касающиеся Go? Конечно же! Почему? Потому что тем всегда больше, чем можно уместить в одной книге. Если бы я попытался охватить их все, то эта книга никогда не была бы готова к печати. Ситуация в чем-то аналогична спецификациям программы: в нее всегда можно добавлять новые интересные функции, но если в какой-то момент не заморозить спецификации, то программа всегда будет находиться в состоянии разработки.

Главное, что, прочитав эту книгу, вы будете готовы учиться самостоятельно. Это самое большое преимущество любой хорошей книги по программированию. Основная цель издания — помочь вам научиться программировать на Go и получить некоторый опыт. Однако ничто не заменит самостоятельных упражнений и частых ошибок, потому что единственный способ изучить язык программирования — постоянно разрабатывать нетривиальные вещи. Теперь вы готовы писать собственное программное обеспечение на Go и изучать что-то новое.

Я хочу поздравить вас и поблагодарить за выбор этой книги. Надеюсь, что она была для вас полезной и вы будете и дальше пользоваться ею в качестве справочного пособия. Go — отличный язык программирования, и я уверен, вы не пожалеете о том, что изучили его. Для меня это завершение еще одной книги о Go, но для вас только начало пути!

Soli Deo Gloria — только Богу слава.

Михалис Цукалос

**Golang для профи: работа с сетью, многопоточность,
структуры данных и машинное обучение с Go**

Перевела с английского *Е. Сандицкая*

Заведующая редакцией	<i>Ю. Сергиенко</i>
Руководитель проекта	<i>С. Давид</i>
Ведущий редактор	<i>Н. Гринчик</i>
Научный редактор	<i>Р. Пресняков</i>
Литературный редактор	<i>А. Дубейко</i>
Художественный редактор	<i>В. Мостипан</i>
Корректор	<i>А. Лавровская, Е. Павлович</i>
Верстка	<i>Г. Блинов</i>

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,

Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2020. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 —

Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 27.04.20. Формат 70×100/16. Бумага офсетная. Усл. п. л. 58,050. Тираж 600. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».

142300, Московская область, г. Чехов, ул. Полиграфистов, 1.

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru

Факс: 8(496) 726-54-10, телефон: (495) 988-63-87



КУПИТЬ

Пол Тронкон, Карл Олбинг

BASH И КИБЕРБЕЗОПАСНОСТЬ: АТАКА, ЗАЩИТА И АНАЛИЗ ИЗ КОМАНДНОЙ СТРОКИ LINUX

Командная строка может стать идеальным инструментом для обеспечения кибербезопасности. Невероятная гибкость и абсолютная доступность превращают стандартный интерфейс командной строки (CLI) в фундаментальное решение, если у вас есть соответствующий опыт. Авторы Пол Тронкон и Карл Олбинг рассказывают об инструментах и хитростях командной строки, помогающих собирать данные при упреждающей защите, анализировать логи и отслеживать состояние сетей. Пентестеры узнают, как проводить атаки, используя колоссальный функционал, встроенный практически в любую версию Linux.



КУПИТЬ

Мехди Меджуи, Эрик Уайлд, Ронни Митра, Майк Амундсен

НЕПРЕРЫВНОЕ РАЗВИТИЕ API. ПРАВИЛЬНЫЕ РЕШЕНИЯ В ИЗМЕНЧИВОМ ТЕХНОЛОГИЧЕСКОМ ЛАНДШАФТЕ

Для реализации API необходимо провести большую работу. Чрезмерное планирование может стать пустой тратой сил, а его недостаток приводит к катастрофическим последствиям. В этой книге вы получите решения, которые позволят вам распределить необходимые ресурсы и достичь требуемого уровня эффективности за оптимальное время. Как соблюсти баланс гибкости и производительности, сохранив надежность и простоту настройки? Четыре эксперта из Академии API объясняют разработчикам ПО, руководителям продуктов и проектов, как максимально увеличить ценность их API, управляя интерфейсами как продуктами с непрерывным жизненным циклом.

EXPERT INSIGHT

Mastering Go

Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures

Second Edition



Mihalis Tsoukalos

Packt

Mastering Go

Second Edition

Create Golang production applications using network libraries, concurrency, machine learning, and advanced data structures

Mihalis Tsoukalos



BIRMINGHAM - MUMBAI

Mastering Go

Second Edition

Copyright © 2019 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Acquisition Editor: Andrew Waldron
Acquisition Editor – Peer Reviews: Suresh Jain
Project Editor: Kishor Rit
Development Editor: Joanne Lovell
Technical Editor: Aniket Shetty
Copy Editor: Safis Editing
Proofreader: Safis Editing
Indexer: Pratik Shirodkar
Production Designer: Sandip Tadge

First published: April 2018
Second edition: August 2019

Production reference: 1270819

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-83855-933-5

www.packtpub.com



Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.packt.com and, as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at customercare@packtpub.com for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Mihalis Tsoukalos is a UNIX administrator, a programmer, a DBA, and a mathematician who enjoys writing technical books and articles and learning new things. He is the author of *Go Systems Programming* and *Mastering Go*. He has written more than 250 technical articles for many magazines, including *Sys Admin*, *MacTech*, *Linux User and Developer*, *Usenix ;login:*, *Linux Format*, and *Linux Journal*. His research interests include databases, visualization, statistics and machine learning.

You can reach him at <https://www.mtsoukalos.eu/> and @mactsouk.

Mihalis is also a photographer.

I would like to thank the people at Packt Publishing for helping me to write this book, including my technical reviewer, Mat Ryer, and Kishor Rit for answering all my questions and encouraging me during the whole process.

I would like to dedicate this book to the loving memory of my parents, Ioannis and Argetta.

About the reviewer

Mat Ryer has been programming computers since he was six years old. He would build games and programs, first in BASIC on a ZX Spectrum, and then in AmigaBASIC and AMOS on Commodore Amiga with his father. Many hours were spent manually copying the code from the Amiga Format magazine and tweaking variables or moving GOTO statements around to see what might happen. The same spirit of exploration and obsession with programming led Mat to starting work with a local agency in Mansfield, England, when he was 18, where he started to build websites and other online services.

After several years of working with various technologies and industries in London and around the world, Mat noticed a new systems language called Go that Google was pioneering. Since it addressed very pertinent and relevant modern technical challenges, he started using it to solve problems while the language was still in the beta stage, and he has been using it ever since. Mat contributes to open-source projects and has founded Go packages, including Testify, Moq, Silk, and Is, as well as a macOS developer tool called BitBar.

In 2018, Mat co-founded Machine Box and still spends a lot of time speaking at conferences, writing about Go on his blog, and being an active member of the Go community.

Table of Contents

Chapter 1: Go and the Operating System	8
The history of Go	8
Where is Go going?	9
The advantages of Go	9
Is Go perfect?	10
What is a preprocessor?	11
The godoc utility	11
Compiling Go code	12
Executing Go code	13
Two Go rules	14
You either use a Go package or you do not include it	14
There is only one way to format curly braces	16
Downloading Go packages	17
UNIX stdin, stdout, and stderr	18
About printing output	19
Using standard output	21
Getting user input	23
About := and =	23
Reading from standard input	24
Working with command-line arguments	25
About error output	28
Writing to log files	30
Logging levels	30
Logging facilities	30
Log servers	31
A Go program that sends information to log files	32
About log.Fatal()	35
About log.Panic()	35
Writing to a custom log file	37
Printing line numbers in log entries	39
Error handling in Go	41
The error data type	41
Error handling	43
Using Docker	46
Exercises and links	51
Summary	52
Chapter 2: Understanding Go Internals	53
The Go compiler	54

Garbage collection	55
The tricolor algorithm	58
More about the operation of the Go garbage collector	61
Maps, slices, and the Go garbage collector	62
Using a slice	63
Using a map with pointers	63
Using a map without pointers	64
Splitting the map	65
Comparing the performance of the presented techniques	65
Unsafe code	66
About the unsafe package	68
Another example of the unsafe package	69
Calling C code from Go	70
Calling C code from Go using the same file	70
Calling C code from Go using separate files	71
The C code	72
The Go code	72
Mixing Go and C code	74
Calling Go functions from C code	75
The Go package	75
The C code	76
The defer keyword	77
Using defer for logging	80
Panic and recover	82
Using the panic function on its own	84
Two handy UNIX utilities	85
The strace tool	85
The dtrace tool	86
Your Go environment	88
The go env command	90
The Go assembler	91
Node trees	92
Finding out more about go build	98
Creating WebAssembly code	100
A quick introduction to WebAssembly	100
Why is WebAssembly important?	100
Go and WebAssembly	100
An example	101
Using the generated WebAssembly code	102
General Go coding advice	104
Exercises and links	104
Summary	105
Chapter 3: Working with Basic Go Data Types	107
Numeric data types	108

Integers	108
Floating-point numbers	108
Complex numbers	108
Number literals in Go 2	111
Go loops	111
The for loop	111
The while loop	112
The range keyword	113
An example with multiple Go loops	113
Go arrays	115
Multi-dimensional arrays	116
The shortcomings of Go arrays	118
Go slices	119
Performing basic operations on slices	119
Slices are expanded automatically	121
Byte slices	123
The copy() function	123
Multi-dimensional slices	125
Another example with slices	126
Sorting slices using sort.Slice()	128
Appending an array to a slice	130
Go maps	131
Storing to a nil map	133
When you should use a map	134
Go constants	134
The constant generator iota	136
Go pointers	139
Why use pointers?	142
Times and dates	142
Working with times	144
Parsing times	144
Working with dates	146
Parsing dates	146
Changing date and time formats	147
Measuring execution time	149
Measuring the operation of the Go garbage collector	151
Web links and exercises	151
Summary	152
Chapter 4: The Uses of Composite Types	153
About composite types	154
Structures	154
Pointers to structures	156
Using the new keyword	158
Tuples	159

Regular expressions and pattern matching	160
Introducing some theory	161
A simple example	161
A more advanced example	164
Matching IPv4 addresses	167
Strings	171
What is a rune?	174
The unicode package	176
The strings package	177
The switch statement	181
Calculating Pi with high accuracy	185
Developing a key-value store in Go	187
Go and the JSON format	193
Reading JSON data	193
Saving JSON data	195
Using Marshal() and Unmarshal()	197
Parsing JSON data	198
Go and XML	201
Reading an XML file	204
Customizing XML output	206
Go and the YAML format	207
Additional resources	208
Exercises and web links	208
Summary	209
Chapter 5: How to Enhance Go Code with Data Structures	210
About graphs and nodes	211
Algorithm complexity	211
Binary trees in Go	212
Implementing a binary tree in Go	213
Advantages of binary trees	215
Hash tables in Go	216
Implementing a hash table in Go	217
Implementing the lookup functionality	219
Advantages of hash tables	220
Linked lists in Go	220
Implementing a linked list in Go	222
Advantages of linked lists	225
Doubly linked lists in Go	226
Implementing a doubly linked list in Go	227
Advantages of doubly linked lists	230
Queues in Go	231
Implementing a queue in Go	231
Stacks in Go	234

Implementing a stack in Go	234
The container package	237
Using container/heap	238
Using container/list	240
Using container/ring	243
Generating random numbers	244
Generating random strings	248
Generating secure random numbers	250
Performing matrix calculations	252
Adding and subtracting matrices	252
Multiplying matrices	255
Dividing matrices	258
A tip on finding out the dimensions of an array	264
Solving Sudoku puzzles	264
Additional resources	268
Exercises	268
Summary	269
Chapter 6: What You Might Not Know About Go Packages and Functions	271
About Go packages	272
About Go functions	272
Anonymous functions	273
Functions that return multiple values	273
The return values of a function can be named	275
Functions with pointer parameters	277
Functions that return pointers	278
Functions that return other functions	279
Functions that accept other functions as parameters	281
Variadic functions	282
Developing your own Go packages	284
Compiling a Go package	286
Private variables and functions	286
The init() function	286
Go modules	289
Creating and using a Go module	289
Creating version v1.0.0	290
Using version v1.0.0	290
Creating version v1.1.0	292
Using version v1.1.0	293
Creating version v2.0.0	294
Using version v2.0.0	296
Creating version v2.1.0	296
Using version v2.1.0	297
Using two different versions of the same Go module	298
Where Go stores Go modules	299

The go mod vendor command	300
Creating good Go packages	300
The syscall package	302
Finding out how fmt.Println() really works	305
The go/scanner, go/parser, and go/token packages	307
The go/ast package	307
The go/scanner package	307
The go/parser package	310
A practical example	312
Finding variable names with a given string length	314
Text and HTML templates	319
Generating text output	319
Constructing HTML output	322
Additional resources	329
Exercises	330
Summary	331
Chapter 7: Reflection and Interfaces for All Seasons	332
Type methods	333
Go interfaces	335
About type assertions	336
Writing your own interfaces	338
Using a Go interface	339
Using switch with interface and data types	341
Reflection	343
A simple reflection example	344
A more advanced reflection example	346
The three disadvantages of reflection	349
The reflectwalk library	350
Object-oriented programming in Go	352
An introduction to git and GitHub	356
Using git	356
The git status command	356
The git pull command	357
The git commit command	357
The git push command	357
Working with branches	358
Working with files	359
The .gitignore file	360
Using git diff	360
Working with tags	360
The git cherry-pick command	362
Debugging with Delve	362
A debugging example	363
Additional resources	367
Exercises	367

Summary	368
Chapter 8: Telling a UNIX System What to Do	369
About UNIX processes	370
The flag package	370
The viper package	375
A simple viper example	376
From flag to viper	377
Reading JSON configuration files	379
Reading YAML configuration files	380
The cobra package	383
A simple cobra example	384
Creating command aliases	389
The io.Reader and io.Writer Interfaces	391
Buffered and unbuffered file input and output	392
The bufio package	392
Reading text files	392
Reading a text file line by line	393
Reading a text file word by word	394
Reading a text file character by character	396
Reading from /dev/random	398
Reading a specific amount of data	400
The advantages of binary formats	402
Reading CSV files	403
Writing to a file	406
Loading and saving data on disk	409
The strings package revisited	413
About the bytes package	414
File permissions	416
Handling UNIX signals	417
Handling two signals	418
Handling all signals	420
Programming UNIX pipes in Go	422
Implementing the cat(1) utility in Go	423
About syscall.PtraceRegs	425
Tracing system calls	427
User ID and group ID	432
The Docker API and Go	433
Additional resources	436
Exercises	437
Summary	438
Chapter 9: Concurrency in Go – Goroutines, Channels, and Pipelines	439
About processes, threads, and goroutines	440

The Go scheduler	441
Concurrency and parallelism	441
Goroutines	442
Creating a goroutine	442
Creating multiple goroutines	444
Waiting for your goroutines to finish	445
What if the number of Add() and Done() calls do not agree?	448
Channels	449
Writing to a channel	450
Reading from a channel	452
Receiving from a closed channel	453
Channels as function parameters	454
Pipelines	455
Race conditions	459
Comparing Go and Rust concurrency models	461
Comparing Go and Erlang concurrency models	461
Additional resources	462
Exercises	462
Summary	463
Chapter 10: Concurrency in Go – Advanced Topics	464
The Go scheduler revisited	465
The GOMAXPROCS environment variable	467
The select keyword	468
Timing out a goroutine	471
Timing out a goroutine – take 1	471
Timing out a goroutine – take 2	473
Go channels revisited	475
Signal channels	476
Buffered channels	476
Nil channels	478
Channels of channels	480
Specifying the order of execution for your goroutines	483
How not to use goroutines	485
Shared memory and shared variables	486
The sync.Mutex type	487
What happens if you forget to unlock a mutex?	490
The sync.RWMutex type	492
The atomic package	495
Sharing memory using goroutines	497
Revisiting the go statement	500
Catching race conditions	503
The context package	508
An advanced example of the context package	512
Another example of the context package	517

Worker pools	519
Additional resources	524
Exercises	524
Summary	525
Chapter 11: Code Testing, Optimization, and Profiling	526
About optimization	527
Optimizing Go code	528
Profiling Go code	529
The net/http/pprof standard Go package	529
A simple profiling example	529
A convenient external package for profiling	537
The web interface of the Go profiler	539
A profiling example that uses the web interface	539
A quick introduction to Graphviz	542
The go tool trace utility	543
Testing Go code	549
Writing tests for existing Go code	549
Test code coverage	554
Testing an HTTP server with a database backend	556
The testing/quick package	564
What if testing takes too long or never finishes?	567
Benchmarking Go code	569
A simple benchmarking example	570
Wrongly defined benchmark functions	576
Benchmarking buffered writing	577
Finding unreachable Go code	581
Cross-compilation	583
Creating example functions	585
From Go code to machine code	587
Using assembly with Go	588
Generating documentation	590
Using Docker images	596
Additional resources	598
Exercises	599
Summary	599
Chapter 12: The Foundations of Network Programming in Go	601
About net/http, net, and http.RoundTripper	602
The http.Response type	602
The http.Request type	603
The http.Transport type	604
About TCP/IP	605
About IPv4 and IPv6	605
The nc(1) command-line utility	606

Reading the configuration of network interfaces	606
Performing DNS lookups	611
Getting the NS records of a domain	613
Getting the MX records of a domain	615
Creating a web server in Go	616
Using the atomic package	620
Profiling an HTTP server	623
Creating a website in Go	628
HTTP tracing	638
Testing HTTP handlers	641
Creating a web client in Go	644
Making your Go web client more advanced	646
Timing out HTTP connections	649
More information about SetDeadline	651
Setting the timeout period on the server side	651
Yet another way to time out	653
The Wireshark and tshark tools	655
gRPC and Go	656
Defining the interface definition file	656
The gRPC client	658
The gRPC server	660
Additional resources	662
Exercises	664
Summary	664
Chapter 13: Network Programming – Building Your Own Servers and Clients	666
Working with HTTPS traffic	667
Creating certificates	667
An HTTPS client	668
A simple HTTPS server	670
Developing a TLS server and client	671
The net standard Go package	675
A TCP client	675
A slightly different version of the TCP client	677
A TCP server	679
A slightly different version of the TCP server	681
A UDP client	683
Developing a UDP server	686
A concurrent TCP server	688
A handy concurrent TCP server	692
Creating a Docker image for a Go TCP/IP server	699
Remote Procedure Call (RPC)	701
The RPC client	702

The RPC server	703
Doing low-level network programming	705
Grabbing raw ICMP network data	707
Additional resources	712
Exercises	713
Summary	714
Chapter 14: Machine Learning in Go	715
Calculating simple statistical properties	716
Regression	720
Linear regression	720
Implementing linear regression	720
Plotting data	722
Classification	727
Clustering	731
Anomaly detection	733
Neural networks	736
Outlier analysis	738
Working with TensorFlow	741
Talking to Kafka	746
Additional resources	751
Exercises	752
Summary	752
Where to go next?	752
Other Books You May Enjoy	754
Index	756

Preface

The book you are reading right now is *Mastering Go, Second Edition*, and is all about helping you become a better Go developer!

There are many exciting new topics, including an entirely new chapter that talks about Machine Learning in Go as well as information and code examples relating to the Viper and Cobra Go packages, gRPC, working with Docker images, working with YAML files, working with the `go/scanner` and `go/token` packages, and generating WebAssembly code from Go. In total, there are more than 130 new pages in this second edition of *Mastering Go*.

Who this book is for

This book is for amateur and intermediate Go programmers who want to take their Go knowledge to the next level, as well as experienced developers in other programming languages who want to learn Go without learning again how a `for` loop works.

Some of the information found in this book can be also found in my other book, *Go Systems Programming*. The main difference between these two books is that *Go Systems Programming* is about developing system tools using the capabilities of Go, whereas *Mastering Go* is about explaining the capabilities and the internals of Go in order to become a better Go developer. Both books can be used as a reference after reading them for the first or the second time.

What this book covers

Chapter 1, *Go and the Operating System*, begins by talking about the history of Go and the advantages of Go before describing the `godoc` utility and explaining how you can compile and execute Go programs. After that, it talks about printing output and getting user input, working with the command-line arguments of a program and using log files. The final topic in the first chapter is error handling, which plays a key role in Go.

Chapter 2, *Understanding Go Internals*, discusses the Go garbage collector and the way it operates. Then, it talks about unsafe code and the `unsafe` package, how to call C code from a Go program, and how to call Go code from a C program.

After that, it showcases the use of the `defer` keyword and presents the `strace(1)` and `dtrace(1)` utilities. In the remaining sections of the chapter, you will learn how to find information about your Go environment, the use of the Go assembler, and how to generate WebAssembly code from Go.

Chapter 3, *Working with Basic Go Data Types*, talks about the data types offered by Go, which includes arrays, slices, and maps, as well as Go pointers, constants, loops, and working with dates and times. You would not want to miss this chapter!

Chapter 4, *The Uses of Composite Types*, begins by teaching you about Go structures and the `struct` keyword before discussing tuples, strings, runes, byte slices, and string literals. The remainder of the chapter talks about regular expressions and pattern matching, the `switch` statement, the `strings` package, the `math/big` package, about developing a key-value store in Go, and about working with XML and JSON files.

Chapter 5, *How to Enhance Go Code with Data Structures*, is about developing your own data structures when the structures offered by Go do not fit a particular problem. This includes developing binary trees, linked lists, hash tables, stacks, and queues, and learning about their advantages. This chapter also showcases the use of the structures found in the `container` standard Go package, as well as how to use Go to verify Sudoku puzzles and generate random numbers.

Chapter 6, *What You Might Not Know About Go Packages and Functions*, is all about packages and functions, which also includes the use of the `init()` function, the `syscall` standard Go package, and the `text/template` and `html/template` packages. Additionally, it shows the use of the `go/scanner`, `go/parser`, and `go/token` advanced packages. This chapter will definitely make you a better Go developer!

Chapter 7, *Reflection and Interfaces for All Seasons*, discusses three advanced Go concepts: reflection, interfaces, and type methods. Additionally, it discusses the object-oriented capabilities of Go and how to debug Go programs using Delve!

Chapter 8, *Telling a UNIX System What to Do*, is about Systems Programming in Go, which includes subjects such as the `flag` package for working with command-line arguments, handling UNIX signals, file input and output, the `bytes` package, the `io.Reader` and `io.Writer` interfaces, and the use of the Viper and Cobra Go packages. As I told you before, if you are really into systems programming in Go, then getting *Go Systems Programming* after reading *Mastering Go, Second Edition*, is highly recommended!

Chapter 9, *Concurrency in Go – Goroutines, Channels, and Pipelines*, discusses goroutines, channels, and pipelines, which is the Go way of achieving concurrency.

You will also learn about the differences between processes, threads, and goroutines, the `sync` package, and the way the Go scheduler operates.

Chapter 10, *Concurrency in Go – Advanced Topics*, will continue from the point where the previous chapter left off and make you a master of goroutines and channels! You will learn more about the Go scheduler, the use of the powerful `select` keyword and the various types of Go channels, as well as shared memory, mutexes, the `sync.Mutex` type, and the `sync.RWMutex` type. The final part of the chapter talks about the `context` package, worker pools, and how to detect race conditions.

Chapter 11, *Code Testing, Optimization, and Profiling*, discusses code testing, code optimization and code profiling, as well as cross-compilation, creating documentation, benchmarking Go code, creating example functions, and finding unreachable Go code.

Chapter 12, *The Foundations of Network Programming in Go*, is all about the `net/http` package and how you can develop web clients and web servers in Go. This also includes the use of the `http.Response`, `http.Request`, and `http.Transport` structures, and the `http.NewServeMux` type. You will even learn how to develop an entire web site in Go! Furthermore, in this chapter, you will learn how to read the configuration of your network interfaces and how to perform DNS lookups in Go. Additionally, you will learn how to use gRPC with Go.

Chapter 13, *Network Programming – Building Your Own Servers and Clients*, talks about working with HTTPS traffic, and creating UDP and TCP servers and clients in Go using the functionality offered by the `net` package. Other topics included in this chapter are creating RPC clients and servers as well as developing a concurrent TCP server in Go and reading raw network packages!

Chapter 14, *Machine Learning in Go*, talks about machine learning in Go, including classification, clustering, anomaly detection, outliers, neural networks and TensorFlow, as well as working with Apache Kafka with Go.

This book can be divided into three logical parts. The first part takes a sophisticated look at some important Go concepts, including user input and output, downloading external Go packages, compiling Go code, calling C code from Go, and creating WebAssembly from Go, as well as using Go basic types and Go composite types.

The second part starts with Chapter 5, *How to Enhance Go Code with Data Structures*, and also includes Chapter 6, *What You Might Not Know About Go Packages and Go Functions*, and Chapter 7, *Reflection and Interfaces for All Seasons*. These three chapters deal with Go code organization in packages and modules, the design of Go projects, and some advanced features of Go, respectively.

The last part includes the remaining seven chapters and deals with more practical Go topics. Chapters 8, 9, 10, and 11 talk about systems programming in Go, concurrency in Go, code testing, optimization, and profiling. The last three chapters of this book will talk about network programming and machine learning in Go.

The book includes content such as Go and WebAssembly, using Docker with Go, creating professional command-line tools with the Viper and Cobra packages, parsing JSON and YAML records, performing operations with matrices, working with Sudoku puzzles, the `go/scanner` and `go/token` packages, working with `git` (1) and GitHub, the `atomic` package, gRPC and Go, and HTTPS.

The book will present relatively small yet complete Go programs that illustrate the presented concepts. This has two main advantages: firstly, you do not have to look at an endless code listing when trying to learn a single technique and secondly, you can use this code as a starting point when creating your own applications and utilities.



Realizing the importance of **containers** and **Docker**, this book includes various examples of Go executable files that are used from within Docker images because Docker images offer a great way to deploy server software.

To get the most out of this book

This book requires a UNIX computer with a relatively recent Go version installed, which includes any machine running Mac OS X, macOS, or Linux. Most of the code presented will also run on Microsoft Windows machines.

To get the most out of this book, you should try to apply the knowledge of each chapter in your own programs as soon as possible and see what works and what does not! As I told you before, try to solve the exercises found at the end of each chapter or create your own programming problems.

Download the example code files

You can download the example code files for this book from your account at www.packt.com. If you purchased this book elsewhere, you can visit www.packt.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packt.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Go-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: <https://static.packt-cdn.com/downloads/9781838559335.pdf>.

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first way is similar to using the `man(1)` command, but for Go functions and packages."

A block of code is set as follows:

```
package main
import (
    "fmt"
```

```
)  
func main() {  
    fmt.Println("This is a sample Go program!")  
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
import (  
    "fmt"  
)  
func main() {  
    fmt.Println("This is a sample Go program!")  
}
```

Any command-line input or output is written as follows:

```
$ date  
Sat Oct 21 20:09:20 EEST 2017  
$ go version  
go version go1.12.7 darwin/amd64
```

Bold: Indicates a new term, an important word, or words that you see on screen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at customercare@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packt.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at copyright@packt.com with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

1

Go and the Operating System

This chapter is an introduction to various Go topics that beginners will find very useful. More experienced Go developers can also use this chapter as a refresher course on the fundamentals of Go. As it happens with most practical subjects, the best way to understand something is to experiment with it. In this case, experimenting means writing Go code on your own, making your own mistakes, and learning from them! Just don't let error messages and bugs discourage you.

In this chapter, you will learn about:

- The history and the future of the Go programming language
- The advantages of Go
- Compiling Go code
- Executing Go code
- Downloading and using external Go packages
- UNIX standard input, output, and error
- Printing data on the screen
- Getting user input
- Printing data to standard error
- Working with log files
- Using Docker to compile and execute a Go source file
- Error handling in Go

The history of Go

Go is a modern, generic-purpose, open source programming language that was officially announced at the end of 2009. Go began as an internal Google project, which means that it was started as an experiment, and has since been inspired by many other programming languages, including **C**, **Pascal**, **Alef**, and **Oberon**. Go's spiritual fathers are the professional programmers *Robert Griesemer*, *Ken Thomson*, and *Rob Pike*.

They designed Go as a language for professional programmers who want to build reliable, robust, and efficient software. Apart from its syntax and its standard functions, Go comes with a pretty rich standard library.

At the time of writing, the latest stable Go version is version 1.13. However, even if your version number is higher, the contents of the book will still be relevant.

If you are going to install Go for the first time, you can start by visiting <https://golang.org/dl/>. However, there is a big chance that your UNIX variant has a ready-to-install package for the Go programming language, so you might want to get Go by using your favorite package manager.

Where is Go going?

The Go community is already discussing the next major version of Go, which is going to be called Go 2, but there is nothing definitive at the moment.

The intention of the current Go 1 team is to make Go 2 more community driven. Although this is a good idea in general, it is always dangerous when lots of people try to make important decisions about a programming language that was initially designed and developed as an internal project by a small group of great people.

Some of the big changes that are being considered for Go 2 are generics, package versioning, and improved error handling. All these new features are under discussion at the moment and you should not be worried about them, but it is worthwhile to have an idea of the direction that Go is going in.

The advantages of Go

Go has many advantages, and some of them are unique to Go, while others are shared with other programming languages.

The list of the most significant Go advantages and features includes the following:

- Go is a modern programming language that is easy to read, easy to understand, and was made by experienced developers.
- Go wants happy developers because happy developers write better code!
- The Go compiler prints practical warning and error messages that help you to solve the actual problem. Putting it simply, the Go compiler is there to help you, not to make your life miserable by printing pointless output!

- Go code is portable, especially among UNIX machines.
- Go has support for procedural, concurrent, and distributed programming.
- Go supports **garbage collection**, so you do not have to deal with memory allocation and deallocation.
- Go does not have a **preprocessor** and does high-speed compilation. As a consequence, Go can also be used as a scripting language.
- Go can build web applications and provides a simple web server for testing purposes.
- The standard Go library offers many packages that simplify the work of the developer. Additionally, the functions found in the standard Go library are tested and debugged in advance by the people who develop Go, which means that, most of the time, they come without bugs.
- Go uses **static linking** by default, which means that the binary files produced can be easily transferred to other machines with the same OS. As a consequence, once a Go program is compiled successfully and an executable file is generated, you do not need to worry about libraries, dependencies, and different library versions anymore.
- You will not need a **graphical user interface (GUI)** for developing, debugging, and testing Go applications, as Go can be used from the command-line, which I think many UNIX people prefer.
- Go supports **Unicode**, which means that you do not need any extra code for printing characters from multiple human languages.
- Go keeps concepts orthogonal because a few orthogonal features work better than many overlapping ones.

Is Go perfect?

There is no such thing as the perfect programming language, and Go is not an exception to this rule. However, some programming languages are better at some areas of programming or we like them more than other programming languages. Personally, I do not like Java, and while I used to like C++, I do not like it anymore. C++ has become too complex as a programming language, whereas, in my opinion, Java, code does not look good.

Some of the disadvantages of Go are:

- Go does not have direct support for **object-oriented programming**, which can be a problem for programmers who are used to writing code in an object-oriented manner. Nevertheless, you can use composition in Go to mimic inheritance.

- For some people, Go will never replace C.
- C is still faster than any other programming language for systems programming and this is mainly because UNIX is written in C.

Nevertheless, Go is a pretty decent programming language that will not disappoint you if you find the time to learn it and program in it.

What is a preprocessor?

I said earlier that Go does not have a **preprocessor** and that this is a good thing. A preprocessor is a program that processes your input data and generates output that will be used as the input to another program. In the context of programming languages, the input of a preprocessor is source code that will be processed by the preprocessor before being given as input to the compiler of the programming language.

The biggest disadvantage of a preprocessor is that it knows nothing about the underlying language or its syntax! This means that when a preprocessor is used, you cannot be certain that the final version of your code will do what you really want because the preprocessor might alter the logic as well as the semantics of your original code.

The list of programming languages with a preprocessor includes C, C++, Ada, and PL/SQL. The infamous C preprocessor processes lines that begin with # and are called **directives** or **pragmas**. As stated before, directives and pragmas are not part of the C programming language!

The godoc utility

The Go distribution comes with a plethora of tools that can make your life as a programmer easier. One of these tools is the `godoc` utility, which allows you to see the documentation of existing Go functions and packages without needing an internet connection.

The `godoc` utility can be executed either as a normal command-line application that displays its output on a terminal, or as a command-line application that starts a web server. In the latter case, you will need a web browser to look at the Go documentation.



If you type `godoc` without any command-line parameters, you will get a list of the command-line options supported by `godoc`.

The first way is similar to using the `man(1)` command, but for Go functions and packages. So, in order to find information about the `Printf()` function of the `fmt` package, you should execute the following command:

```
$ go doc fmt.Printf
```

Similarly, you can find information about the entire `fmt` package by running the following command:

```
$ go doc fmt
```

The second way requires executing `godoc` with the `-http` parameter:

```
$ godoc -http=:8001
```

The numeric value in the preceding command, which in this case is `8001`, is the port number the HTTP server will listen to. You can choose any port number that is available provided that you have the right privileges. However, note that port numbers 0-1023 are restricted and can only be used by the root user, so it is better to avoid choosing one of those and pick something else, provided that it is not already in use by a different process.

You can omit the equal sign in the presented command and put a space character in its place. So, the following command is completely equivalent to the previous one:

```
$ godoc -http :8001
```

After that, you should point your web browser to the `http://localhost:8001/pkg/` URL in order to get the list of available Go packages and browse their documentation.

Compiling Go code

In this section, you will learn how to compile Go code. The good news is that you can compile your Go code from the command line without the need for a graphical application. Furthermore, Go does not care about the name of the source file of an autonomous program as long as the package name is `main` and there is a single `main()` function in it. This is because the `main()` function is where the program execution begins. As a result, you cannot have multiple `main()` functions in the files of a single project.

We will start our first Go program compilation with a program named `aSourceFile.go` that contains the following Go code:

```
package main
import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

Notice that the Go community prefers to name the Go source file `source_file.go` instead of `aSourceFile.go`. Whatever you choose, be consistent.

In order to compile `aSourceFile.go` and create a **statically linked** executable file, you will need to execute the following command:

```
$ go build aSourceFile.go
```

After that, you will have a new executable file named `aSourceFile` that you will need to execute:

```
$ file aSourceFile
aSourceFile: Mach-O 64-bit executable x86_64
$ ls -l aSourceFile
-rwxr-xr-x 1 mtsouk staff 2007576 Jan 10 21:10 aSourceFile
$ ./aSourceFile
This is a sample Go program!
```

The main reason why the file size of `aSourceFile` is that big is because it is statically linked, which means that it does not require any external libraries to run.

Executing Go code

There is another way to execute your Go code that does not create any permanent executable files – it just generates some intermediate files that are automatically deleted afterward.



The way presented allows you to use Go as if it is a scripting programming language like **Python**, **Ruby**, or **Perl**.

So, in order to run `aSourceFile.go` without creating an executable file, you will need to execute the following command:

```
$ go run aSourceFile.go  
This is a sample Go program!
```

As you can see, the output of the preceding command is exactly the same as before.



Please note that with `go run`, the Go compiler still needs to create an executable file. It is because you do not see it, it is automatically executed, and it is automatically deleted after the program has finished that you might think that there is no need for an executable file.

This book mainly uses `go run` to execute the example code; primarily because it is simpler than running `go build` and then executing the executable file. Additionally, `go run` does not leave any files on your hard disk after the program has finished its execution.

Two Go rules

Go has strict coding rules that are there to help you avoid silly errors and bugs in your code, as well as to make your code easier to read for the Go community. This section will present two such Go rules that you need to know.

As mentioned earlier, please remember that the Go compiler is there to help and not make your life miserable. As a result, the main purpose of the Go compiler is to compile and increase the quality of your Go code.

You either use a Go package or you do not include it

Go has strict rules about package usage. Therefore, you cannot just include any package you might think that you will need and then not use it afterward.

Look at the following naive program, which is saved as `packageNotUsed.go`:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("Hello there!")
}
```



In this book, you are going to see lots of error messages, error situations, and warnings. I believe that looking at code that fails to compile is also useful and sometimes even more valuable than just looking at Go code that compiles without any errors. The Go compiler usually displays useful error messages and warnings that will most likely help you to resolve an erroneous situation, so do not underestimate Go error messages and warnings.

If you execute `packageNotUsed.go`, you will get the following error message from Go and the program will not get executed:

```
$ go run packageNotUsed.go
# command-line-arguments
./packageNotUsed.go:5:2: imported and not used: "os"
```

If you remove the `os` package from the `import` list of the program, `packageNotUsed.go` will compile just fine; try it on your own.

Although this is not the perfect time to start talking about breaking Go rules, there is a way to bypass this restriction. This is showcased in the following Go code that is saved in the `packageNotUsedUnderscore.go` file:

```
package main

import (
    "fmt"
    _ "os"
)

func main() {
    fmt.Println("Hello there!")
}
```

So, using an underscore character in front of a package name in the `import` list will not create an error message in the compilation process even if that package will not be used in the program:

```
$ go run packageNotUsedUnderscore.go
Hello there!
```



The reason that Go is allowing you to bypass this rule will become more evident in Chapter 6, *What You Might Not Know About Go Packages and Go Functions*.

There is only one way to format curly braces

Look at the following Go program named `curly.go`:

```
package main

import (
    "fmt"
)

func main()
{
    fmt.Println("Go has strict rules for curly braces!")
}
```

Although it looks just fine, if you try to execute it, you will be fairly disappointed, because you will get the following **syntax error** message and the code will not compile and therefore run:

```
$ go run curly.go
# command-line-arguments
./curly.go:7:6: missing function body for "main"
./curly.go:8:1: syntax error: unexpected semicolon or newline before {
```

The official explanation for this error message is that Go requires the use of semicolons as statement terminators in many contexts, and the compiler automatically inserts the required semicolons when it thinks that they are necessary. Therefore, putting the opening curly brace (`{`) in its own line will make the Go compiler insert a semicolon at the end of the previous line (`func main()`), which is the cause of the error message.

Downloading Go packages

Although the standard Go library is very rich, there are times that you will need to download external Go packages in order to use their functionality. This section will teach you how to download an external Go package and where it will be placed on your UNIX machine.



Have in mind that although Go **modules**, which is a new Go feature that is still under development, might introduce changes to the way you work with external Go code, the process of downloading a single Go package into your computer will remain the same.



You will learn a lot more about Go packages and Go modules in [Chapter 6, What You Might Not Know About Go Packages and Go Functions](#).

Look at the following naive Go program that is saved as `getPackage.go`:

```
package main

import (
    "fmt"
    "github.com/mactsouk/go/simpleGitHub"
)

func main() {
    fmt.Println(simpleGitHub.AddTwo(5, 6))
}
```

This program uses an external package because one of the `import` commands uses an internet address. In this case, the external package is called `simpleGitHub` and is located at `github.com/mactsouk/go/simpleGitHub`.

If you try to execute `getPackage.go` right away, you will be disappointed:

```
$ go run getPackage.go
getPackage.go:5:2: cannot find package
"github.com/mactsouk/go/simpleGitHub" in any of:
    /usr/local/Cellar/go/1.9.1/libexec/src/github.com/mactsouk/go/
simpleGitHub (from $GOROOT)
    /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub (from $GOPATH)
```

So, you will need to get the missing package on your computer. In order to download it, you will need to execute the following command:

```
$ go get -v github.com/mactsouk/go/simpleGitHub
github.com/mactsouk/go (download)
github.com/mactsouk/go/simpleGitHub
```

After that, you can find the downloaded files at the following directory:

```
$ ls -l ~/go/src/github.com/mactsouk/go/simpleGitHub/
total 8
-rw-r--r--  1 mtsouk  staff   66 Oct 17 21:47 simpleGitHub.go
```

However, the `go get` command also compiles the package. The relevant files can be found at the following place:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
-rw-r--r--  1 mtsouk  staff  1050 Oct 17 21:47
/Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
```

You are now ready to execute `getPackage.go` without any problems:

```
$ go run getPackage.go
11
```

You can delete the intermediate files of a downloaded Go package as follows:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
cd /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub
rm -f simpleGitHub.test simpleGitHub.test.exe
rm -f /Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/go/
simpleGitHub.a
```

Similarly, you can delete an entire Go package you have downloaded locally using the `rm(1)` UNIX command to delete its Go source after using `go clean`:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
$ rm -rf ~/go/src/github.com/mactsouk/go/simpleGitHub
```

After executing the former commands, you will need to download the Go package again.

UNIX stdin, stdout, and stderr

Every UNIX OS has three files open all the time for its processes. Remember that UNIX considers everything, even a printer or your mouse, a file.

UNIX uses **file descriptors**, which are positive integer values, as an internal representation for accessing all of its open files, which is much prettier than using long paths.

So, by default, all UNIX systems support three special and standard filenames: `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`, which can also be accessed using file descriptors 0, 1, and 2, respectively. These three file descriptors are also called **standard input**, **standard output**, and **standard error**, respectively. Additionally, file descriptor 0 can be accessed as `/dev/fd/0` on a macOS machine and as both `/dev/fd/0` and `/dev/pts/0` on a Debian Linux machine.

Go uses `os.Stdin` for accessing standard input, `os.Stdout` for accessing standard output, and `os.Stderr` for accessing standard error. Although you can still use `/dev/stdin`, `/dev/stdout`, and `/dev/stderr` or the related file descriptor values for accessing the same devices, it is better, safer, and more portable to stick with `os.Stdin`, `os.Stdout`, and `os.Stderr` offered by Go.

About printing output

As with UNIX and C, Go offers a variety of ways for printing your output on the screen. All the printing functions of this section require the use of the `fmt` Go standard package and are illustrated in the `printing.go` program, which will be presented in two parts.

The simplest way to print something in Go is by using the `fmt.Println()` and the `fmt.Printf()` functions. The `fmt.Printf()` function has many similarities with the C `printf(3)` function. You can also use the `fmt.Print()` function instead of `fmt.Println()`. The main difference between `fmt.Print()` and `fmt.Println()` is that the latter automatically adds a newline character each time you call it.

On the other hand, the biggest difference between `fmt.Println()` and `fmt.Printf()` is that the latter requires a **format specifier** for each *thing* that you want to print, just like the C `printf(3)` function, which means that you have better control of what you are doing, but you have to write more code. Go calls these format specifiers **verbs**. You can find more information about verbs at <https://golang.org/pkg/fmt/>.

If you have to perform any formatting before printing something or have to arrange multiple variables, then using `fmt.Printf()` might be a better choice. However, if you only have to print a single variable, then you might need to choose either `fmt.Print()` or `fmt.Println()`, depending on whether you need a newline character or not.

The first part of `printing.go` contains the following Go code:

```
package main

import (
    "fmt"
)

func main() {
    v1 := "123"
    v2 := 123
    v3 := "Have a nice day\n"
    v4 := "abc"
```

In this part, you see the `import` of the `fmt` package and the definition of four Go variables. The `\n` used in `v3` is the line break character. However, if you just want to insert a line break in your output, you can call `fmt.Println()` without any arguments, instead of using something like `fmt.Print("\n")`.

The second part is as follows:

```
    fmt.Print(v1, v2, v3, v4)
    fmt.Println()
    fmt.Println(v1, v2, v3, v4)
    fmt.Print(v1, " ", v2, " ", v3, " ", v4, "\n")
    fmt.Printf("%s%d %s %s\n", v1, v2, v3, v4)
}
```

In this part, you print the four variables using `fmt.Println()`, `fmt.Print()`, and `fmt.Printf()` in order to better understand how they differ.

If you execute `printing.go`, you will get the following output:

```
$ go run printing.go
123123Have a nice day
abc
123 123 Have a nice day
abc
123 123 Have a nice day
abc
123123 Have a nice day
abc
```

As you can see from the preceding output, the `fmt.Println()` function also adds a space character between its parameters, which is not the case with `fmt.Print()`.

As a result, a statement such as `fmt.Println(v1, v2)` is equivalent to `fmt.Print(v1, " ", v2, "\n")`.

Apart from `fmt.Println()`, `fmt.Print()`, and `fmt.Printf()`, which are the simplest functions that can be used for generating output on the screen, there is also the S family of functions that includes `fmt.Sprintln()`, `fmt.Sprint()`, and `fmt.Sprintf()`. These functions are used to create strings based on a given format.

Finally, there is the F family of functions, which includes `fmt.Fprintln()`, `fmt.Fprint()`, and `fmt.Fprintf()`. They are used for writing to files using an `io.Writer`.



You will learn more about the `io.Writer` and `io.Reader` **interfaces** in Chapter 8, *Telling a UNIX System What to Do*.

The next section will teach you how to print your data using standard output, which is pretty common in the UNIX world.

Using standard output

Standard output is more or less equivalent to printing on the screen. However, using standard output might require the use of functions that do not belong to the `fmt` package, which is why it is presented in its own section.

The relevant technique will be illustrated in `stdOUT.go` and will be offered in three parts. The first part of the program is as follows:

```
package main

import (
    "io"
    "os"
)
```

So, `stdOUT.go` will use the `io` package instead of the `fmt` package. The `os` package is used for reading the command-line arguments of the program and for accessing `os.Stdout`.

The second portion of `stdOUT.go` contains the following Go code:

```
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
}
```

The `myString` variable holds the text that will be printed on the screen, which is either the first command-line argument of the program or, if the program was executed without any command-line arguments, a hardcoded text message.

The third part of the program is as follows:

```
io.WriteString(os.Stdout, myString)
io.WriteString(os.Stdout, "\n")
}
```

In this case, the `io.WriteString()` function works in the same way as the `fmt.Print()` function; however, it takes only two parameters. The first parameter is the file you want to write to, which, in this case, is `os.Stdout`, and the second parameter is a string variable.



Strictly speaking, the type of the first parameter of the `io.WriteString()` function should be `io.Writer`, which requires a **slice** of bytes as the second parameter. However, in this case, a string variable does the job just fine. You will learn more about slices in Chapter 3, *Working with Basic Go Data Types*.

Executing `stdOUT.go` will produce the following output:

```
$ go run stdOUT.go
Please give me one argument!
$ go run stdOUT.go 123 12
123
```

The preceding output verifies that the `io.WriteString()` function sends the contents of its second parameter onto the screen when its first parameter is `os.Stdout`.

Getting user input

There are three main ways to get user input: firstly, by reading the command-line arguments of a program; secondly, by asking the user for input; or thirdly, by reading external files. This section will present the first two ways. Should you wish to learn how to read an external file, you should visit [Chapter 8, Telling a UNIX System What to Do](#).

About := and =

Before continuing, it will be very useful to talk about the use of := and how it differs from =. The official name for := is the **short assignment statement**. The short assignment statement can be used in place of a var declaration with an implicit type.



You will rarely see the use of var in Go; the var keyword is mostly used for declaring global variables in Go programs, as well as for declaring variables without an initial value. The reason for the former is that every statement that exists outside of the code of a function must begin with a keyword such as func or var. This means that the short assignment statement cannot be used outside of a function because it is not available there.

The := operator works as follows:

```
m := 123
```

The result of the preceding statement is a new integer variable named m with a value of 123.

However, if you try to use := on an already declared variable, the compilation will fail with the following error message, which makes perfect sense:

```
$ go run test.go
# command-line-arguments
./test.go:5:4: no new variables on left side of :=
```

So, you might now ask, what will happen if you are expecting two or more values from a function and you want to use an existing variable for one of them. Should you use := or =? The answer is simple: you should use :=, as in the following code example:

```
i, k := 3, 4
j, k := 1, 2
```

As the `j` variable is used for the first time in the second statement, you use `:=` even though `k` has already been defined in the first statement.

Although it may seem boring to talk about such insignificant things, knowing them will save you from various types of errors in the long run!

Reading from standard input

The reading of data from the standard input will be illustrated in `stdIN.go`, which you will see in two parts. The first part is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
)
```

In the preceding code, you can see the use of the `bufio` package for the first time in this book.



You will learn more about the `bufio` package in Chapter 8, *Telling a UNIX System What to Do*.

Although the `bufio` package is mostly used for file input and output, you will keep seeing the `os` package all the time in this book because it contains many handy functions; its most common functionality is that it provides a way to access the command-line arguments of a Go program (`os.Args`).

The official description of the `os` package tells us that it offers functions that perform OS operations. This includes functions for creating, deleting, and renaming files and directories, as well as functions for learning the UNIX permissions and other characteristics of files and directories. The main advantage of the `os` package is that it is platform independent. Put simply, its functions will work on both UNIX and Microsoft Windows machines.

The second part of `stdIN.go` contains the following Go code:

```
func main() {
```

```
var f *os.File
f = os.Stdin
defer f.Close()

scanner := bufio.NewScanner(f)
for scanner.Scan() {
    fmt.Println(">", scanner.Text())
}
}
```

First, there is a call to `bufio.NewScanner()` using standard input (`os.Stdin`) as its parameter. This call returns a `bufio.Scanner` variable, which is used with the `Scan()` function for reading from `os.Stdin` line by line. Each line that is read is printed on the screen before getting the next one. Please note that each line that is printed by the program begins with the `>` character.

The execution of `stdIN.go` will produce the following kind of output:

```
$ go run stdIN.go
This is number 21
> This is number 21
This is Mihalis
> This is Mihalis
Hello Go!
> Hello Go!
Press Control + D on a new line to end this program!
> Press Control + D on a new line to end this program!
```

According to the UNIX way, you can tell a program to stop reading data from standard input by pressing `Ctrl + D`.



The Go code of `stdIN.go` and `stdOUT.go` will be very useful when we talk about UNIX **pipes** in Chapter 8, *Telling a UNIX System What to Do*, so do not underestimate their simplicity.

Working with command-line arguments

The technique of this section will be illustrated using the Go code of `cla.go`, which will be presented in three parts. The program will find the minimum and the maximum of its command-line arguments.

The first part of the program is as follows:

```
package main
```

```
import (
    "fmt"
    "os"
    "strconv"
)
```

What is important here is realizing that getting the command-line arguments requires the use of the `os` package. Additionally, you need another package, named `strconv`, in order to be able to convert a command-line argument, which is given as a string, into an arithmetical data type.

The second part of the program is the following:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    min, _ := strconv.ParseFloat(arguments[1], 64)
    max, _ := strconv.ParseFloat(arguments[1], 64)
}
```

Here, `cla.go` checks whether you have any command-line arguments by checking the length of `os.Args`. This is because the program needs at least one command-line argument to operate. Please note that `os.Args` is a Go slice with `string` values. The first element in the slice is the name of the executable program. Therefore, in order to initialize the `min` and `max` variables, you will need to use the second element of the `string` type `os.Args` slice that has an index value of 1.

There is an important point here: the fact that you are expecting one or more floats does not necessarily mean that the user will give you valid floats, either by accident or on purpose. However, as we have not talked about error handling in Go so far, `cla.go` assumes that all command-line arguments are in the right format and therefore will be acceptable. As a result, `cla.go` ignores the `error` value returned by the `strconv.ParseFloat()` function using the following statement:

```
n, _ := strconv.ParseFloat(arguments[i], 64)
```

The preceding statement tells Go that you only want to get the first value returned by `strconv.ParseFloat()` and that you are not interested in the second value, which in this case is an `error` variable, by assigning it to the underscore character. The underscore character, which is called **blank identifier**, is the Go way of discarding a value. If a Go function returns multiple values, you can use the blank identifier multiple times.



Ignoring all or some of the return values of a Go function, especially the error values, is a very dangerous technique that should not be used in production code!

The third part comes with the following Go code:

```
for i := 2; i < len(arguments); i++ {
    n, _ := strconv.ParseFloat(arguments[i], 64)

    if n < min {
        min = n
    }
    if n > max {
        max = n
    }
}

fmt.Println("Min:", min)
fmt.Println("Max:", max)
}
```

Here, you use a `for` loop that will help you to visit all the elements of the `os.Args` slice, which was previously assigned to the `arguments` variable.

Executing `cla.go` will create the following kind of output:

```
$ go run cla.go -10 0 1
Min: -10
Max: 1
$ go run cla.go -10
Min: -10
Max: -10
```

As you might expect, the program does not behave well when it receives erroneous input; the worst thing of all is that it does not generate any warnings to inform the user that there was an error (or several) while processing the command-line arguments of the program:

```
$ go run cla.go a b c 10
Min: 0
Max: 10
```

About error output

This section will present a technique for sending data to **UNIX standard error**, which is the UNIX way of differentiating between actual values and error output.

The Go code for illustrating the use of standard error in Go is included in `stdERR.go` and will be presented in two parts. As writing to standard error requires the use of the file descriptor related to standard error, the Go code of `stdERR.go` will be based on the Go code of `stdOUT.go`.

The first part of the program is as follows:

```
package main

import (
    "io"
    "os"
)
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
}
```

So far, `stdERR.go` is almost identical to `stdOUT.go`.

The second portion of `stdERR.go` is the following:

```
    io.WriteString(os.Stdout, "This is Standard output\n")
    io.WriteString(os.Stderr, myString)
    io.WriteString(os.Stderr, "\n")
}
```

You call `io.WriteString()` two times to write to standard error (`os.Stderr`) and one more time to write to standard output (`os.Stdout`).

Executing `stdERR.go` will create the following output:

```
$ go run stdERR.go
This is Standard output
Please give me one argument!
```

The preceding output cannot help you to differentiate between data written to standard output and data written to standard error, which can be very useful sometimes. However, if you are using the `bash(1)` shell, there is a trick you can use in order to distinguish between standard output data and standard error data. Almost all UNIX shells offer this functionality in their own way.

When using `bash(1)`, you can redirect the standard error output to a file as follows:

```
$ go run stdERR.go 2>/tmp/stdError
This is Standard output
$ cat /tmp/stdError
Please give me one argument!
```



The number after the name of a UNIX program or system call refers to the section of the manual its page belongs to. Although most of the names can be found only once in the manual pages, which means that putting the section number is not required, there are names that can be located in multiple sections because they have multiple meanings, such as `crontab(1)` and `crontab(5)`. Therefore, if you try to retrieve the manual page of a name with multiple meanings without stating its section number, you will get the entry that has the smallest section number.

Similarly, you can discard error output by redirecting it to the `/dev/null` device, which is like telling UNIX to completely ignore it:

```
$ go run stdERR.go 2>/dev/null
This is Standard output
```

In the two examples, we redirected the file descriptor of standard error into a file and `/dev/null`, respectively. If you want to save both standard output and standard error to the same file, you can redirect the file descriptor of standard error (2) to the file descriptor of standard output (1). The following command shows the technique, which is pretty common in UNIX systems:

```
$ go run stdERR.go >/tmp/output 2>&1
$ cat /tmp/output
This is Standard output
Please give me one argument!
```

Finally, you can send both standard output and standard error to `/dev/null` as follows:

```
$ go run stdERR.go >/dev/null 2>&1
```

Writing to log files

The `log` package allows you to send log messages to the system logging service of your UNIX machine, whereas the `syslog` Go package, which is part of the `log` package, allows you to define the **logging level** and the **logging facility** your Go program will use.

Usually, most system log files of a UNIX OS can be found under the `/var/log` directory. However, the log files of many popular services, such as Apache and Nginx, can be found elsewhere, depending on their configuration.

Generally speaking, using a log file to write some information is considered a better practice than writing the same output on the screen for two reasons: firstly, because the output does not get lost as it is stored on a file, and secondly, because you can search and process log files using UNIX tools, such as `grep(1)`, `awk(1)`, and `sed(1)`, which cannot be done when messages are printed on a terminal window.

The `log` package offers many functions for sending output to the syslog server of a UNIX machine. The list of functions includes `log.Printf()`, `log.Print()`, `log.Println()`, `log.Fatalf()`, `log.Fatalln()`, `log.Panic()`, `log.Panicln()`, and `log.Panicf()`.



Please note that logging functions can be extremely handy for debugging your programs, especially server processes written in Go, so you should not underestimate their power.

Logging levels

The **logging level** is a value that specifies the severity of the log entry. There are various logging levels, including *debug*, *info*, *notice*, *warning*, *err*, *crit*, *alert*, and *emerg*, in reverse order of severity.

Logging facilities

A **logging facility** is like a category used for logging information. The value of the logging facility part can be one of *auth*, *authpriv*, *cron*, *daemon*, *kern*, *lpr*, *mail*, *mark*, *news*, *syslog*, *user*, *UUCP*, *local0*, *local1*, *local2*, *local3*, *local4*, *local5*, *local6*, or *local7* and is defined inside `/etc/syslog.conf`, `/etc/rsyslog.conf` or another appropriate file depending on the server process used for system logging on your UNIX machine.

This means that if a logging facility is not defined and therefore handled, the log messages you send to it might get ignored and therefore lost.

Log servers

All UNIX machines have a separate server process that is responsible for receiving logging data and writing it to log files. There are various log servers that work on UNIX machines. However, only two of them are used on most UNIX variants: `syslogd(8)` and `rsyslogd(8)`.

On macOS machines, the name of the process is `syslogd(8)`. On the other hand, most Linux machines use `rsyslogd(8)`, which is an improved and more reliable version of `syslogd(8)`, which was the original UNIX system utility for message logging.

However, despite the UNIX variant you are using or the name of the server process used for logging, logging works the same way on every UNIX machine and therefore does not affect the Go code that you will write.

The configuration file of `rsyslogd(8)` is usually named `rsyslog.conf` and is located in `/etc`. The contents of a `rsyslog.conf` configuration file, without the lines with comments and lines starting with `$`, might look like the following:

```
$ grep -v '^#' /etc/rsyslog.conf | grep -v '^$' | grep -v '^$'
auth,authpriv.*          /var/log/auth.log
*.*;auth,authpriv.none   -/var/log/syslog
daemon.*                 -/var/log/daemon.log
kern.*                   -/var/log/kern.log
lpr.*                    -/var/log/lpr.log
mail.*                   -/var/log/mail.log
user.*                   -/var/log/user.log
mail.info                -/var/log/mail.info
mail.warn                -/var/log/mail.warn
mail.err                 /var/log/mail.err
news.crit                /var/log/news/news.crit
news.err                 /var/log/news/news.err
news.notice              -/var/log/news/news.notice
*.=debug;\
    auth,authpriv.none;\
    news.none;mail.none   -/var/log/debug
*.=info;*.=notice;*.=warn;\
    auth,authpriv.none;\
    cron,daemon.none;\
    mail,news.none        -/var/log/messages
*.emerg                  :omusrmsg:*
```

```
daemon.*;mail.*;\
news.err;\
*.=debug;*.=info;\
*.=notice;*.=warn    | /dev/xconsole
local7.* /var/log/cisco.log
```

So, in order to send your logging information to `/var/log/cisco.log`, you will need to use the `local7` logging facility. The star character after the name of the facility tells the logging server to catch every logging level that goes to the `local7` logging facility and write it to `/var/log/cisco.log`.

The `syslogd(8)` server has a pretty similar configuration file that is usually `/etc/syslog.conf`. On macOS High Sierra, the `/etc/syslog.conf` file is almost empty and has been replaced by `/etc/asl.conf`. Nevertheless, the logic behind the configuration of `/etc/syslog.conf`, `/etc/rsyslog.conf`, and `/etc/asl.conf` is the same.

A Go program that sends information to log files

The Go code of `logFiles.go` will explain the use of the `log` and `log/syslog` packages to write to the system log files.



Please note that the `log/syslog` package is not implemented on the Microsoft Windows version of Go.

The first part of `logFiles.go` is as follows:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
    "os"
    "path/filepath"
)

func main() {
    programName := filepath.Base(os.Args[0])
    syslog, err := syslog.New(syslog.LOG_INFO|syslog.LOG_LOCAL7,
        programName)
```

The first parameter to the `syslog.New()` function is the priority, which is a combination of the logging facility and the logging level. Therefore, a priority of `LOG_NOTICE | LOG_MAIL`, which is mentioned as an example, will send notice logging level messages to the `MAIL` logging facility.

As a result, the preceding code sets the default logging to the `local7` logging facility using the `info` logging level. The second parameter to the `syslog.New()` function is the name of the process that will appear on the logs as the sender of the message. Generally speaking, it is considered a good practice to use the real name of the executable in order to be able to easily find the information you want in the log files at another time.

The second part of the program contains the following Go code:

```
if err != nil {
    log.Fatal(err)
} else {
    log.SetOutput(sysLog)
}
log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")
```

After the call to `syslog.New()`, you will have to check the `error` variable that is returned from it so that you can make sure that everything is fine. If everything is OK, which means that the value of the `error` variable is equal to `nil`, you call the `log.SetOutput()` function, which sets the output destination of the default logger, which, in this case, is the logger you created earlier on (`sysLog`). Then, you can use `log.Println()` to send information to the log server.

The third part of `logFiles.go` comes with the following code:

```
sysLog, err = syslog.New(syslog.LOG_MAIL, "Some program!")
if err != nil {
    log.Fatal(err)
} else {
    log.SetOutput(sysLog)
}

log.Println("LOG_MAIL: Logging in Go!")
fmt.Println("Will you see this?")
}
```

The last part shows that you can change the logging configuration in your programs as many times as you want and that you can still use `fmt.Println()` for printing output on the screen.

The execution of `logFiles.go` will create the following output on the screen of a Debian Linux machine:

```
$ go run logFiles.go
Broadcast message from systemd-journald@mail (Tue 2017-10-17 20:06:08
EEST):
logFiles[23688]: Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
Message from syslogd@mail at Oct 17 20:06:08 ...
Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
Will you see this?
```

Executing the same Go code on a macOS High Sierra machine generated the following output:

```
$ go run logFiles.go
Will you see this?
```

Please bear in mind that most UNIX machines store logging information in more than one log file, which is also the case with the Debian Linux machine used in this section. As a result, `logFiles.go` sends its output to multiple log files, which can be verified by the output of the following shell commands:

```
$ grep LOG_MAIL /var/log/mail.log
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
$ grep LOG_LOCAL7 /var/log/cisco.log
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
$ grep LOG_ /var/log/syslog
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
```

The preceding output shows that the message of the `log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")` statement was written on both `/var/log/cisco.log` and `/var/log/syslog`, whereas the message of the `log.Println("LOG_MAIL: Logging in Go!")` statement was written on both `/var/log/syslog` and `/var/log/mail.log`.

The important thing to remember from this section is that if the logging server of a UNIX machine is not configured to catch all logging facilities, some of the log entries you send to it might get discarded without any warnings.

About log.Fatal()

In this section, you will see the `log.Fatal()` function in action. The `log.Fatal()` function is used when something really bad has happened and you just want to exit your program as fast as possible after reporting the bad situation.

The use of `log.Fatal()` is illustrated in the `logFatal.go` program, which contains the following Go code:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    syslog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some
program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(syslog)
    }

    log.Fatal(syslog)
    fmt.Println("Will you see this?")
}
```

Executing `log.Fatal()` will create the following output:

```
$ go run logFatal.go
exit status 1
```

As you can easily understand, the use of `log.Fatal()` terminates a Go program at the point where `log.Fatal()` was called, which is the reason that you did not see the output from the `fmt.Println("Will you see this?")` statement.

However, because of the parameters of the `syslog.New()` call, a log entry has been added to the log file that is related to mail, which is `/var/log/mail.log`:

```
$ grep "Some program" /var/log/mail.log
Jan 10 21:29:34 iMac Some program![7123]: 2019/01/10 21:29:34 &{17 Some
program! iMac.local {0 0} 0xc00000c220}
```

About log.Panic()

There are situations where a program will fail for good and you want to have as much information about the failure as possible.

In such difficult times, you might consider using `log.Panic()`, which is the logging function that is illustrated in this section using the Go code of `logPanic.go`.

The Go code of `logPanic.go` is as follows:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    syslog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL, "Some
program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Panic(sysLog)
    fmt.Println("Will you see this?")
}
```

Executing `logPanic.go` on macOS Mojave will produce the following output:

```
$ go run logPanic.go
panic: &{17 Some program! iMac.local {0 0} 0xc0000b21e0}
goroutine 1 [running]:
log.Panic(0xc00004ef68, 0x1, 0x1)
    /usr/local/Cellar/go/1.11.4/libexec/src/log/log.go:326 +0xc0
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch01/logPanic.go:17 +0xd6
exit status 2
```

Running the same program on a Debian Linux with Go version 1.3.3 will generate the following output:

```
$ go run logPanic.go
```

```

panic: &{17 Some program! mail {0 0} 0xc2080400e0}
goroutine 16 [running]:
runtime.panic(0x4ec360, 0xc208000320)
    /usr/lib/go/src/pkg/runtime/panic.c:279 +0xf5
log.Panic(0xc208055f20, 0x1, 0x1)
    /usr/lib/go/src/pkg/log/log.go:307 +0xb6
main.main()
    /home/mtsouk/Desktop/masterGo/ch/ch1/code/logPanic.go:17 +0x169
goroutine 17 [runnable]:
runtime.MHeap_Scavenger()
    /usr/lib/go/src/pkg/runtime/mheap.c:507
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 18 [runnable]:
bgsweep()
    /usr/lib/go/src/pkg/runtime/mgc0.c:1976
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 19 [runnable]:
runfinq()
    /usr/lib/go/src/pkg/runtime/mgc0.c:2606
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
exit status 2

```

So, the output of `log.Panic()` includes additional low-level information that will hopefully help you to resolve difficult situations that happened in your Go code.

Analogous to the `log.Fatal()` function, the use of the `log.Panic()` function will add an entry to the proper log file and will immediately terminate the Go program.

Writing to a custom log file

Sometimes, you just need to write your logging data in a file of your choice. This can happen for many reasons, including writing debugging data, which sometimes can be too much, without messing with the system log files, keeping your own logging data separate from system logs in order to transfer it or store it in a database, and storing your data using a different format. This subsection will teach you how to write to a custom log file.

The name of the Go utility will be `customLog.go`, and the log file used will be `/tmp/mGo.log`.

The Go code of `customLog.go` will be presented in three parts. The first part is as follows:

```
package main
```

```
import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"
```

The path of the log file is hardcoded into `customLog.go` using a global variable named `LOGFILE`. For the purposes of this chapter, that log file resides inside the `/tmp` directory, which is not the usual place for storing data because usually, the `/tmp` directory is emptied after each system reboot. However, at this point, this will save you from having to execute `customLog.go` with root privileges and from putting unnecessary files into system directories. If you ever decide to use the code of `customLog.go` in a real application, you should change that path into something more rational.

The second part of `customLog.go` is as follows:

```
func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY,
0644)

    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

Here, you create a new log file using `os.OpenFile()` using the desired UNIX file permissions (0644).

The last part of `customLog.go` is the following:

```
iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)

iLog.SetFlags(log.LstdFlags)
iLog.Println("Hello there!")
iLog.Println("Another log entry!")
}
```

If you look at the documentation page of the `log` package, which, among other places, can be found at <https://golang.org/pkg/log/>, you will see that the `SetFlags` function allows you to set the output flags (options) for the current logger. The default values as defined by `LstdFlags` are `Ldate` and `Ltime`, which means that you will get the current date and the time in each log entry you write in your log file.

Executing `customLog.go` will generate no visible output. However, after executing it twice, the contents of `/tmp/mGo.log` will be as follows:

```
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
customLog 2019/01/10 18:16:09 Another log entry!
$ go run customLog.go
$ cat /tmp/mGo.log
customLog 2019/01/10 18:16:09 Hello there!
customLog 2019/01/10 18:16:09 Another log entry!
customLog 2019/01/10 18:16:17 Hello there!
customLog 2019/01/10 18:16:17 Another log entry!
```

Printing line numbers in log entries

In this section, you are going to learn how to print the line number of the source file that executed the statement that wrote the log entry to a log file using the Go code of `customLogLineNumber.go`. This will be presented in two parts. The first part is as follows:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

func main() {
    f, err := os.OpenFile(LOGFILE, os.O_APPEND|os.O_CREATE|os.O_WRONLY,
        0644)

    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

So far, there is nothing special when compared to the code of `customLog.go`.

The remaining Go code of `customLogLineNumber.go` is as follows:

```
iLog := log.New(f, "customLogLineNumber ", log.LstdFlags)
iLog.SetFlags(log.LstdFlags | log.Lshortfile)
iLog.Println("Hello there!")
iLog.Println("Another log entry!")
}
```

All the magic happens with the `iLog.SetFlags(log.LstdFlags | log.Lshortfile)` statement, which, apart from `log.LstdFlags`, also includes `log.Lshortfile`. The latter flag adds the full filename as well as the line number of the Go statement that printed the log entry in the log entry itself.

Executing `customLogLineNumber.go` will generate no visible output. However, after two executions of `customLogLineNumber.go`, the contents of the `/tmp/mGo.log` log file will be similar to the following:

```
$ go run customLogLineNumber.go
$ cat /tmp/mGo.log
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:26: Hello
there!
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:27: Another
log entry!
$ go run customLogLineNumber.go
$ cat /tmp/mGo.log
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:26: Hello
there!
customLogLineNumber 2019/01/10 18:25:14 customLogLineNumber.go:27: Another
log entry!
customLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:26: Hello
there!
customLogLineNumber 2019/01/10 18:25:23 customLogLineNumber.go:27: Another
log entry!
```

As you can see, using long names for your command-line utilities makes your log files difficult to read.



In Chapter 2, *Understanding Go Internals*, you will learn how to use the `defer` keyword for printing the log messages of a Go function more elegantly.

Error handling in Go

Errors and error handling are two very important Go topics. Go likes error messages so much that it has a separate data type for errors, named `error`. This also means that you can easily create your own **error messages** if you find that what Go gives you is not adequate.

You will most likely need to create and handle your own errors when you are developing your own Go packages.

Please note that having an error condition is one thing, but deciding how to react to an error condition is a totally different thing. Putting it simply, not all error conditions are created equal, which means that some error conditions might require that you immediately stop the execution of a program, whereas other error situations might require printing a warning message for the user to see while continuing the execution of the program. It is up to the developer to use common sense and decide what to do with each `error` value the program might get.



Errors in Go are not like exceptions or errors in other programming languages; they are normal Go objects that get returned from functions or methods just like any other value.

The error data type

There are many scenarios where you might end up having to deal with a new error case while you are developing your own Go application. The `error` data type is here to help you to define your own errors.

This subsection will teach you how to create your own `error` variables. As you will see, in order to create a new `error` variable, you will need to call the `New()` function of the `errors` standard Go package.

The example Go code to illustrate this process can be found in `newError.go` and will be presented in two parts. The first part of the program is as follows:

```
package main

import (
    "errors"
    "fmt"
)

func returnError(a, b int) error {
```

```
    if a == b {
        err := errors.New("Error in returnError() function!")
        return err
    } else {
        return nil
    }
}
```

There are many interesting things happening here. First of all, you can see the definition of a Go function other than `main()` for the first time in this book. The name of this new naive function is `returnError()`. Additionally, you can see the `errors.New()` function in action, which takes a `string` value as its parameter. Lastly, if a function should return an error variable but there is not an error to report, it returns `nil` instead.



You will learn more about the various types of Go functions in Chapter 6, *What You Might Not Know About Go Packages and Go Functions*.

The second part of `newError.go` is the following:

```
func main() {
    err := returnError(1, 2)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    err = returnError(10, 10)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    if err.Error() == "Error in returnError() function!" {
        fmt.Println("!!")
    }
}
```

As the code illustrates, most of the time, you need to check whether an error variable is equal to `nil` and then act accordingly. What is also presented here is the use of the `errors.Error()` function, which allows you to convert an error variable into a string variable. This function lets you compare an error variable with a string variable.



It is considered good practice to send your error messages to the logging service of your UNIX machine, especially when a Go program is a server or some other critical application. However, the code of this book will not follow this principle everywhere in order to avoid filling your log files with unnecessary data.

Executing `newError.go` will produce the following output:

```
$ go run newError.go
returnError() ended normally!
Error in returnError() function!
!!
```

If you try to compare an `error` variable with a `string` variable without first converting the `error` variable to a `string` variable, the Go compiler will create the following error message:

```
# command-line-arguments
./newError.go:33:9: invalid operation: err == "Error in returnError()
function!" (mismatched types error and string)
```

Error handling

Error handling is a very important feature of Go because almost all Go functions return an error message or `nil`, which is the Go way of saying whether there was an error condition while executing a function. You will most likely get tired of seeing the following Go code not only in this book but also in every other Go program you can find on the internet:

```
if err != nil {
    fmt.Println(err)
    os.Exit(10)
}
```



Please do not confuse error handling with printing to error output because they are two totally different things. The former has to do with Go code that handles error conditions, whereas the latter has to do with writing something to the standard error file descriptor.

The preceding code prints the generated error message on the screen and exits your program using `os.Exit()`. Please note that you can also exit your program by calling the `return` keyword inside the `main()` function. Generally speaking, calling `os.Exit()` from a function other than `main()` is considered a bad practice. Functions other than `main()` tend to return the error message before exiting, which is handled by the calling function.

Should you wish to send the error message to the logging service instead of the screen, you should use the following variation of the preceding Go code:

```
if err != nil {
    log.Println(err)
    os.Exit(10)
}
```

Lastly, there is another variation of the preceding code that is used when something really bad has happened and you want to terminate the program:

```
if err != nil {
    panic(err)
    os.Exit(10)
}
```

Panic is a built-in Go function that stops the execution of a program and starts panicking! If you find yourself using `panic` too often, you might want to reconsider your Go implementation. People tend to avoid panic situations in favor of errors wherever possible.

As you will see in the next chapter, Go also offers the `recover` function, which might be able to save you from some bad situations. For now, you will need to wait for Chapter 2, *Understanding Go Internals*, to learn more about the power of the `panic` and `recover` function duo.

It is now time to see a Go program that not only handles error messages generated by standard Go functions, but also defines its own error message. The name of the program is `errors.go` and it will be presented to you in five parts. As you will see, the `errors.go` utility tries to improve the functionality of the `cla.go` program you saw earlier in this chapter by examining whether its command-line arguments are acceptable floats.

The first part of the program is as follows:

```
package main

import (
    "errors"
    "fmt"
    "os"
    "strconv"
)
```

This part of `errors.go` contains the expected `import` statements.

The second portion of `errors.go` comes with the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    var err error = errors.New("An error")
    k := 1
    var n float64
```

Here, you create a new error variable named `err` in order to initialize it with your own value.

The third part of the program is as follows:

```
    for err != nil {
        if k >= len(arguments) {
            fmt.Println("None of the arguments is a float!")
            return
        }
        n, err = strconv.ParseFloat(arguments[k], 64)
        k++
    }

    min, max := n, n
```

This is the trickiest part of the program because if the first command-line argument is not a proper float, you will need to check the next one and keep checking until you find a suitable command-line argument. If none of the command-line arguments are in the correct format, `errors.go` will terminate and print a message on the screen. All this checking happens by examining the error value that is returned by `strconv.ParseFloat()`. All this code is just for the accurate initialization of the `min` and `max` variables.

The fourth portion of the program comes with the following Go code:

```
    for i := 2; i < len(arguments); i++ {
        n, err := strconv.ParseFloat(arguments[i], 64)
        if err == nil {
            if n < min {
                min = n
            }
            if n > max {
                max = n
            }
        }
    }
```

```
    }  
}
```

Here, you just process all the right command-line arguments in order to find the minimum and maximum floats among them.

Finally, the last code portion of the program deals with just printing out the current values of the `min` and `max` variables:

```
    fmt.Println("Min:", min)  
    fmt.Println("Max:", max)  
}
```

As you can see from the Go code of `errors.go`, most of its code is about error handling rather than about the actual functionality of the program. Unfortunately, this is the case with most modern software developed in Go, as well as most other programming languages.

If you execute `errors.go`, you will get the following kind of output:

```
$ go run errors.go a b c  
None of the arguments is a float!  
$ go run errors.go b c 1 2 3 c -1 100 -200 a  
Min: -200  
Max: 100
```

Using Docker

In the last section of this chapter, you will learn how to use a Docker image in order to compile and execute your Go code inside the Docker image.

As you might already know, everything in Docker begins with a Docker image; you can either build your own Docker image from scratch or begin with an existing one. For the purposes of this section, the base Docker image will be downloaded from Docker Hub and we will continue with building the Go version of the `Hello World!` program inside that Docker image.

The contents of the `Dockerfile` that will be used are as follows:

```
FROM golang:alpine  
  
RUN mkdir /files  
COPY hw.go /files  
WORKDIR /files
```

```
RUN go build -o /files/hw hw.go
ENTRYPOINT ["/files/hw"]
```

The first line defines the Docker image that will be used. The remaining three commands create a new directory in the Docker image, copy a file (`hw.go`) from the current user directory into the Docker image, and change the current working directory of the Docker image, respectively. The last two commands create a binary executable from the Go source file and specify the path of the binary file that will be executed when you run that Docker image.

So, how do you use that `Dockerfile`? Provided that a file named `hw.go` exists in the current working directory, you can build a new Docker image as follows:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
alpine: Pulling from library/golang
cd784148e348: Pull complete
7e273b0dfc44: Pull complete
952c3806fd1a: Pull complete
ee1f873f86f9: Pull complete
7172cd197d12: Pull complete
Digest:
sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb018ad77d647
Status: Downloaded newer image for golang:alpine
----> f56365ec0638
Step 2/6 : RUN mkdir /files
----> Running in 18fa7784d82c
Removing intermediate container 18fa7784d82c
----> 9360e95d7cb4
Step 3/6 : COPY hw.go /files
----> 680517bc4aa3
Step 4/6 : WORKDIR /files
----> Running in f3f678fcc38d
Removing intermediate container f3f678fcc38d
----> 640117aea82f
Step 5/6 : RUN go build -o /files/hw hw.go
----> Running in 271cae1fa7f9
Removing intermediate container 271cae1fa7f9
----> dc7852b6aeeb
Step 6/6 : ENTRYPOINT ["/files/hw"]
----> Running in cdadf286f025
Removing intermediate container cdadf286f025
----> 9bec016712c4
Successfully built 9bec016712c4
Successfully tagged go_hw:v1
```

The name of the newly created Docker image is `go_hw:v1`.

If the `golang:alpine` Docker image is already present on your computer, the output of the preceding command will be as follows:

```
$ docker build -t go_hw:v1 .
Sending build context to Docker daemon 2.237MB
Step 1/6 : FROM golang:alpine
----> f56365ec0638
Step 2/6 : RUN mkdir /files
----> Running in 982e6883bb13
Removing intermediate container 982e6883bb13
----> 0632577d852c
Step 3/6 : COPY hw.go /files
----> 68a0feb2e7dc
Step 4/6 : WORKDIR /files
----> Running in d7d4d0c846c2
Removing intermediate container d7d4d0c846c2
----> 6597a7cb3882
Step 5/6 : RUN go build -o /files/hw hw.go
----> Running in 324400d532e0
Removing intermediate container 324400d532e0
----> 5496dd3d09d1
Step 6/6 : ENTRYPOINT ["/files/hw"]
----> Running in bbd24840d6d4
Removing intermediate container bbd24840d6d4
----> 5a0d2473aa96
Successfully built 5a0d2473aa96
Successfully tagged go_hw:v1
```

You can verify that the `go_hw:v1` Docker image exists on your machine as follows:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
go_hw	v1	9bec016712c4	About a minute ago	312MB
golang	alpine	f56365ec0638	11 days ago	310MB

The contents of the `hw.go` file are as follows:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello World!")
}
```

You can use a Docker image that is on your local computer as follows:

```
$ docker run go_hw:v1
Hello World!
```

There are other more complex ways to execute a Docker image, but for such a naive Docker image, this is the simplest way to use it.

If you want, you can store (**push**) a Docker image at a Docker registry on the internet in order to be able to retrieve it (**pull**) from there afterward.

Docker Hub can be such a place, provided that you have a Docker Hub account, which is easy to create and free. So, after creating a Docker Hub account, you should execute the following commands on your UNIX machine and push that image to Docker Hub:

```
$ docker login
Authenticating with existing credentials...
Login Succeeded
$ docker tag go_hw:v1 "mactsouk/go_hw:v1"
$ docker push "mactsouk/go_hw:v1"
The push refers to repository [docker.io/mactsouk/go_hw]
bdb6946938e3: Pushed
99e21c42e35d: Pushed
0257968d27b2: Pushed
e121936484eb: Pushed
61b145086eb8: Pushed
789935042c6f: Pushed
b14874cfef59: Pushed
7bff100f35cb: Pushed
v1: digest:
sha256:c179d5d48a51b74b0883e582d53bf861c6884743eb51d9b77855949b5d91dd
e1 size: 1988
```

The first command is needed to log in to Docker Hub and should be executed only once. The `docker tag` command is needed for specifying the name that a local image will have on Docker Hub and should be executed before the `docker push` command. The last command sends the desired Docker image to Docker Hub, hence the rich output it generates. If you make your Docker image public, anyone will be able to pull it and use it.

You can delete one or more Docker images from your local UNIX machine in many ways. One of them is by using the `IMAGE ID` of a Docker image:

```
$ docker rmi 5a0d2473aa96 f56365ec0638
Untagged: go_hw:v1
Deleted:
sha256:5a0d2473aa96bcdafbef92751a0e1c1bf146848966c8c971f462eb1eb242d2
a6
```

```
Deleted:
sha256:5496dd3d09d13c63bf7a9ac52b90bb812690cdfd33cfc3340509f9bfe6215c
48
Deleted:
sha256:598c4e474b123eccb84f41620d2568665b88a8f176a21342030917576b9d82
a8
Deleted:
sha256:6597a7cb3882b73855d12111787bd956a9ec3abb11d9915d32f2bba4d0e92e
c6
Deleted:
sha256:68a0feb2e7dc5a139eaa7ca04e54c20e34b7d06df30bcd4934ad6511361f2c
b8
Deleted:
sha256:c04452ea9f45d85a999bdc54b55ca75b6b196320c021d777ec1f766d115aa5
14
Deleted:
sha256:0632577d852c4f9b66c0eff2481ba06c49437e447761d655073eb034fa0ac3
33
Deleted:
sha256:52efd0fa2950c8f3c3e2e44fbc4eb076c92c0f85fff46a07e060f5974c1007
a9
Untagged: golang:alpine
Untagged:
golang@sha256:198cb8c94b9ee6941ce6d58f29aadb855f64600918ce602cdeacb01
8ad77d647
Deleted:
sha256:f56365ec0638b16b752af4bf17e6098f2fda027f8a71886d6849342266cc3a
b7
Deleted:
sha256:d6a4b196ed79e7ff124b547431f77e92dce9650037e76da294b3b3aded709b
dd
Deleted:
sha256:f509ec77b9b2390c745afd76cd8dd86977c86e9ff377d5663b42b664357c35
22
Deleted:
sha256:1ee98fa99e925362ef980e651c5a685ad04cef41dd80df9be59f158cf9e529
51
Deleted:
sha256:78c8e55f8cb4c661582af874153f88c2587a034ee32d21cb57ac1fef51c610
9e
Deleted:
sha256:7bfff100f35cb359a368537bb07829b055fe8e0b1cb01085a3a628ae9c187c7
b8
```



Docker is a huge and really important topic that we will revisit in several chapters of this book.

Exercises and links

- Visit the Go site: <https://golang.org/>.
- Visit the site of Docker: <https://www.docker.com/>.
- Visit the Docker Hub site: <https://hub.docker.com/>.
- Go 2 Draft Designs: <https://blog.golang.org/go2draft>.
- Browse the Go documentation site: <https://golang.org/doc/>.
- Visit the documentation of the `log` package at <https://golang.org/pkg/log/>.
- Visit the documentation of the `log/syslog` package at <https://golang.org/pkg/log/syslog/>.
- Visit the documentation of the `os` package at <https://golang.org/pkg/os/>.
- Have a look at <https://golang.org/cmd/gofmt/>, which is the documentation page of the `gofmt` tool that is used for formatting Go code.
- Write a Go program that finds the sum of all command-line arguments that are valid numbers.
- Write a Go program that finds the average value of floating-point numbers that are given as command-line arguments.
- Write a Go program that keeps reading integers until it gets the word `END` as input.
- Can you modify `customLog.go` in order to write its log data into two log files at the same time? You might need to read Chapter 8, *Telling a UNIX System What to Do*, for help.
- If you are working on a Mac machine, check the **TextMate** editor at <http://macromates.com/>, as well as **BEdit** at <https://www.barebones.com/products/bbedit/>.
- Visit the documentation page of the `fmt` package at <https://golang.org/pkg/fmt/> to learn more about verbs and the available functions.
- Please visit <https://blog.golang.org/why-generics> to learn more about Go and Generics.

Summary

This chapter talked about many interesting Go topics, including compiling Go code, working with standard input, standard output, and standard error in Go, processing command-line arguments, printing on the screen, and using the logging service of a UNIX system, as well as error handling and some general information about Go. You should consider all these topics as foundational information about Go.

The next chapter is all about the internals of Go, which includes talking about garbage collection, the Go compiler, calling C code from Go, the `defer` keyword, the Go assembler, and WebAssembly, as well as `panic` and `recover`.

2

Understanding Go Internals

All the Go features that you learned in the previous chapter are extremely handy and you will be using them all the time. However, there is nothing as rewarding as being able to see and understand what is going on in the background and how Go operates behind the scenes.

In this chapter, you are going to learn about the Go garbage collector and the way it works. Additionally, you will find out how to call C code from your Go programs, which you might find indispensable in some situations. However, you will not need to use this capability too often because Go is a very capable programming language.

Furthermore, you will learn how to call Go code from your C programs, along with how to use the `panic()` and `recover()` functions and the `defer` keyword.

This chapter covers:

- The Go compiler
- How garbage collection works in Go
- How to check the operation of the garbage collector
- Calling C code from your Go code
- Calling Go code from a C program
- The `panic()` and `recover()` functions
- The `unsafe` package
- The handy, yet tricky, `defer` keyword
- The `strace(1)` Linux utility
- The `dtrace(1)` utility, which can be found in FreeBSD systems, including macOS Mojave
- Finding out information about your Go environment
- The node trees created by Go
- Creating WebAssembly code from Go
- The Go assembler

The Go compiler

The Go compiler is executed with the help of the `go` tool, which does many more things than just generating executable files.



The `unsafe.go` file used in this section does not contain any special code – the presented commands will work on every valid Go source file.

You can compile a Go source file using the `go tool compile` command. What you will get is an **object file**, which is a file with the `.o` file extension. This is illustrated in the output of the next commands, which were executed on a macOS Mojave machine:

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:39 unsafe.o
$ file unsafe.o
unsafe.o: current ar archive
```

An object file is a file that contains **object code**, which is machine code in relocatable format that, most of the time, is not directly executable. The biggest advantage of the relocatable format is that it requires as low memory as possible during the linking phase.

If you use the `-pack` command-line flag when executing `go tool compile`, you will get an **archive file** instead of an object file:

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r--  1 mtsouk  staff  6926 Jan 22 21:40 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

An archive file is a binary file that contains one or more files, and it is mainly used for grouping multiple files into a single file. One of these formats is `ar`, which is used by Go.

You can list the contents of an `.a` archive file as follows:

```
$ ar t unsafe.a
___.PKGDEF
_go_.o
```

Another truly valuable command-line flag of the `go tool compile` command that is worth mentioning is `-race`, which allows you to detect **race conditions**. You will learn more about what a race condition is and why you want to avoid it in [Chapter 10, Concurrency in Go – Advanced Topics](#).

You will find more uses of the `go tool compile` command near the end of this chapter when we will talk about assembly language and node trees. However, for a tester, try executing the next command:

```
$ go tool compile -S unsafe.go
```

The preceding command generates lots of output that you might find difficult to understand, which means that Go does a pretty good job of hiding any unnecessary complexities, unless you ask for them!

Garbage collection

Garbage collection is the process of freeing up memory space that is not being used. In other words, the garbage collector sees which objects are out of scope and cannot be referenced anymore, and frees the memory space they consume. This process happens in a concurrent way while a Go program is running and not before or after the execution of the program. The documentation of the Go garbage collector implementation states the following:

"The GC runs concurrently with mutator threads, is type accurate (also known as precise), allows multiple GC threads to run in parallel. It is a concurrent mark and sweep that uses a write barrier. It is non-generational and non-compacting. Allocation is done using size segregated per P allocation areas to minimize fragmentation while eliminating locks in the common case."

There is lots of terminology here that will be explained in a while. But first, I will show you a way to look at some parameters of the garbage collection process.

Fortunately, the Go standard library offers functions that allow you to study the operation of the garbage collector and learn more about what the garbage collector secretly does. The relevant code is saved as `gColl.go` and will be presented in three parts.

The first code segment of `gColl.go` is the following:

```
package main

import (
    "fmt"
    "runtime"
    "time"
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("-----")
}
```

Note that each time you need to get the more recent garbage collection statistics, you will need to call the `runtime.ReadMemStats()` function. The purpose of the `printStats()` function is to avoid writing the same Go code all the time.

The second part of the program is the following:

```
func main() {
    var mem runtime.MemStats
    printStats(mem)

    for i := 0; i < 10; i++ {
        s := make([]byte, 50000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
    }
    printStats(mem)
}
```

The `for` loop creates many big Go **slices** in order to allocate large amounts of memory and trigger the garbage collector.

The last part of `gColl.go` comes with the next Go code, which does more memory allocations using Go slices:

```
    for i := 0; i < 10; i++ {
        s := make([]byte, 100000000)
        if s == nil {
            fmt.Println("Operation failed!")
        }
        time.Sleep(5 * time.Second)
    }
    printStats(mem)
}
```

The output of `gColl.go` on a macOS Mojave machine is next:

```
$ go run gColl.go
mem.Alloc: 66024
mem.TotalAlloc: 66024
mem.HeapAlloc: 66024
mem.NumGC: 0
-----
mem.Alloc: 50078496
mem.TotalAlloc: 500117056
mem.HeapAlloc: 50078496
mem.NumGC: 10
-----
mem.Alloc: 76712
mem.TotalAlloc: 1500199904
mem.HeapAlloc: 76712
mem.NumGC: 20
-----
```

Although you are not going to examine the operation of the Go garbage collector all the time, being able to watch the way that it operates on a slow application can save so much time in the long run. I can assure you that you will not regret the time you spend learning about garbage collection in general and, more specifically, about the way the Go garbage collector works.

There is a trick that allows you to get even more detailed output about the way the Go garbage collector operates, which is illustrated in the next command:

```
$ GODEBUG=gctrace=1 go run gColl.go
```

So, if you put `GODEBUG=gctrace=1` in front of any `go run` command, Go will print analytical data about the operation of the garbage collector. The data will be in this form:

```
gc 4 @0.025s 0%: 0.002+0.065+0.018 ms clock,  
0.021+0.040/0.057/0.003+0.14 ms cpu, 47->47->0 MB, 48 MB goal, 8 P  
gc 17 @30.103s 0%: 0.004+0.080+0.019 ms clock,  
0.033+0/0.076/0.071+0.15 ms cpu, 95->95->0 MB, 96 MB goal, 8 P
```

The preceding output tells us more information about the heap sizes during the garbage collection process. So, let us take the `47->47->0 MB` trinity of values as an example. The first number is the heap size when the garbage collector is about to run. The second value is the heap size when the garbage collector ends its operation. The last value is the size of the live heap.

The tricolor algorithm

The operation of the Go garbage collector is based on the **tricolor algorithm**.



Please note that the tricolor algorithm is not unique to Go and can be used in other programming languages.

Strictly speaking, the official name for the algorithm used in Go is the **tricolor mark-and-sweep algorithm**. It can work concurrently with the program and uses a **write barrier**. This means that when a Go program runs, the Go scheduler is responsible for the scheduling of the application and the garbage collector. This is as if the Go scheduler has to deal with a regular application with multiple **goroutines**! You will learn more about goroutines and the Go scheduler in [Chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines](#).

The core idea behind this algorithm came from Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens and was first illustrated in a paper named *On-the-Fly Garbage Collection: An Exercise in Cooperation*.

The primary principle behind the tricolor mark-and-sweep algorithm is that it divides the objects of the heap into three different sets according to their color, which is assigned by the algorithm. It is now time to talk about the meaning of each color set. The objects of the **black set** are guaranteed to have no pointers to any object of the **white set**.

However, an object of the white set can have a pointer to an object of the black set because this has no effect on the operation of the garbage collector. The objects of the **gray set** might have pointers to some objects of the white set. Finally, the objects of the white set are the candidates for garbage collection.

Please note that no object can go directly from the black set to the white set, which allows the algorithm to operate and be able to clear the objects on the white set. Additionally, no object of the black set can directly point to an object of the white set.

So, when the garbage collection begins, all objects are white and the garbage collector visits all the root objects and colors them gray. The **roots** are the objects that can be directly accessed by the application, which includes global variables and other things on the stack. These objects mostly depend on the Go code of a particular program.

After that, the garbage collector picks a gray object, makes it black, and starts looking at whether that object has pointers to other objects of the white set. This means that when a gray object is being scanned for pointers to other objects, it is colored black. If that scan discovers that this particular object has one or more pointers to a white object, it puts that white object in the gray set. This process keeps going for as long as there exist objects in the gray set. After that, the objects in the white set are unreachable and their memory space can be reused. Therefore, at this point, the elements of the white set are said to be garbage collected.



Please note that if an object of the gray set becomes unreachable at some point in a garbage collection cycle, it will not be collected at this garbage collection cycle but in the next one! Although this is not an optimal situation, it is not that bad.

During this process, the running application is called the **mutator**. The mutator runs a small function named **write barrier**, which is executed each time a pointer in the heap is modified. If the pointer of an object in the heap is modified, which means that this object is now reachable, the write barrier colors it gray and puts it in the gray set.



The mutator is responsible for the invariant that no element of the black set has a pointer to an element of the white set. This is accomplished with the help of the write barrier function. Failing to accomplish this invariant will ruin the garbage collection process and will most likely crash your program in a pretty bad and undesirable way!

As a result, the heap is pictured as a graph of connected objects, which is also shown in *Figure 2.1*, which demonstrates a single phase of a garbage collection cycle.

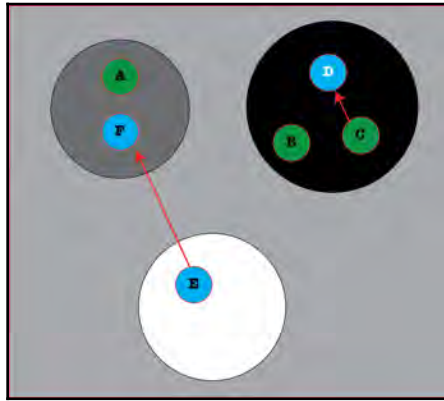


Figure 2.1: The Go garbage collector represents the heap of a program as a graph

So, there are three different colors: black, white, and gray. When the algorithm begins, all objects are colored white. As the algorithm keeps going, white objects are moved into one of the other two sets. The objects that are left in the white set are the ones that are going to be cleared at some point.

In the presented graph, you can see that while object E, which is in the white set, can access object F, it cannot be accessed by any other object because no other object points to object E, which makes it a perfect candidate for garbage collection! Additionally, objects A, B, and C are root objects and are always reachable; therefore, they cannot be garbage collected.

Can you guess what will happen next in that graph? Well, it is not that difficult to realize that the algorithm will have to process the remaining elements of the gray set, which means that both objects A and F will go to the black set. Object A will go to the black set because it is a root element and F will go to the black set because it does not point to any other object while it is in the gray set.

After object A is garbage collected, object F will become unreachable and will be garbage collected in the next cycle of the garbage collector because an unreachable object cannot magically become reachable in the next iteration of the garbage collection cycle.

Go garbage collection can also be applied to variables such as **channels**. When the garbage collector finds out that a channel is unreachable, which is when the channel variable cannot be accessed anymore, it will free its resources even if the channel has not been closed. You will learn more about channels in *Chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines*.

Go allows you to manually initiate garbage collection by putting a `runtime.GC()` statement in your Go code. However, have in mind that `runtime.GC()` will block the caller and it might block the entire program, especially if you are running a very busy Go program with many objects. This mainly happens because you cannot perform garbage collections while everything else is rapidly changing, as this will not give the garbage collector the opportunity to clearly identify the members of the white, black, and gray sets. This garbage collection status is also called a **garbage collection safe-point**.

You can find the long and relatively advanced Go code of the garbage collector at <https://github.com/golang/go/blob/master/src/runtime/mgc.go>, which you can study if you want to learn even more information about the garbage collection operation. You can even make changes to that code if you are brave enough!



Note that the Go garbage collector is always being improved by the Go team. They are trying to make it faster by lowering the number of scans it needs to perform over the data of the three sets. However, despite the various optimizations, the general idea behind the algorithm remains the same.

More about the operation of the Go garbage collector

This section will talk more about the Go garbage collector and present additional information about its activities. The main concern of the Go garbage collector is low latency, which basically means short pauses in its operation in order to have a real-time operation. On the other hand, what a program does is create new objects and manipulate existing objects with pointers all the time. This process can end up creating objects that cannot be accessed anymore because there are no pointers pointing to these objects. These objects are then garbage and wait for the garbage collector to clean them up and free their memory space. After that, the memory space that has been freed is ready to be used again.

The mark-and-sweep algorithm is the simplest algorithm used. The algorithm stops the program execution (**stop-the-world garbage collector**) in order to visit all the accessible objects of the heap of a program and *marks* them. After that, it *sweeps* the inaccessible objects. During the mark phase of the algorithm, each object is marked as white, gray, or black. The children of a gray object are colored gray, whereas the original gray object is now colored black. The sweep phase begins when there are no more gray objects to examine. This technique works because there are no pointers from the black set to the white set, which is a fundamental invariant of the algorithm.

Although the mark-and-sweep algorithm is simple, it suspends the execution of the program while it is running, which means that it adds latency to the actual process. Go tries to lower that particular latency by running the garbage collector as a concurrent process and by using the tricolor algorithm described in the previous section. However, other processes can move pointers or create new objects while the garbage collector runs concurrently. This fact can make things pretty difficult for the garbage collector. As a result, the point that will allow the tricolor algorithm to run concurrently will be when maintaining the fundamental invariant of the mark-and-sweep algorithm: no object of the black set can point to an object of the white set.

The solution to this problem is fixing all the cases that can cause a problem for the algorithm. Therefore, new objects must go to the gray set because this way the fundamental invariant of the mark-and-sweep algorithm cannot be altered. Additionally, when a pointer of the program is moved, you color the object that the pointer points to as gray. The gray set acts like a barrier between the white set and the black set. Finally, each time a pointer is moved, some Go code gets automatically executed, which is the write barrier mentioned earlier, which does some recoloring. The latency introduced by the execution of the write barrier code is the price we have to pay for being able to run the garbage collector concurrently.

Please note that the **Java** programming language has many garbage collectors that are highly configurable with the help of multiple parameters. One of these Java garbage collectors is called **G1** and it is recommended for low-latency applications.



It is really important to remember that the Go garbage collector is a real-time garbage collector that runs concurrently with the other goroutines of a Go program and only optimizes for low latency.

In Chapter 11, *Code Testing, Optimization, and Profiling*, you will learn how to graphically represent the performance of a program, which also includes information about the operations of the Go garbage collector.

Maps, slices, and the Go garbage collector

In this section, I am going to present you with some examples showing why you should be cautious regarding the operation of the garbage collector. The point of this section is to understand that the way you store pointers has a great impact on the performance of the garbage collector, especially when you are dealing with very large amounts of pointers.



The presented examples use pointers, slices, and maps, which are all native Go data types. You will learn more about pointers, slices, and maps in Go in Chapter 3, *Working with Basic Go Data Types*.

Using a slice

The example in this section will use a **slice** to store a large amount of structures. Each structure stores two integer values. The Go code `sliceGC.go` is as follows:

```
package main

import (
    "runtime"
)

type data struct {
    i, j int
}

func main() {
    var N = 40000000
    var structure []data
    for i := 0; i < N; i++ {
        value := int(i)
        structure = append(structure, data{value, value})
    }

    runtime.GC()
    _ = structure[0]
}
```

The last statement (`_ = structure[0]`) is used for preventing the garbage collector from garbage collecting the `structure` variable too early, as it is not referenced or used outside of the `for` loop. The same technique will be used in the three Go programs that follow. Apart from this important detail, a `for` loop is used for putting all values into structures that are stored in the slice.

Using a map with pointers

In this subsection, we are going to use a map for storing all our pointers as integers. The name of the program is `mapStar.go` and it contains the following Go code:

```
package main
```

```
import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]*int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = &value
    }
    runtime.GC()
    _ = myMap[0]
}
```

The name of the map that stores the integer pointers is `myMap`. A `for` loop is used for putting the integer values into the map.

Using a map without pointers

In this subsection, we are going to use a map that stores plain values without pointers. The Go code of `mapNoStar.go` is as follows:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    myMap := make(map[int]int)
    for i := 0; i < N; i++ {
        value := int(i)
        myMap[value] = value
    }
    runtime.GC()
    _ = myMap[0]
}
```

As before, a `for` loop is used for putting the integer values into the map.

Splitting the map

The implementation of this subsection will split the map into a map of maps, which is also called **sharding**. The program of this subsection is saved as `mapSplit.go` and will be presented in two parts. The first part of `mapSplit.go` contains the following Go code:

```
package main

import (
    "runtime"
)

func main() {
    var N = 40000000
    split := make([]map[int]int, 200)
```

This is where the hash of hashed is defined.

The second part is as follows:

```
    for i := range split {
        split[i] = make(map[int]int)
    }
    for i := 0; i < N; i++ {
        value := int(i)
        split[i%200][value] = value
    }
    runtime.GC()
    _ = split[0][0]
}
```

This time, we are using two `for` loops: one `for` loop for creating the hash of hashes and another one for storing the desired data in the hash of hashes.

Comparing the performance of the presented techniques

As all four programs are using huge data structures, they are consuming large amounts of memory. Programs that consume lots of memory space trigger the Go garbage collector more often. So, in this subsection, we are going to compare the performance of each one of these four implementations using the `time(1)` command.

What will be important in the presented output is not the exact numbers but the time difference between the four different approaches. Here we go:

```
$ time go run sliceGC.go
real 1.50s
user 1.72s
sys 0.71s
$ time go run mapStar.go
real 13.62s
user 23.74s
sys 1.89s
$ time go run mapNoStar.go
real 11.41s
user 10.35s
sys 1.15s
$ time go run mapSplit.go
real 10.60s
user 10.01s
sys 0.74s
```

So, it turns out that maps slow down the Go garbage collector whereas slices collaborate much better with it. It should be noted here that this is not a problem with maps but a result of the way the Go garbage collector works. However, unless you are dealing with maps that store huge amounts of data, this problem will not become evident in your programs.



You will learn more about **benchmarking** in Go in Chapter 11, *Code Testing, Optimization, and Profiling*. Additionally, you will learn a more professional way to measure the time it took a command or a program in Go to execute in Chapter 3, *Working with Basic Go Data Types*.

Enough with garbage collection and its quirks; the topic of the next section will be unsafe code and the `unsafe` standard Go package.

Unsafe code

Unsafe code is Go code that bypasses the type safety and the memory security of Go. Most of the time, unsafe code is related to pointers. However, have in mind that using unsafe code can be dangerous for your programs, so if you are not completely sure that you need to use unsafe code in one of your programs, do not use it!

The use of unsafe code will be illustrated in the `unsafe.go` program, which will be presented in three parts.

The first part of `unsafe.go` is next:

```
package main

import (
    "fmt"
    "unsafe"
)
```

As you will notice, in order to use unsafe code, you will need to import the unsafe standard Go package.

The second part of the program comes with the following Go code:

```
func main() {
    var value int64 = 5
    var p1 = &value
    var p2 = (*int32)(unsafe.Pointer(p1))
}
```

Note the use of the `unsafe.Pointer()` function here, which allows us, at our own risk, to create an `int32` pointer named `p2` that points to an `int64` variable named `value`, which is accessed using the `p1` pointer. Any Go pointer can be converted to `unsafe.Pointer`.



A pointer of type `unsafe.Pointer` can override the type system of Go. This is unquestionably fast but it can also be dangerous if used incorrectly or carelessly. Additionally, it gives developers more control over data.

The last part of `unsafe.go` has the next Go code:

```
fmt.Println(*p1: ", *p1)
fmt.Println(*p2: ", *p2)
*p1 = 5434123412312431212
fmt.Println(value)
fmt.Println(*p2: ", *p2)
*p1 = 54341234
fmt.Println(value)
fmt.Println(*p2: ", *p2)
}
```



You can **dereference a pointer** and get, use, or set its value using the star character (*).

If you execute `unsafe.go`, you will have the next output:

```
$ go run unsafe.go
*p1:  5
*p2:  5
5434123412312431212
*p2:  -930866580
54341234
*p2:  54341234
```

What does this output tell us? It tells us that a 32-bit pointer cannot store a 64-bit integer.

As you will see in the next section, the functions of the `unsafe` package can do many more interesting things with memory.

About the `unsafe` package

Now you have seen the `unsafe` package in action, it is a good time to talk more about what makes it a special kind of package. First of all, if you look at the source code of the `unsafe` package, you might be a little surprised. On a macOS Mojave system with Go version 1.11.4 that is installed using Homebrew (<https://brew.sh/>), the source code of the `unsafe` package is located at

`/usr/local/Cellar/go/1.11.4/libexec/src/unsafe/unsafe.go` and its contents without the comments are the following:

```
$ cd /usr/local/Cellar/go/1.11.4/libexec/src/unsafe/
$ grep -v '^//' unsafe.go | grep -v '^$'
package unsafe
type ArbitraryType int
type Pointer *ArbitraryType
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

So, where is the rest of the Go code of the `unsafe` package? The answer to that question is relatively simple: the Go compiler implements the `unsafe` package when you import it in your programs.



Many low-level packages, such as `runtime`, `syscall`, and `os`, constantly use the `unsafe` package.

Another example of the unsafe package

In this subsection, you will learn more things about the `unsafe` package and its capabilities with the help of another small Go program named `moreUnsafe.go`. This will be presented in three parts. What `moreUnsafe.go` does is access all the elements of an array using pointers.

The first part of the program is next:

```
package main

import (
    "fmt"
    "unsafe"
)
```

The second part of `moreUnsafe.go` comes with the next Go code:

```
func main() {
    array := [...]int{0, 1, -2, 3, 4}
    pointer := &array[0]
    fmt.Print(*pointer, " ")
    memoryAddress := uintptr(unsafe.Pointer(pointer)) +
        unsafe.Sizeof(array[0])

    for i := 0; i < len(array)-1; i++ {
        pointer = (*int)(unsafe.Pointer(memoryAddress))
        fmt.Print(*pointer, " ")
        memoryAddress = uintptr(unsafe.Pointer(pointer)) +
            unsafe.Sizeof(array[0])
    }
}
```

At first, the `pointer` variable points to the memory address of `array[0]`, which is the first element of the array of integers. Then, the `pointer` variable that points to an integer value is converted to an `unsafe.Pointer()` and then to an `uintptr`. The result is stored in `memoryAddress`.

The value of `unsafe.Sizeof(array[0])` is what gets you to the next element of the array because this is the memory occupied by each array element. So, that value is added to the `memoryAddress` variable in each iteration of the `for` loop, which allows you to get the memory address of the next array element. The `*pointer` notation dereferences the pointer and returns the stored integer value.

The third part is the following:

```
fmt.Println()
pointer = (*int)(unsafe.Pointer(memoryAddress))
fmt.Print("One more: ", *pointer, " ")
memoryAddress = uintptr(unsafe.Pointer(pointer)) +
unsafe.Sizeof(array[0])
fmt.Println()
}
```

In the last part, we are trying to access an element of the array that does not exist using pointers and memory addresses. The Go compiler cannot catch such a logical error due to the use of the `unsafe` package and therefore will return something inaccurate.

Executing `moreUnsafe.go` will create the next output:

```
$ go run moreUnsafe.go
0 1 -2 3 4
One more: 824634208008
```

You have just accessed all the elements of a Go array using pointers. However, the real problem here is that when you tried to access an invalid array element, the program did not complain and returned a random number instead.

Calling C code from Go

Although Go intends to make your programming experience better and save you from the quirks of C, C remains a very capable programming language that is still useful. This means that there are situations, such as when using a database or a device driver written in C, that still require the use of C, which means that you will need to work with C code in your Go projects.



If you find yourself using this capability many times in the same project, you might need to reconsider your approach or your choice of programming language.

Calling C code from Go using the same file

The simplest way to call C code from a Go program is to include the C code in your Go source file. This needs special treatment but it is pretty fast and not that difficult.

The name of the Go source file that contains both C and Go code is `cGo.go` and will be presented in three parts.

The first part is next:

```
package main

// #include <stdio.h>
// void callC() {
//     printf("Calling C code!\n");
// }
import "C"
```



As you can see, the C code is included in the comments of the Go program. However, the `go` tool knows what to do with these kind of comments because of the use of the `C` Go package.

The second part of the program has the next Go code:

```
import "fmt"

func main() {
```

So, all the other packages should be imported separately.

The last part of `cGo.go` contains the next code:

```
    fmt.Println("A Go statement!")
    C.callC()
    fmt.Println("Another Go statement!")
}
```

In order to execute the `callC()` C function, you will need to call it as `C.callC()`.

Executing `cGo.go` will create the next output:

```
$ go run cGo.go
A Go statement!
Calling C code!
Another Go statement!
```

Calling C code from Go using separate files

Now let us continue with how to call C code from a Go program when the C code is located in a separate file.

First, I will explain the imaginary problem that we will solve with our program. We will need to use two C functions that we have implemented in the past and that we do not want or cannot rewrite in Go.

The C code

This subsection will present you with the C code for the example. It comes in two files: `callC.h` and `callC.c`. The included file (`callC.h`) contains the next code:

```
#ifndef CALLC_H
#define CALLC_H

void cHello();
void printMessage(char* message);

#endif
```

The C source file (`callC.c`) contains the next C code:

```
#include <stdio.h>
#include "callC.h"

void cHello() {
    printf("Hello from C!\n");
}

void printMessage(char* message) {
    printf("Go send me %s\n", message);
}
```

Both the `callC.c` and `callC.h` files are stored in a separate directory, which in this case is going to be `callClib`. However, you can use any directory name you want.



The actual C code is not important as long as you call the right C functions with the correct type and number of parameters. There is nothing in the C code that tells us that it is going to be used from a Go program. You should look at the Go code for the juicy part.

The Go code

This subsection will present you with the Go source code for the example, which will be named `callC.go` and will be presented to you in three parts.

The first part of `callC.go` comes with the next Go code:

```
package main

// #cgo CFLAGS: -I${SRCDIR}/callClib
// #cgo LDFLAGS: ${SRCDIR}/callC.a
// #include <stdlib.h>
// #include <callC.h>
import "C"
```

The single most important Go statement of the entire Go source file is the inclusion of the `C` package using a separate `import` statement. However, `C` is a virtual Go package that just tells `go build` to preprocess its input file using the `cgo` tool before the Go compiler processes the file. You can still see that you need to use comments to inform the Go program about the C code. In this case, you tell `callC.go` where to find the `callC.h` file as well as where to find the `callC.a` library file that we will create in a while. Such lines begin with `#cgo`.

The second part of the program is the following:

```
import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println("Going to call a C function!")
    C.cHello()
}
```

The last part of `callC.go` is next:

```
    fmt.Println("Going to call another C function!")
    myMessage := C.CString("This is Mihalis!")
    defer C.free(unsafe.Pointer(myMessage))
    C.printMessage(myMessage)

    fmt.Println("All perfectly done!")
}
```

In order to pass a string to a C function from Go, you will need to create a C string using `C.CString()`. Additionally, you will need a `defer` statement in order to free the memory space of the C string when it is no longer needed. The `defer` statement includes a call to `C.free()` and another one to `unsafe.Pointer()`.

In the next section, you will see how to compile and execute `callC.go`.

Mixing Go and C code

Now that you have the C code and the Go code, it is time to learn what to do next in order to execute the Go file that calls the C code.

The good news is that you do not need to do anything particularly difficult because all the critical information is contained in the Go file. The only critical thing that you will need to do is compile the C code in order to create a library, which requires the execution of the following commands:

```
$ ls -l callClib/
total 16
-rw-r--r--@ 1 mtsouk  staff  162 Jan 10 09:17 callC.c
-rw-r--r--@ 1 mtsouk  staff   89 Jan 10 09:17 callC.h
$ gcc -c callClib/*.c
$ ls -l callC.o
-rw-r--r--  1 mtsouk  staff  952 Jan 22 22:03 callC.o
$ file callC.o
callC.o: Mach-O 64-bit object x86_64
$ /usr/bin/ar rs callC.a *.o
ar: creating archive callC.a
$ ls -l callC.a
-rw-r--r--  1 mtsouk  staff  4024 Jan 22 22:03 callC.a
$ file callC.a
callC.a: current ar archive
$ rm callC.o
```

After that, you are going to have a file named `callC.a` located in the same directory as the `callC.go` file. The `gcc` executable is the name of the **C compiler**.

Now you are ready to compile the file with the Go code and create a new executable file:

```
$ go build callC.go
$ ls -l callC
-rwxr-xr-x  1 mtsouk  staff  2403184 Jan 22 22:10 callC
$ file callC
callC: Mach-O 64-bit executable x86_64
```

Executing the `callC` executable file will create the next output:

```
$ ./callC
Going to call a C function!
Hello from C!
Going to call another C function!
Go send me This is Mihalis!
All perfectly done!
```




If you are going to call a small amount of C code, then using a single Go file for both C and Go code is highly recommended because of its simplicity. However, if you are going to do something more complex and advanced, creating a static C library should be the preferred way.

Calling Go functions from C code

It is also possible to call a Go function from your C code. Therefore, this section will present you with a small example where two Go functions are going to be called from a C program. The Go package is going to be converted into a **C shared library** that is going to be used in the C program.

The Go package

This subsection will present you with the code of the Go package that will be used in a C program. The name of the Go package needs to be `main` but its filename can be anything you want; in this case, the filename will be `usedByC.go` and it will be presented in three parts.



You will learn more about Go packages in [Chapter 6, What You Might Not Know About Go Packages and Go Functions](#).

The first part of the code of the Go package is next:

```
package main

import "C"

import (
    "fmt"
)
```

As I said before, it is mandatory to name the Go package `main`. You will also need to import the `C` package in your Go code.

The second part comes with the following Go code:

```
//export PrintMessage
func PrintMessage() {
    fmt.Println("A Go function!")
}
```

```
}
```

Each Go function that will be called by the C code needs to be exported first. This means that you should put a comment line starting with `//export` before its implementation. After `//export`, you will need to put the name of the function because this is what the C code will use.

The last part of `usedByC.go` is next:

```
//export Multiply
func Multiply(a, b int) int {
    return a * b
}

func main() {
}
```

The `main()` function of `usedByC.go` needs no code because it is not going to be exported and therefore used by the C program. Additionally, as you also want to export the `Multiply()` function, you will need to put `//export Multiply` before its implementation.

After that, you will need to generate a C shared library from the Go code by executing the following command:

```
$ go build -o usedByC.o -buildmode=c-shared usedByC.go
```

The preceding command will generate two files named `usedByC.h` and `usedByC.o`:

```
$ ls -l usedByC.*
-rw-r--r--@ 1 mtsouk  staff      204 Jan 10 09:17 usedByC.go
-rw-r--r--  1 mtsouk  staff    1365 Jan 22 22:14 usedByC.h
-rw-r--r--  1 mtsouk  staff  2329472 Jan 22 22:14 usedByC.o
$ file usedByC.o
usedByC.o: Mach-O 64-bit dynamically linked shared library x86_64
```

You should not make any changes to `usedByC.h`.

The C code

The relevant C code can be found in the `willUseGo.c` source file, which will be presented in two parts. The first part of `willUseGo.c` is next:

```
#include <stdio.h>
#include "usedByC.h"
```

```
int main(int argc, char **argv) {
    GoInt x = 12;
    GoInt y = 23;

    printf("About to call a Go function!\n");
    PrintMessage();
}
```

If you already know C, you will understand why you need to include `usedByC.h`; this is the way the C code knows about the available functions of a library.

The second part of the C program is next:

```
GoInt p = Multiply(x,y);
printf("Product: %d\n", (int)p);
printf("It worked!\n");
return 0;
}
```

The `GoInt p` variable is needed for getting an integer value from a Go function, which is converted to a C integer using the `(int) p` notation.

Compiling and executing `willUseGo.c` on a macOS Mojave machine will create the next output:

```
$ gcc -o willUseGo willUseGo.c ./usedByC.o
$ ./willUseGo
About to call a Go function!
A Go function!
Product: 276
It worked!
```

The defer keyword

The `defer` keyword postpones the execution of a function until the surrounding function returns, which is widely used in file input and output operations because it saves you from having to remember when to close an opened file. The `defer` keyword allows you to put the function call that closes an opened file near to the function call that opened it. As you will learn about the use of `defer` in file-related operations in Chapter 8, *Telling a UNIX System What to Do*, this section will present two different usages of `defer`. You will also see `defer` in action in the section that talks about the `panic()` and `recover()` built-in Go functions, as well as in the section that is related to logging.

It is very important to remember that **deferred functions** are executed in **last in, first out (LIFO)** order after the return of the surrounding function. Putting it simply, this means that if you `defer` function `f1()` first, function `f2()` second, and function `f3()` third in the same surrounding function, when the surrounding function is about to return, function `f3()` will be executed first, function `f2()` will be executed second, and function `f1()` will be the last one to get executed.

As this definition of `defer` is a little unclear, I think that you will understand the use of `defer` a little better by looking at the Go code and the output of the `defer.go` program, which will be presented in three parts.

The first part of the program is next:

```
package main

import (
    "fmt"
)

func d1() {
    for i := 3; i > 0; i-- {
        defer fmt.Print(i, " ")
    }
}
```

Apart from the `import` block, the preceding Go code implements a function named `d1()` with a `for` loop and a `defer` statement that will be executed three times.

The second part of `defer.go` comes with the next Go code:

```
func d2() {
    for i := 3; i > 0; i-- {
        defer func() {
            fmt.Print(i, " ")
        }()
    }
    fmt.Println()
}
```

In this part of the code, you can see the implementation of another function, which is named `d2()`. The `d2()` function also contains a `for` loop and a `defer` statement, which will also be executed three times. However, this time, the `defer` keyword is applied to an **anonymous function** instead of a single `fmt.Print()` statement. Additionally, the anonymous function takes no parameters.

The last part has the following Go code:

```
func d3() {
    for i := 3; i > 0; i-- {
        defer func(n int) {
            fmt.Print(n, " ")
        }(i)
    }
}

func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

Apart from the `main()` function that calls the `d1()`, `d2()`, and `d3()` functions, you can also see the implementation of the `d3()` function, which has a `for` loop that uses the `defer` keyword on an anonymous function. However, this time, the anonymous function requires one integer parameter named `n`. The Go code tells us that the `n` parameter takes its value from the `i` variable used in the `for` loop.

Executing `defer.go` will create the next output:

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

You will most likely find the generated output complicated and challenging to understand, which proves that the operation and the results of the use of `defer` can be tricky if your code is not clear and unambiguous. Let me explain the results in order to get a better idea of how tricky `defer` can be if you do not pay close attention to your code.

We will start with the first line of the output (`1 2 3`) that is generated by the `d1()` function. The values of `i` in `d1()` are 3, 2, and 1 in that order. The function that is deferred in `d1()` is the `fmt.Print()` statement; as a result, when the `d1()` function is about to return, you get the three values of the `i` variable of the `for` loop in reverse order because deferred functions are executed in LIFO order.

Now, let me explain the second line of the output that is produced by the `d2()` function. It is really strange that we got three zeros instead of `1 2 3` in the output; however, the reason for that is relatively simple.

After the `for` loop has ended, the value of `i` is 0, because it is that value of `i` that made the `for` loop terminate. However, the tricky point here is that the deferred anonymous function is evaluated after the `for` loop ends because it has no parameters, which means that it is evaluated three times for an `i` value of 0, hence the generated output. This kind of confusing code is what might lead to the creation of nasty bugs in your projects, so try to avoid it.

Finally, we will talk about the third line of the output, which is generated by the `d3()` function. Due to the parameter of the anonymous function, each time the anonymous function is deferred, it gets and therefore uses the current value of `i`. As a result, each execution of the anonymous function has a different value to process, hence the generated output.

After that, it should be clear that the best approach to the use of `defer` is the third one, which is exhibited in the `d3()` function, because you intentionally pass the desired variable in the anonymous function in an easy-to-understand way.

Using defer for logging

This section will present an application of `defer` related to logging. The purpose of this technique is to help you organize the logging information of a function in a better way and make it more visible. The name of the Go program that will illustrate the use of the `defer` keyword in logging is `logDefer.go` and it will be presented in three parts.

The first part of `logDefer.go` is as follows:

```
package main

import (
    "fmt"
    "log"
    "os"
)

var LOGFILE = "/tmp/mGo.log"

func one(aLog *log.Logger) {
    aLog.Println("-- FUNCTION one -----")
    defer aLog.Println("-- FUNCTION one -----")

    for i := 0; i < 10; i++ {
        aLog.Println(i)
    }
}
```

```
}
```

The function named `one()` is using `defer` to make sure that the second `aLog.Println()` call is going to be executed just before the function is about to return. Therefore, all log messages from the function are going to be embedded between the opening `aLog.Println()` and the closing `aLog.Println()` calls. As a result, it will be much easier to discover the log messages of that function in your log files.

The second part of `logDefer.go` is the following:

```
func two(aLog *log.Logger) {
    aLog.Println("---- FUNCTION two")
    defer aLog.Println("FUNCTION two -----")

    for i := 10; i > 0; i-- {
        aLog.Println(i)
    }
}
```

The function named `two()` also uses `defer` to easily group its log messages. However, this time `two()` uses slightly different messages than function `one()`. It is up to you to choose the format of the logging messages.

The last part of `logDefer.go` contains the following Go code:

```
func main() {
    f, err := os.OpenFile(LOGFILE,
        os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()

    iLog := log.New(f, "logDefer ", log.LstdFlags)
    iLog.Println("Hello there!")
    iLog.Println("Another log entry!")

    one(iLog)
    two(iLog)
}
```

Executing `logDefer.go` will generate no visible output. However, looking at the contents of `/tmp/mGo.log`, which is the log file that is being used by the program, will make it pretty clear how handy the use of `defer` is in this case:

```
$ cat /tmp/mGo.log
```

```
logDefer 2019/01/19 21:15:11 Hello there!
logDefer 2019/01/19 21:15:11 Another log entry!
logDefer 2019/01/19 21:15:11 -- FUNCTION one -----
logDefer 2019/01/19 21:15:11 0
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 -- FUNCTION one -----
logDefer 2019/01/19 21:15:11 ---- FUNCTION two
logDefer 2019/01/19 21:15:11 10
logDefer 2019/01/19 21:15:11 9
logDefer 2019/01/19 21:15:11 8
logDefer 2019/01/19 21:15:11 7
logDefer 2019/01/19 21:15:11 6
logDefer 2019/01/19 21:15:11 5
logDefer 2019/01/19 21:15:11 4
logDefer 2019/01/19 21:15:11 3
logDefer 2019/01/19 21:15:11 2
logDefer 2019/01/19 21:15:11 1
logDefer 2019/01/19 21:15:11 FUNCTION two -----
```

Panic and recover

This section will present you with a tricky technique that was first mentioned in the previous chapter. This technique involves the use of the `panic()` and `recover()` functions and will be presented in `panicRecover.go`, which you will see in three parts.

Strictly speaking, `panic()` is a built-in Go function that terminates the current flow of a Go program and starts panicking. On the other hand, the `recover()` function, which is also a built-in Go function, allows you to take back control of a `goroutine` that just panicked using `panic()`.

The first part of the program is next:

```
package main

import (
    "fmt"
)
```



```
func a() {
    fmt.Println("Inside a()")
    defer func() {
        if c := recover(); c != nil {
            fmt.Println("Recover inside a()!")
        }
    }()
    fmt.Println("About to call b()")
    b()
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
}
```

Apart from the `import` block, this part includes the implementation of the `a()` function. The most important part of function `a()` is the `defer` block of code, which implements an anonymous function that will be called when there is a call to `panic()`.

The second code segment of `panicRecover.go` is next:

```
func b() {
    fmt.Println("Inside b()")
    panic("Panic in b()!")
    fmt.Println("Exiting b()")
}
```

The last part of the program that illustrates the `panic()` and `recover()` functions is the following:

```
func main() {
    a()
    fmt.Println("main() ended!")
}
```

Executing `panicRecover.go` will create the next output:

```
$ go run panicRecover.go
Inside a()
About to call b()
Inside b()
Recover inside a()!
main() ended!
```

What just happened was really impressive. However, as you can see from the output, the `a()` function did not end normally because its last two statements did not get executed:

```
fmt.Println("b() exited!")
fmt.Println("Exiting a()")
```

Nevertheless, the good thing is that `panicRecover.go` ended according to our will without panicking because the anonymous function used in `defer` took control of the situation. Also note that function `b()` knows nothing about function `a()`; however, function `a()` contains Go code that handles the panic condition of function `b()`.

Using the panic function on its own

You can also use the `panic()` function on its own without any attempt to **recover** and this subsection will show its results using the Go code of `justPanic.go`, which will be presented in two parts.

The first part of `justPanic.go` is next:

```
package main

import (
    "fmt"
    "os"
)
```

As you can see, the use of `panic()` does not require any extra Go packages.

The second part of `justPanic.go` comes with the next Go code:

```
func main() {
    if len(os.Args) == 1 {
        panic("Not enough arguments!")
    }

    fmt.Println("Thanks for the argument(s)!")
}
```

If your Go program does not have at least one command-line argument, it will call the `panic()` function. The `panic()` function takes one parameter, which is the error message that you want to print on the screen.

Executing `justPanic.go` on a macOS Mojave machine will create the next output:

```
$ go run justPanic.go
panic: Not enough arguments!
goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch2/code/justPanic.go:10 +0x91
exit status 2
```

Therefore, using the `panic()` function on its own will terminate the Go program without giving you the opportunity to recover. So, the use of the `panic()` and `recover()` pair is much more practical and professional than just using `panic()`.



The output of the `panic()` function looks like the output of the `Panic()` function from the `log` package. However, the `panic()` function sends nothing to the logging service of your UNIX machine.

Two handy UNIX utilities

There are times when a UNIX program fails for some unknown reason or does not perform well and you want to find out why without having to rewrite your code and add a plethora of debugging statements.

This section will present two command-line utilities that allow you to see the C system calls executed by an executable file. The names of the two tools are `strace(1)` and `dtrace(1)` and they allow you to inspect the operation of a program.



Please remember that at the end of the day, all programs that work on UNIX machines end up using C system calls to communicate with the UNIX kernel and perform most of their tasks.

Although both tools can work with the `go run` command, you will get less unrelated output if you first create an executable file using `go build` and use this file. This mainly occurs because, as you already know, `go run` makes various temporary files before actually running your Go code and both tools will see that and try to display information about the temporary files, which is not what you want.

The strace tool

The `strace(1)` command-line utility allows you to trace system calls and signals. As `strace(1)` only works on Linux machines, this section will use a Debian Linux machine to showcase `strace(1)`.

The output that `strace(1)` generates looks like the following:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 15 vars */]) = 0
```

```
brk(0) = 0x186c000
fstat(3, {st_mode=S_IFREG|0644, st_size=35288, ...}) = 0
```

The `strace(1)` output displays each system call with its parameters as well as its return value. Please note that in the UNIX world, a return value of 0 is a good thing.

In order to process a binary file, you will need to put the `strace(1)` command in front of the executable you want to process. However, you will need to interpret the output on your own in order to make useful conclusions from it. The good thing is that tools like `grep(1)` can get you the output that you are really looking for:

```
$ strace find /usr 2>&1 | grep ioctl
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59c50) = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or
TCGETS, 0x7ffe3bc59be0) = -1 ENOTTY (Inappropriate ioctl for device)
```

The `strace(1)` tool can print count time, calls, and errors for each system call when used with the `-c` command-line option:

```
$ strace -c find /usr 1>/dev/null
```

% time	seconds	usecs/call	calls	errors	syscall
82.88	0.063223	2	39228		getdents
16.60	0.012664	1	19587		newfstatat
0.16	0.000119	0	19618	13	open

As the normal program output is printed in standard output and the output of `strace(1)` is printed in standard error, the previous command discards the output of the command that is examined and shows the output of `strace(1)`. As you can see from the last line of the output, the `open(2)` system call was called 19,618 times, generated 13 errors, and took about 0.16% of the execution time of the entire command or about 0.000119 seconds.

The dtrace tool

Although debugging utilities such as `strace(1)` and `truss(1)` can trace system calls produced by a process, they can be slow and therefore not appropriate for solving performance problems on busy UNIX systems. Another tool, named **DTrace**, allows you to see what happens behind the scenes on a system-wide basis without the need to modify or recompile anything. It also allows you to work on production systems and watch running programs or server processes dynamically without introducing a big overhead.



Although there is a version of `dtrace(1)` that works on Linux, the `dtrace(1)` tool works best on macOS and the other FreeBSD variants.

This subsection will use the `dtruss(1)` command-line utility that comes with macOS, which is just a `dtrace(1)` script that shows the system calls of a process and saves us from having to write `dtrace(1)` code. Note that both `dtrace(1)` and `dtruss(1)` need root privileges to run.

The output that `dtruss(1)` generates looks like the following:

```
$ sudo dtruss godoc
ioctl(0x3, 0x80086804, 0x7FFEEFBFEC20)      = 0 0
close(0x3)                               = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0)      = -1 Err#2
thread_selfid(0x0, 0x0, 0x0)             = 1895378 0
geteuid(0x0, 0x0, 0x0)                   = 0 0
getegid(0x0, 0x0, 0x0)                   = 0 0
```

So, `dtruss(1)` works the same way as the `strace(1)` utility. Analogous to `strace(1)`, `dtruss(1)` will print system call counts when used with the `-c` parameter:

```
$ sudo dtruss -c go run unsafe.go 2>&1
```

CALL	COUNT
access	1
bsdthread_register	1
getuid	1
ioctl	1
issetugid	1
kqueue	1
write	1
mkdir	2
read	244
kevent	474
fcntl	479
lstat64	553
psynch_cvsignal	649
psynch_cvwait	654

The preceding output will quickly inform you about potential bottlenecks in your Go code or allow you to compare the performance of two different command-line programs.



Utilities like `strace(1)`, `dtrace(1)`, and `dtruss(1)` need some getting used to, but such tools can make our lives so much easier and more comfortable that I strongly suggest you start learning at least one such tool right now.

You can learn more about the `dtrace(1)` utility by reading *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* by Brendan Gregg and Jim Mauro or by visiting <http://dtrace.org/>.

Please have in mind that `dtrace(1)` is much more powerful than `strace(1)` because it has its own programming language. However, `strace(1)` is more versatile when all you want to do is watch the system calls of an executable file.

Your Go environment

This section will talk about finding out information about your current Go environment using the functions and the properties of the `runtime` package. The name of the program that will be developed in this section is `goEnv.go` and it will be presented in two parts.

The first part of `goEnv.go` is next:

```
package main

import (
    "fmt"
    "runtime"
)
```

As you will see in a while, the `runtime` package contains functions and properties that will reveal the desired information. The second code portion of `goEnv.go` contains the implementation of the `main()` function:

```
func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("Using Go version", runtime.Version())
    fmt.Println("Number of CPUs:", runtime.NumCPU())
    fmt.Println("Number of Goroutines:", runtime.NumGoroutine())
}
```

Executing `goEnv.go` on a macOS Mojave machine with Go version 1.11.4 will create the next output:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.11.4
Number of CPUs: 8
Number of Goroutines: 1
```

The same program generates the next output on a Debian Linux machine with Go version 1.3.3:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.3.3
Number of CPUs: 1
Number of Goroutines: 4
```

However, the real benefit you can get from being able to find information about your Go environment is illustrated in the next program, named `requiredVersion.go`, which tells you if you are using Go version 1.8 or higher:

```
package main

import (
    "fmt"
    "runtime"
    "strconv"
    "strings"
)

func main() {
    myVersion := runtime.Version()
    major := strings.Split(myVersion, ".")[0][2]
    minor := strings.Split(myVersion, ".")[1]
    m1, _ := strconv.Atoi(string(major))
    m2, _ := strconv.Atoi(minor)

    if m1 == 1 && m2 < 8 {
        fmt.Println("Need Go version 1.8 or higher!")
        return
    }

    fmt.Println("You are using Go version 1.8 or higher!")
}
```

The `strings` Go standard package is used for splitting the Go version string you get from `runtime.Version()` in order to get its first two parts, whereas the `strconv.Atoi()` function is used for converting a string to an integer.

Executing `requiredVersion.go` on the macOS Mojave machine will create the next output:

```
$ go run requiredVersion.go
You are using Go version 1.8 or higher!
```

However, if you run `requiredVersion.go` on the Debian Linux machine you saw earlier in this section, it will generate the next output:

```
$ go run requiredVersion.go
Need Go version 1.8 or higher!
```

So, by using the Go code of `requiredVersion.go`, you will be able to identify whether your UNIX machine has the required Go version or not.

The go env command

If you need to get a list of all environment variables supported by Go and the Go compiler, along with their current values, then the solution is to execute `go env`.

On my macOS Mojave, which uses Go version 1.11.4, the pretty rich output of `go env` is as follows:

```
$ go env
GOARCH="amd64"
GOBIN=""
GOCACHE="/Users/mtsouk/Library/Caches/go-build"
GOEXE=""
GOFLAGS=""
GOHOSTARCH="amd64"
GOHOSTOS="darwin"
GOOS="darwin"
GOPATH="/Users/mtsouk/go"
GOPROXY=""
GORACE=""
GOROOT="/usr/local/Cellar/go/1.11.4/libexec"
GOTMPDIR=""
GOTOOLDIR="/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64"
GCCGO="gccgo"
CC="clang"
CXX="clang++"
```



```
CGO_ENABLED="1"
GOMOD=""
CGO_CFLAGS="-g -O2"
CGO_CPPFLAGS=""
CGO_CXXFLAGS="-g -O2"
CGO_FFLAGS="-g -O2"
CGO_LDFLAGS="-g -O2"
PKG_CONFIG="pkg-config"
GOGCCFLAGS="-fPIC -m64 -pthread -fno-caret-diagnostics -Qunused-
arguments -fmessage-length=0 -fdebug-prefix-
map=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build790367620=/tmp/go-build -gno-record-gcc-switches -fno-common"
```

Please note that some of these environment variables might change if you are using a different Go version, if your username is not `mtsouk`, if you are using a different UNIX variant on a different hardware, or if you are using Go modules (`GOMOD`) by default.

The Go assembler

This section will briefly talk about the assembly language and the **Go assembler**, which is a Go tool that allows you to see the assembly language used by the Go compiler.

As an example, you can see the assembly language of the `goEnv.go` program you saw in the previous section of this chapter by executing the next command:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S goEnv.go
```

The value of the `GOOS` variable defines the name of the target operating system whereas the value of the `GOARCH` variable defines the compilation architecture. The preceding command was executed on a macOS Mojave machine, hence the use of the `darwin` value for the `GOOS` variable.

The output of the previous command is pretty large even for a small program such as `goEnv.go`. Some of its output is next:

```
"".main TEXT size=859 args=0x0 locals=0x118
0x0000 00000 (goEnv.go:8)      TEXT    "".main(SB), $280-0
0x00be 00190 (goEnv.go:9)      PCDATA  $0, $1
0x0308 00776 (goEnv.go:13)     PCDATA  $0, $5
0x0308 00776 (goEnv.go:13)     CALL    runtime.convT2E64(SB)
"".init TEXT size=96 args=0x0 locals=0x8
0x0000 00000 (<autogenerated>:1) TEXT    "".init(SB), $8-0
0x0000 00000 (<autogenerated>:1) MOVQ    (TLS), CX
0x001d 00029 (<autogenerated>:1) FUNCDATA $0,
```

```
gclocals d4dc2f11db048877dbc0f60a22b4adb3 (SB)
0x001d 00029 (<autogenerated>:1) FUNCDATA      $1,
gclocals 33cdeccccebe80329f1fdbee7f5874cb (SB)
```

The lines that contain the `FUNCDATA` and `PCDATA` directives are read and used by the Go garbage collector and are automatically generated by the Go compiler.

An equivalent variant of the preceding command is next:

```
$ GOOS=darwin GOARCH=amd64 go build -gcflags -S goEnv.go
```

The list of valid `GOOS` values includes `android`, `darwin`, `dragonfly`, `freebsd`, `linux`, `nacl`, `netbsd`, `openbsd`, `plan9`, `solaris`, `windows`, and `zos`. On the other hand, the list of valid `GOARCH` values includes `386`, `amd64`, `amd64p32`, `arm`, `armbe`, `arm64`, `arm64be`, `ppc64`, `ppc64le`, `mips`, `mipsle`, `mips64`, `mips64le`, `mips64p32`, `mips64p32le`, `ppc`, `s390`, `s390x`, `sparc`, and `sparc64`.



If you are really interested in the Go assembler and you want more information, you should visit <https://golang.org/doc/asm>.

Node trees

A Go node is a `struct` with a large number of properties. You will learn more about defining and using Go **structures** in Chapter 4, *The Uses of Composite Types*. Everything in a Go program is parsed and analyzed by the modules of the Go compiler according to the grammar of the Go programming language. The final product of this analysis is a **tree** that is specific to the provided Go code and represents the program in a different way that is suited to the compiler rather than to the developer.



Please note that `go tool 6g -W test.go` does not work on newer Go versions. You should use `go tool compile -W test.go` instead.

This section will first use the following Go code, which is saved as `nodeTree.go`, as an example in order to see the kind of low-level information the `go tool` can provide us with:

```
package main

import (
    "fmt"
```

```

)

func main() {
    fmt.Println("Hello there!")
}

```

The Go code of `nodeTree.go` is pretty easy to understand, so you will not be surprised by its output, which is next:

```

$ go run nodeTree.go
Hello there!

```

Now it is time to see some internal Go workings by executing the next command:

```

$ go tool compile -W nodeTree.go
before walk main
.   CALLFUNC 1(8) tc(1) STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) 1(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG 1(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   CONVIFACE 1(8) esc(h) tc(1) implicit(true) INTER-interface {}
.   .   .   NAME-main.statictmp_0 a(true) 1(8) x(0) class(PEXTERN) tc(1)
used string
.   VARKILL 1(8) tc(1)
.   .   NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N) used
ARRAY-[1]interface {}
after walk main
.   CALLFUNC-init
.   .   AS 1(8) tc(1)
.   .   .   NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
.   .   AS 1(8) tc(1)
.   .   .   NAME-main..autotmp_2 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used PTR64-*[1]interface {}
.   .   .   ADDR 1(8) tc(1) PTR64-*[1]interface {}
.   .   .   .   NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
.   .   BLOCK 1(8)
.   .   BLOCK-list
.   .   .   AS 1(8) tc(1) hascall
.   .   .   .   INDEX 1(8) tc(1) assigned bounded hascall INTER-interface
{}
.   .   .   .   .   IND 1(8) tc(1) implicit(true) assigned hascall ARRAY-
[1]interface {}
.   .   .   .   .   .   NAME-main..autotmp_2 a(true) 1(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
.   .   .   .   .   .   LITERAL-0 1(8) tc(1) int

```

```

. . . . EFACE 1(8) tc(1) INTER-interface {}
. . . . . ADDR a(true) 1(8) tc(1) PTR64-*uint8
. . . . . NAME-type.string a(true) x(0) class(PEXTERN) tc(1)
uint8
. . . . . ADDR 1(8) tc(1) PTR64-*string
. . . . . NAME-main.statictmp_0 a(true) 1(8) x(0)
class(PEXTERN) tc(1) addrtaken used string
. . . BLOCK 1(8)
. . . BLOCK-list
. . . . AS 1(8) tc(1) hascall
. . . . . NAME-main..autotmp_1 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
. . . . . SLICEARR 1(8) tc(1) hascall SLICE-[]interface {}
. . . . . NAME-main..autotmp_2 a(true) 1(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
. . . CALLFUNC 1(8) tc(1) hascall STRUCT-(int, error)
. . . . NAME-fmt.Println a(true) 1(263) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
. . . . . DDDARG 1(8) esc(no) PTR64-*[1]interface {}
. . . CALLFUNC-list
. . . . AS 1(8) tc(1)
. . . . . INDREGSP-SP a(true) 1(8) x(0) tc(1) addrtaken main.__ SLICE-
[]interface {}
. . . . . NAME-main..autotmp_1 a(true) 1(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
. . . . VARKILL 1(8) tc(1)
. . . . . NAME-main..autotmp_0 a(true) 1(8) x(0) class(PAUTO) esc(N) tc(1)
addrtaken assigned used ARRAY-[1]interface {}
before walk init
. . . IF 1(1) tc(1)
. . . . GT 1(1) tc(1) bool
. . . . . NAME-main.initdone a(true) 1(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . . . LITERAL-1 1(1) tc(1) uint8
. . . IF-body
. . . . RETURN 1(1) tc(1)
. . . IF 1(1) tc(1)
. . . . EQ 1(1) tc(1) bool
. . . . . NAME-main.initdone a(true) 1(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . . . LITERAL-1 1(1) tc(1) uint8
. . . IF-body
. . . . CALLFUNC 1(1) tc(1)
. . . . . NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
. . . . AS 1(1) tc(1)
. . . . . NAME-main.initdone a(true) 1(1) x(0) class(PEXTERN) tc(1) assigned
used uint8

```

```

. . LITERAL-1 l(1) tc(1) uint8
. CALLFUNC l(1) tc(1)
. . NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
. . LITERAL-2 l(1) tc(1) uint8
. RETURN l(1) tc(1)
after walk init
. IF l(1) tc(1)
. . GT l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . RETURN l(1) tc(1)
. IF l(1) tc(1)
. . EQ l(1) tc(1) bool
. . . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
. . . LITERAL-1 l(1) tc(1) uint8
. IF-body
. . CALLFUNC l(1) tc(1) hascall
. . . NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
. . LITERAL-1 l(1) tc(1) uint8
. CALLFUNC l(1) tc(1) hascall
. . NAME-fmt.init a(true) l(1) x(0) class(PFUNC) tc(1) used FUNC-func()
. AS l(1) tc(1)
. . NAME-main.initdone a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
. . LITERAL-2 l(1) tc(1) uint8
. RETURN l(1) tc(1)

```

As you can understand, the Go compiler and its tools do many things behind the scenes, even for a small program such as `nodeTree.go`.



The `-w` parameter tells the `go tool compile` command to print the **debug parse tree** after the type checking.

Look at the output of the next two commands:

```
$ go tool compile -W nodeTree.go | grep before
before walk main
before walk init
$ go tool compile -W nodeTree.go | grep after
after walk main
after walk init
```

As you can see, the `before` keyword is about the beginning of the execution of a function. If your program had more functions, you would have got more output, which is illustrated in the next example:

```
$ go tool compile -W defer.go | grep before
before d1
before d2
before d3
before main
before d2.func1
before d3.func1
before init
before type..hash.[2]interface {}
before type..eq.[2]interface {}
```

The previous example uses the Go code of `defer.go`, which is much more complicated than `nodeTree.go`. However, it should be obvious that the `init()` function is automatically generated by Go as it exists in both outputs of `go tool compile -W (nodeTree.go and defer.go)`. I will now present you with a juicier version of `nodeTree.go`, named `nodeTreeMore.go`:

```
package main

import (
    "fmt"
)

func functionOne(x int) {
    fmt.Println(x)
}

func main() {
    varOne := 1
    varTwo := 2
    fmt.Println("Hello there!")
    functionOne(varOne)
    functionOne(varTwo)
}
```

The `nodeTreeMore.go` program has two variables, named `varOne` and `varTwo`, and an extra function named `functionOne`. Searching the output of `go tool compile -W` for `varOne`, `varTwo`, and `functionOne` will reveal the following information:

```
$ go tool compile -W nodeTreeMore.go | grep functionOne | uniq
before walk functionOne
after walk functionOne
. . . NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used
FUNC-func(int)
$ go tool compile -W nodeTreeMore.go | grep varTwo | uniq
. . . NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1) used
int
. . . . NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) tc(1)
used int
$ go tool compile -W nodeTreeMore.go | grep varOne | uniq
. . . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used
int
. . . . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1)
used int
```

So, `varOne` is represented as `NAME-main.varOne` while `varTwo` is denoted by `NAME-main.varTwo`. The `functionOne()` function is referenced as `NAME-main.functionOne`. Consequently, the `main()` function is referenced as `NAME-main`.

Now, let us see the next code of the debug parse tree of `nodeTreeMore.go`:

```
before walk functionOne
. AS l(8) tc(1)
. . NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
assigned used int
. . . NAME-main.x a(true) g(1) l(7) x(0) class(PPARAM) tc(1) used int
```

This data is related to the definition of `functionOne()`. The `l(8)` string tells us that the definition of this node can be found in line eight, that is, after reading line seven. The `NAME-main..autotmp_2` integer variable is automatically generated by the compiler.

The next part of the debug parse tree output that will be explained here is:

```
. CALLFUNC l(15) tc(1)
. . NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used
FUNC-func(int)
. CALLFUNC-list
. . NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) tc(1) used
int
```

The first line says that at line 15 of the program, which is specified by `l(15)`, you will call `NAME-main.functionOne`, which is defined at line seven of the program, as specified by `l(7)`, which is a function that requires a single integer parameter, as specified by `FUNC-func(int)`. The function list of parameters, which is specified after `CALLFUNC-list`, includes the `NAME-main.varOne` variable, which is defined at line 12 of the program, as `l(12)` shows.

Finding out more about go build

If you want to learn more about what is happening behind the scenes when you execute a `go build` command, you should add the `-x` flag to it:

```
$ go build -x defer.go
WORK=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-build254573394
mkdir -p $WORK/b001/
cat >$WORK/b001/importcfg.link << 'EOF' # internal
packagefile command-line-arguments=/Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
packagefile fmt=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/fmt.a
packagefile
runtime=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/runtime.a
packagefile
errors=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/errors.a
packagefile io=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/io.a
packagefile
math=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math.a
packagefile os=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/os.a
packagefile
reflect=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/reflect.a
packagefile
strconv=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/strconv.a
packagefile
sync=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync.a
packagefile
unicode/utf8=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode/u
tf8.a
packagefile
internal/bytealg=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/inter
nal/bytealg.a
packagefile
internal/cpu=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/
cpu.a
packagefile
runtime/internal/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd6
```



```

4/runtime/internal/atomic.a
packagefile
runtime/internal/sys=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/r
untime/internal/sys.a
packagefile
sync/atomic=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/sync/atomic.a
packagefile
internal/poll=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/poll.a
packagefile
internal/syscall/unix=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/syscall/unix.a
packagefile
internal/testlog=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/testlog.a
packagefile
syscall=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/syscall.a
packagefile
time=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/time.a
packagefile
unicode=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/unicode.a
packagefile
math/bits=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/math/bits.a
packagefile
internal/race=/usr/local/Cellar/go/1.11.4/libexec/pkg/darwin_amd64/internal/race.a
EOF
mkdir -p $WORK/b001/exe/
cd .
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/link -o
$WORK/b001/exe/a.out -importcfg $WORK/b001/importcfg.link -buildmode=exe -
buildid=nkFdi6n3HGYZXDdCOju1/VfKOjehfe3PSzik3cZom/OthUDj9rThOtZPf-2627/nkFd
i6n3HGYZXDdCOju1 -extld=clang /Users/mtsouk/Library/Caches/go-
build/9d/9d6ca8651e083f3662adf82bb90a00837fc76f55839e65c7107bb55fcab92458-d
/usr/local/Cellar/go/1.11.4/libexec/pkg/tool/darwin_amd64/buildid -w
$WORK/b001/exe/a.out # internal
mv $WORK/b001/exe/a.out defer
rm -r $WORK/b001/

```

Once again, there are many things happening in the background and it is good to be aware of them. However, most of the time, you will not have to deal with the actual commands of the compilation process.

Creating WebAssembly code

Go allows you to create WebAssembly code with the help of the `go` tool. Before I illustrate the process, I will share more information about WebAssembly.

A quick introduction to WebAssembly

WebAssembly (Wasm) is a machine model and executable format targeting a virtual machine. It is designed for efficiency, both in speed and file size. This means that you can use a WebAssembly binary on any platform you want without a single change.

WebAssembly comes in two formats: plain text format and binary format. Plain text format WebAssembly files have the `.wat` extension, whereas binary files have the `.wasm` file extension. Notice that once you have a WebAssembly binary file, you will have to load and use it using the JavaScript API.

Apart from Go, WebAssembly can also be generated from other programming languages that have support for static typing, including Rust, C, and C++.

Why is WebAssembly important?

WebAssembly is important for the following reasons:

- WebAssembly code runs at a speed that is pretty close to the native speed, which means that WebAssembly is fast.
- You can create WebAssembly code from many programming languages, which might include programming languages that you already know.
- Most modern web browsers natively support WebAssembly without the need for a plugin or any other software installation.
- WebAssembly code is much faster than **JavaScript** code.

Go and WebAssembly

For Go, WebAssembly is just another architecture. Therefore, you can use the cross-compilation capabilities of Go in order to create WebAssembly code.

You will learn more about the cross-compilation capabilities of Go in [Chapter 11, Code Testing, Optimization, and Profiling](#). For now, notice the values of the `GOOS` and `GOARCH` environment variables that are being used while compiling Go code into WebAssembly because this is where all the magic happens.

An example

In this section, we are going to see how a Go program can be compiled into WebAssembly code. The Go code of `toWasm.go` is the following:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Creating WebAssembly code from Go!")
}
```

The important things to notice here are that there is no sign of WebAssembly in this code and that `toWasm.go` can be compiled and executed on its own, which means that it has no external dependencies related to WebAssembly.

The last step that you will need to take in order to create the WebAssembly code is executing the following command:

```
$ GOOS=js GOARCH=wasm go build -o main.wasm toWasm.go
$ ls -l
total 4760
-rwxr-xr-x  1 mtsouk  staff  2430633 Jan 19 21:00 main.wasm
-rw-r--r--@ 1 mtsouk  staff      100 Jan 19 20:53 toWasm.go
$ file main.wasm
main.wasm: , created: Thu Oct 25 20:41:08 2007, modified: Fri May 28
13:51:43 2032
```

So, the values of `GOOS` and `GOARCH` found in the first command tell Go to create WebAssembly code. If you do not put the right `GOOS` and `GOARCH` values, the compilation will not generate WebAssembly code or it might fail.

Using the generated WebAssembly code

So far, we have only generated a WebAssembly binary file. However, there are still some steps that you will need to take in order to use that WebAssembly binary file and see its results on the window of a web browser.



If you are using **Google Chrome** as your web browser, then there is a flag that allows you to enable Liftoff, which is a compiler for WebAssembly that will theoretically improve the running time of WebAssembly code. It does not hurt to try it! In order to change that flag, you should visit `chrome://flags/#enable-wasmbaseline`.

The first step is to copy `main.wasm` into a directory of your web server. Next, you will need to execute the following command:

```
$ cp "$(go env GOROOT)/misc/wasm/wasm_exec.js" .
```

This will copy `wasm_exec.js` from the Go installation into the current directory. You should put that file in the same directory of your web server that you put `main.wasm`.

The JavaScript code found in `wasm_exec.js` will not be presented here. On the other hand, the HTML code of `index.html` will be presented:

```
<HTML>

<head>
  <meta charset="utf-8">
  <title>Go and WebAssembly</title>
</head>

<body>
  <script src="wasm_exec.js"></script>
  <script>
    if (!WebAssembly.instantiateStreaming) { // polyfill
      WebAssembly.instantiateStreaming = async (resp, importObject) => {
        const source = await (await resp).arrayBuffer();
        return await WebAssembly.instantiate(source, importObject);
      };
    }

    const go = new Go();
    let mod, inst;
    WebAssembly.instantiateStreaming(fetch("main.wasm"),
    go.importObject).then((result) => {
      mod = result.module;
      inst = result.instance;
```

```

        document.getElementById("runButton").disabled = false;
    }).catch((err) => {
        console.error(err);
    });

    async function run() {
        console.clear();
        await go.run(inst);
        inst = await WebAssembly.instantiate(mod, go.importObject);
    }
</script>

<button onClick="run();" id="runButton" disabled>Run</button>
</body>
</HTML>

```

Please note that the `Run` button created by the HTML code will not get activated until the WebAssembly code is loaded.

The next figure shows the output of the WebAssembly code as presented in the JavaScript console of the Google Chrome web browser. Other web browsers will show a similar output.

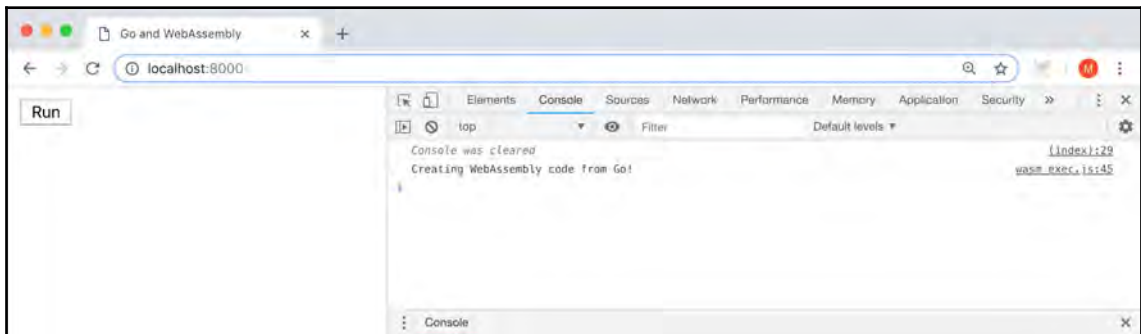


Figure 2.2: Serving WebAssembly code generated by Go



In Chapter 12, *The Foundations of Network Programming in Go*, you will learn how to develop your own web servers in Go.

However, I believe there is a much easier and simpler way to test your WebAssembly applications and that includes the use of **Node.js**. There is no need for a web server because Node.js is a **JavaScript** runtime built on Chrome's V8 JavaScript engine.

Provided that you have Node.js already installed on your local machine, you can execute the following command:

```
$ export PATH="$PATH:$(go env GOROOT)/misc/wasm"
$ GOOS=js GOARCH=wasm go run .
Creating WebAssembly code from Go!
```

The output of the second command verifies that the WebAssembly code is correct and generates the desired message. Please note that the first command is not strictly required as it just changes the current value of the `PATH` environment variable in order to include the directory where the current Go installation stores its WebAssembly-related files.

General Go coding advice

The following is a list of practical advice that will help you to write better Go code:

- If you have an error in a Go function, either log it or return it; do not do both unless you have a really good reason for doing so.
- Go interfaces define behaviors not data and data structures.
- Use the `io.Reader` and `io.Writer` interfaces when possible because they make your code more extensible.
- Make sure that you pass a pointer to a variable to a function only when needed. The rest of the time, just pass the value of the variable.
- Error variables are not `string` variables; they are `error` variables!
- Do not test your Go code on production machines unless you have a really good reason to do so.
- If you do not really know a Go feature, test it before using it for the first time, especially if you are developing an application or a utility that will be used by a large number of users.
- If you are afraid of making mistakes, you will most likely end up doing nothing really interesting. Experiment as much as you can!

Exercises and links

- Learn more about the `unsafe` standard Go package by visiting its documentation page at <https://golang.org/pkg/unsafe/>.
- Visit the web site of DTrace at <http://dtrace.org/>.

- Use `strace(1)` on your Linux machine to inspect the operation of some standard UNIX utilities such as `cp(1)` and `ls(1)`. What do you see?
- If you are using a macOS machine, use `dtruss(1)` to see how the `sync(8)` utility works.
- Write your own example where you use your own C code from a Go program.
- Write a Go function and use it in a C program.
- You can find more information about the functions of the `runtime` package by visiting <https://golang.org/pkg/runtime/>.
- Reading research papers might be difficult but it is very rewarding. Please download the *On-the-Fly Garbage Collection: An Exercise in Cooperation* paper and read it. The paper can be found in many places, including <https://dl.acm.org/citation.cfm?id=359655>.
- Visit <https://github.com/gasche/gc-latency-experiment> in order to find benchmarking code for the garbage collector of various programming languages.
- The Node.js web site can be found at <https://nodejs.org/en/>.
- You can learn more about WebAssembly at <https://webassembly.org/>.
- Should you wish to learn more about garbage collection, you should definitely visit <http://gchandbook.org/>.
- Visit the documentation page of `cgo` at <https://golang.org/cmd/cgo/>.

Summary

This chapter discussed many interesting Go topics, including theoretical and practical information about the Go garbage collector; how to call C code from your Go programs; the handy and sometimes tricky `defer` keyword; the `panic()` and `recover()` functions; the `strace(1)`, `dtrace(1)`, and `dtruss(1)` UNIX tools; the use of the `unsafe` standard Go package; how to generate WebAssembly code from Go; and assembly code generated by Go. Finally, it shared information about your Go environment using the `runtime` package and showed how to reveal and explain the node tree of a Go program, before giving you some handy Go coding advice.

What you should remember from this chapter is that tools such as the `unsafe` Go package and the ability to call C code from Go are usually used on three occasions: firstly, when you want the best performance and you want to sacrifice some Go safety for it; secondly, when you want to communicate with another programming language; and thirdly, when you want to implement something that cannot be implemented in Go.

In the next chapter, we will start learning about the basic data types that come with Go, including **arrays**, **slices**, and **maps**. Despite their simplicity, these data types are the building blocks of almost every Go application because they are the basis of more complex data structures, which allows you to store your data and move information inside your Go projects.

Additionally, you will learn about **pointers**, which can also be found in other programming languages, Go **loops**, and the unique way that Go works with dates and times.

3

Working with Basic Go Data Types

The previous chapter talked about many fascinating topics including the way the Go garbage collector works, the `panic()` and `recover()` functions, the `unsafe` package, how to call C code from a Go program, and how to call Go code from a C program, as well as the node tree created by the Go compiler when compiling a Go program.

The core subject of this chapter is the basic data types of Go. This list includes **numeric types**, **arrays**, **slices**, and **maps**. Despite their simplicity, these data types can help you to make numeric calculations. You can also store, retrieve, and alter the data of your programs in a very convenient and quick way. The chapter also covers **pointers**, **constants**, **loops**, and working with dates and times in Go.

In this chapter, you will learn about:

- Numeric data types
- Go arrays
- Go slices and why slices are much better than arrays
- How to append an array to an existing slice
- Go maps
- Pointers in Go
- Looping in Go
- Constants in Go
- Working with times
- Measuring the execution time of commands and functions
- Operating with dates

Numeric data types

Go has native support for integers and floating-point numbers, as well as complex numbers. The subsections that follow will tell you more about each numeric type supported by Go.

Integers

Go offers support for four different sizes of **signed** and **unsigned** integers, named `int8`, `int16`, `int32`, `int64`; and `uint8`, `uint16`, `uint32`, and `uint64`, respectively. The number at the end of each type shows the number of bits used for representing each type.

Additionally, `int` and `uint` exist and are the most efficient signed and unsigned integers for your current platform. Therefore, when in doubt, use `int` and `uint`, but have in mind that the size of these types changes depending on the architecture.

The difference between signed and unsigned integers is the following: if an integer has eight bits and no sign, then its values can be from binary 00000000 (0) to binary 11111111 (255). If it has a sign, then its values can be from -127 to 127. This means that you get to have seven binary digits to store your number because the eighth bit is used for keeping the sign of the integer. The same rule applies to the other sizes of unsigned integers.

Floating-point numbers

Go supports only two types of floating-point numbers: `float32` and `float64`. The first one provides about six decimal digits of precision, whereas the second one gives you 15 digits of precision.

Complex numbers

Similar to floating-point numbers, Go offers two complex number types named `complex64` and `complex128`. The first one uses two `float32`: one for the real part and the other for the imaginary part of the complex number, whereas `complex128` uses two `float64`. Complex numbers are expressed in the form of $a + bi$, where a and b are real numbers, and i is a solution of the equation $x^2 = -1$.

All these numeric data types are illustrated in the code of `numbers.go`, which will be presented in three parts.

The first part of `numbers.go` is as follows:

```
package main

import (
    "fmt"
)

func main() {
    c1 := 12 + 1i
    c2 := complex(5, 7)
    fmt.Printf("Type of c1: %T\n", c1)
    fmt.Printf("Type of c2: %T\n", c2)

    var c3 complex64 = complex64(c1 + c2)
    fmt.Println("c3:", c3)
    fmt.Printf("Type of c3: %T\n", c3)

    cZero := c3 - c3
    fmt.Println("cZero:", cZero)
```

In this part, we are working with complex numbers and making some calculations with them. There are two ways to create a complex number: directly, as with `c1` and `c2`, or indirectly by making calculations with existing complex numbers, as with `c3` and `cZero`.



If you mistakenly try to create a complex number as `aComplex := 12 + 2 * i`, there will be two possible outcomes because this statement tells Go that you want to perform an addition and a multiplication.

If there is no numeric variable named `i` in the current scope, this statement will create a syntax error and the compilation of your Go code will fail. However, if a numeric variable named `i` is already defined, the calculation will be successful, but you will not get the desired complex number as the result (**bug**).

The second part of `numbers.go` is the following:

```
x := 12
k := 5
fmt.Println(x)
fmt.Printf("Type of x: %T\n", x)

div := x / k
fmt.Println("div", div)
```

In this part, we are working with signed integers. Please note that if you divide two integers, Go thinks that you want the result of the integer division and will calculate and return the **quotient** of the integer division. So, trying to divide 11 by 2 will result in 5 in an integer division and not 5.5. If you are not familiar with mathematics, this might come as a surprise.



When you are converting a floating-point number to an integer, the fraction is discarded by truncating the floating-point number toward zero, which means that some data might get lost in the process.

The last part of `numbers.go` includes the following code:

```
var m, n float64
m = 1.223
fmt.Println("m, n:", m, n)

y := 4 / 2.3
fmt.Println("y:", y)

divFloat := float64(x) / float64(k)
fmt.Println("divFloat", divFloat)
fmt.Printf("Type of divFloat: %T\n", divFloat)
}
```

In this last part of the program, we are working with floating-point numbers. In the presented code, you can see how you can use `float64()` to tell Go to create a floating-point number when dividing two integers. If you just type `divFloat := float64(x) / k`, then you will get the following error message when running the code:

```
$ go run numbers.go
# command-line-arguments
./numbers.go:35:25: invalid operation: float64(x) / k (mismatched types
float64 and int)
```

Executing `numbers.go` will generate the following output:

```
Type of c1: complex128
Type of c2: complex128
c3: (17+8i)
Type of c3: complex64
cZero: (0+0i)
12
Type of x: int
div 2
m, n: 1.223 0
```

```
y: 1.7391304347826086
divFloat 2.4
Type of divFloat: float64
```

Number literals in Go 2

At the time of writing this, there is a proposal for changes in the way Go handles **number literals**. Number literals are related to the way you can define and use numbers in a programming language. This particular proposal is related to the representation of binary integer literals, octal integer literals, the digit separator, and support for hexadecimal floating-point numbers.

You can find more information about the proposal related to Go 2 and number literals at <https://golang.org/design/19308-number-literals>.



The people who develop Go keep a detailed release dashboard that you can view at <https://dev.golang.org/release>.

Go loops

Every programming language has a way of looping and Go is no exception. Go offers the `for` loop, which allows you to iterate over many kinds of data types.



Go does not offer support for the `while` keyword. However, `for` loops in Go can replace `while` loops.

The for loop

The `for` loop allows you to iterate a predefined number of times, for as long as a condition is valid, or according to a value that is calculated at the beginning of the `for` loop. Such values include the size of a slice or an array, or the number of keys on a map. This means that the most common way of accessing all the elements of an array, a slice, or a map is the `for` loop.

The simplest form of a `for` loop follows. A given variable takes a range of predefined values:

```
for i := 0; i < 100; i++ {  
}
```

Generally speaking, a `for` loop has three sections: the first one is called the initialization, the second one is called the condition, and the last one is called the afterthought. All sections are optional.

In the previous loop, the values that `i` will take are from 0 to 99. As soon as `i` reaches 100, the execution of the `for` loop will stop. In this case, `i` is a local and temporary variable, which means that after the termination of the `for` loop, `i` will be garbage collected at some point and disappear. However, if `i` was defined outside the `for` loop, it will keep its value after the termination of the `for` loop. In this case, the value of `i` after the termination of the `for` loop will be 100 as this was the last value of `i` in this particular program at this particular point.

You can completely exit a `for` loop using the `break` keyword. The `break` keyword also allows you to create a `for` loop without an exit condition, such as `i < 100` used in the preceding example, because the exit condition can be included in the code block of the `for` loop. You are also allowed to have multiple exit conditions in a `for` loop. Additionally, you can skip a single iteration of a `for` loop using the `continue` keyword.

The while loop

As discussed earlier, Go does not offer the `while` keyword for writing `while` loops but allows you to use a `for` loop instead of a `while` loop. This section will present two examples where a `for` loop does the job of a `while` loop.

Firstly, let's look at a typical case where you might want to write something like `while(true):`

```
for {  
}
```

It is the job of the developer to use the `break` keyword to exit this `for` loop.

However, the `for` loop can also emulate a `do...while` loop, which can be found in other programming languages.

As an example, the following Go code is equivalent to a `do...while(anExpression)` loop:

```
for ok := true; ok; ok = anExpression {  
}
```

As soon as the `ok` variable has the `false` value, the `for` loop will terminate.

There is also a `for condition {}` loop in Go where you specify the condition and the `for` loop is executed for as long as the condition is `true`.

The range keyword

Go also offers the `range` keyword, which is used in `for` loops and allows you to write easy-to-understand code for iterating over supported Go data types including Go **channels**. The main advantage of the `range` keyword is that you do not need to know the **cardinality** of a slice, a map, or a channel in order to process its elements one by one. You will see `range` in action later in the chapter.

An example with multiple Go loops

This section will display multiple examples of `for` loops. The name of the file is `loops.go` and will be presented in four parts. The first code segment of `loops.go` is next:

```
package main  
  
import (  
    "fmt"  
)  
  
func main() {  
    for i := 0; i < 100; i++ {  
        if i%20 == 0 {  
            continue  
        }  
  
        if i == 95 {  
            break  
        }  
  
        fmt.Print(i, " ")  
    }  
}
```

The preceding code shows a typical `for` loop, as well as the use of the `continue` and `break` keywords.

The next code segment is the following:

```
fmt.Println()
i := 10
for {
    if i < 0 {
        break
    }
    fmt.Print(i, " ")
    i--
}
fmt.Println()
```

The offered code emulates a typical `while` loop. Note the use of the `break` keyword to exit the `for` loop.

The third part of `loops.go` is next:

```
i = 0
anExpression := true
for ok := true; ok; ok = anExpression {
    if i > 10 {
        anExpression = false
    }

    fmt.Print(i, " ")
    i++
}
fmt.Println()
```

In this part, you see the use of a `for` loop that does the job of a `do...while` loop, as discussed earlier on in this chapter. Notice that this `for` loop is difficult to read.

The last part of `loops.go` comes with the next Go code:

```
anArray := [5]int{0, 1, -1, 2, -2}
for i, value := range anArray {
    fmt.Println("index:", i, "value: ", value)
}
}
```

Applying the `range` keyword to an array variable returns two values: an array index and the value of the element at that index, respectively.

You can use both of them, one of them, or none of them in case you just want to count the elements of the array or perform some other task the same number of times as there are items in an array.

Executing `loops.go` will produce the next output:

```
$ go run loops.go
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 81 82
83 84 85 86 87 88 89 90 91 92 93 94
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11
index: 0 value: 0
index: 1 value: 1
index: 2 value: -1
index: 3 value: 2
index: 4 value: -2
```

Go arrays

Arrays are one of the most popular data structures for two reasons. The first reason is that they are simple and easy to understand, while the second reason is that they are very versatile and can store many different kinds of data.

You can declare an array that stores four integers as follows:

```
anArray := [4]int{1, 2, 4, -4}
```

The size of the array is stated before its type, which is defined before its elements. You can find the length of an array with the help of the `len()` function: `len(anArray)`.

The index of the first element of any dimension of an array is zero; the index of the second element of any array dimension is one and so on. This means that for an array with one dimension named `a`, the valid indexes are from 0 to `len(a)-1`.

Although you might be familiar with accessing the elements of an array in other programming languages and using a `for` loop and one or more numeric variables, there exist more idiomatic ways to visit all the elements of an array in Go. They involve the use of the `range` keyword and allow you to bypass the use of the `len()` function in the `for` loop. Look at the Go code of `loops.go` for such an example.

Multi-dimensional arrays

Arrays can have more than one dimension. However, using more than three dimensions without a serious reason can make your program difficult to read and might create bugs.



Arrays can store all the types of elements; we are just using integers here because they are easier to understand and type.

The following Go code shows how you can create an array with two dimensions (`twoD`) and another one with three dimensions (`threeD`):

```
twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12},
{13, 14, 15, 16}}
threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Accessing, assigning, or printing a single element from one of the previous two arrays can be done easily. As an example, the first element of the `twoD` array is `twoD[0][0]` and its value is 1.

Therefore, accessing all the elements of the `threeD` array with the help of multiple `for` loops can be done as follows:

```
for i := 0; i < len(threeD); i++ {
    for j := 0; j < len(v); j++ {
        for k := 0; k < len(m); k++ {
            }
        }
    }
```

As you can see, you need as many `for` loops as the dimensions of the array in order to access all of its elements. The same rule applies to slices, which will be presented in the next section. Using `x`, `y`, and `z` as variable names instead of `i`, `j`, and `k` might be a good idea here.

The code of `usingArrays.go`, which will be presented in three parts, presents a complete example of how to deal with arrays in Go.

The first part of the code is the following:

```
package main

import (
    "fmt"
)
```

```
func main() {
    anArray := [4]int{1, 2, 4, -4}
    twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}, {13, 14,
15, 16}}
    threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Here, you define three array variables named `anArray`, `twoD`, and `threeD`, respectively.

The second part of `usingArrays.go` is next:

```
fmt.Println("The length of", anArray, "is", len(anArray))
fmt.Println("The first element of", twoD, "is", twoD[0][0])
fmt.Println("The length of", threeD, "is", len(threeD))

for i := 0; i < len(threeD); i++ {
    v := threeD[i]
    for j := 0; j < len(v); j++ {
        m := v[j]
        for k := 0; k < len(m); k++ {
            fmt.Print(m[k], " ")
        }
    }
    fmt.Println()
}
```

What you get from the first `for` loop is a two-dimensional array (`threeD[i]`), whereas what you get from the second `for` loop is an array with one dimension (`v[j]`). The last `for` loop iterates over the elements of the array with one dimension.

The last code part comes with the next Go code:

```
for _, v := range threeD {
    for _, m := range v {
        for _, s := range m {
            fmt.Print(s, " ")
        }
    }
    fmt.Println()
}
```

The `range` keyword does exactly the same job as the iteration variables used in the `for` loops of the previous code segment but in a more elegant and clear way. However, if you want to know the number of iterations that are going to be executed in advance, you cannot use the `range` keyword.



The `range` keyword also works with Go maps, which makes it pretty handy and my preferred way of iteration. As you will see in [Chapter 9, Concurrency in Go – Goroutines, Channels, and Pipelines](#), the `range` keyword also works with channels.

Executing `usingArrays.go` will generate the following output:

```
$ go run usingArrays.go
The length of [1 2 4 -4] is 4
The first element of [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16]] is 1
The length of [[[1 0] [-2 4]] [[5 -1] [7 0]]] is 2
1 0 -2 4
5 -1 7 0
1 0 -2 4
5 -1 7 0
```

One of the biggest problems with arrays is out-of-bounds errors, which means trying to access an element that does not exist. This is like trying to access the sixth element of an array with only five elements. The Go compiler considers compiler issues that can be detected as compiler errors because this helps the development workflow. Therefore, the Go compiler can detect out-of-bounds array access errors:

```
./a.go:10: invalid array index -1 (index must be non-negative)
./a.go:10: invalid array index 20 (out of bounds for 2-element array)
```

The shortcomings of Go arrays

Go arrays have many disadvantages that will make you reconsider using them in your Go projects. First of all, once you define an array, you cannot change its size, which means that Go arrays are not dynamic. Putting it simply, if you need to add an element to an existing array that has no space left, you will need to create a bigger array and copy all the elements of the old array to the new one. Also, when you pass an array to a function as a parameter, you actually pass a copy of the array, which means that any changes you make to an array inside a function will be lost after the function exits. Lastly, passing a large array to a function can be pretty slow, mostly because Go has to create a copy of the array. The solution to all these problems is to use Go slices, which will be presented in the next section.



Because of their disadvantages, arrays are rarely used in Go!

Go slices

Go **slices** are very powerful and it would not be an exaggeration to say that slices could totally replace the use of arrays in Go. There are only a few occasions when you will need to use an array instead of a slice. The most obvious one is when you are absolutely sure that you will need to store a fixed number of elements.



Slices are implemented using arrays internally, which means that Go uses an underlying array for each slice.

As slices are **passed by reference** to functions, which means that what is actually passed is the memory address of the slice variable, any modifications you make to a slice inside a function will not get lost after the function exits. Additionally, passing a big slice to a function is significantly faster than passing an array with the same number of elements because Go will not have to make a copy of the slice; it will just pass the memory address of the slice variable.

Performing basic operations on slices

You can create a new **slice literal** as follows:

```
aSliceLiteral := []int{1, 2, 3, 4, 5}
```

This means that slice literals are defined just like arrays but without the element count. If you put an element count in a definition, you will get an array instead.

However, there is also the `make()` function, which allows you to create empty slices with the desired **length** and **capacity** based on the parameters passed to `make()`. The capacity parameter can be omitted. In that case, the capacity of the slice will be the same as its length. So, you can define a new empty slice with 20 places that can be automatically expanded when needed as follows:

```
integer := make([]int, 20)
```

Please note that Go automatically initializes the elements of an empty slice to the zero value of its type, which means that the value of the initialization depends on the type of the object stored in the slice. It is good to know that Go initializes the elements of every slice created with `make`.

You can access all the elements of a slice in the following way:

```
for i := 0; i < len(integer); i++ {  
    fmt.Println(integer[i])  
}
```

If you want to empty an existing slice, the zero value for a slice variable is `nil`:

```
aSliceLiteral = nil
```

You can add an element to the slice, which will automatically increase its size, using the `append()` function:

```
integer = append(integer, 12345)
```

You can access the first element of the `integer` slice as `integer[0]`, whereas you can access the last element of the `integer` slice as `integer[len(integer)-1]`.

Lastly, you can access multiple continuous slice elements using the `[:]` notation. The next statement selects the second and the third elements of a slice:

```
integer[1:3]
```

Additionally, you can use the `[:]` notation for creating a new slice from an existing slice or array:

```
s2 := integer[1:3]
```

Please note that this process is called **re-slicing** and can cause problems in some cases. Look at the next program:

```
package main  
  
import "fmt"  
  
func main() {  
  
    s1 := make([]int, 5)  
    reSlice := s1[1:3]  
    fmt.Println(s1)  
    fmt.Println(reSlice)  
  
    reSlice[0] = -100  
    reSlice[1] = 123456  
    fmt.Println(s1)  
    fmt.Println(reSlice)  
  
}
```

First, note that in order to select the second and third elements of a slice using the `[ : ]` notation, you should use `[ 1 : 3 ]`, which means starting with index number 1 and going up to index number 3, without including index number 3.



Given an array, `a1`, you can create a slice, `s1`, that references that array by executing `s1 := a1[ : ]`.

Executing the previous code, which is saved as `reslice.go`, will create the next output:

```
$ go run reslice.go
[0 0 0 0 0]
[0 0]
[0 -100 123456 0 0]
[-100 123456]
```

So, at the end of the program, the contents of the `s1` slice will be `[0 -100 123456 0 0]` even though we did not change them directly! This means that altering the elements of a re-slice modifies the element of the original slice because they both point to the same underlying array. Putting it simply, the re-slice process does not make a copy of the original slice.

The second problem from re-slicing is that even if you re-slice a slice in order to use a small part of the original slice, the underlying array from the original slice will be kept in memory for as long as the smaller re-slice exists because the original slice is being referenced by the smaller re-slice. Although this is not very important for small slices, it can cause problems when you are reading big files into slices and you want to use only a small part of them.

Slices are expanded automatically

Slices have two main properties: **capacity** and **length**. The tricky thing is that usually these two properties have different values. The length of a slice is the same as the length of an array with the same number of elements and can be found using the `len()` function. The capacity of a slice is the current room that has been allocated for this particular slice and can be found with the `cap()` function. As slices are dynamic in size, if a slice runs out of room, Go automatically doubles its current length to make room for more elements.

Putting it simply, if the length and the capacity of a slice have the same values and you try to add another element to the slice, the capacity of the slice will be doubled whereas its length will be increased by one.

Although this might work well for small slices, adding a single element to a really huge slice might take more memory than expected.

The code of `lenCap.go` illustrates the concepts of capacity and length in more detail and will be presented in three parts. The first part of the program is next:

```
package main

import (
    "fmt"
)

func printSlice(x []int) {
    for _, number := range x {
        fmt.Print(number, " ")
    }
    fmt.Println()
}
```

The `printSlice()` function helps you to print a one-dimensional slice without having to repeat the same Go code all the time.

The second part of `lenCap.go` contains the next Go code:

```
func main() {
    aSlice := []int{-1, 0, 4}
    fmt.Printf("aSlice: ")
    printSlice(aSlice)

    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
    aSlice = append(aSlice, -100)
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}
```

In this part, as well as the next one, you will add some elements to the `aSlice` slice to alter its length and its capacity.

The last portion of Go code is the following:

```
    aSlice = append(aSlice, -2)
    aSlice = append(aSlice, -3)
    aSlice = append(aSlice, -4)
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}
```


The execution of `lenCap.go` will create the next output:

```
$ go run lenCap.go
aSlice: -1 0 4
Cap: 3, Length: 3
aSlice: -1 0 4 -100
Cap: 6, Length: 4
-1 0 4 -100 -2 -3 -4
Cap: 12, Length: 7
```

As you can see, the initial size of the slice was three. As a result, the initial value of its capacity was also three. After adding one element to the slice, its size became four, whereas its capacity became six. After adding three more elements to the slice, its size became seven, whereas its capacity was doubled one more time and became 12.

Byte slices

A **byte slice** is a slice where its type is `byte`. You can create a new byte slice named `s` as follows:

```
s := make([]byte, 5)
```

Go knows that most slices of bytes are used to store strings and so makes it easy to switch between this type and the `string` type. There is nothing special in the way you can access a byte slice compared to the other types of slices. It is just that byte slices are used in file input and output operations. You will see byte slices in action in [Chapter 8, *Telling a UNIX System What to Do*](#).

The `copy()` function

You can create a slice from the elements of an existing array and you can copy an existing slice to another one using the `copy()` function. However, as the use of `copy()` can be very tricky, this subsection will try to clarify its usage with the help of the Go code of `copySlice.go`, which will be presented in four parts.



You should be very careful when using the `copy()` function on slices because the built-in `copy(dst, src)` copies the minimum number of `len(dst)` and `len(src)` elements.

The first part of the program comes with the next Go code:

```
package main

import (
    "fmt"
)

func main() {
    a6 := []int{-10, 1, 2, 3, 4, 5}
    a4 := []int{-1, -2, -3, -4}
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)

    copy(a6, a4)
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)
    fmt.Println()
}
```

So, in the preceding code, we define two slices named `a6` and `a4`, we print them, and then we try to copy `a4` to `a6`. As `a6` has more elements than `a4`, all the elements of `a4` will be copied to `a6`. However, as `a4` has only four elements and `a6` has six elements, the last two elements of `a6` will remain the same.

The second part of `copySlice.go` is next:

```
b6 := []int{-10, 1, 2, 3, 4, 5}
b4 := []int{-1, -2, -3, -4}
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
copy(b4, b6)
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
```

In this case, only the first four elements of `b6` will be copied to `b4` because `b4` has only four elements.

The third code segment of `copySlice.go` comes with the next Go code:

```
fmt.Println()
array4 := [4]int{4, -4, 4, -4}
s6 := []int{1, 1, -1, -1, 5, -5}
copy(s6, array4[0:])
fmt.Println("array4:", array4[0:])
fmt.Println("s6:", s6)
fmt.Println()
```

Here you try to copy an array with four elements to a slice with six elements. Please note that the array is converted to a slice with the help of the `[:]` notation (`array4[0:]`).

The last code portion of `copySlice.go` is the following:

```
array5 := [5]int{5, -5, 5, -5, 5}
s7 := []int{7, 7, -7, -7, 7, -7, 7}
copy(array5[0:], s7)
fmt.Println("array5:", array5)
fmt.Println("s7:", s7)
}
```

Here, you can see how you can copy a slice to an array that has space for five elements. As `copy()` only accepts slice arguments, you should also use the `[:]` notation to convert the array into a slice.

If you try to copy an array into a slice or vice versa without using the `[:]` notation, the program will fail to compile, giving one of the next error messages:

```
# command-line-arguments
./a.go:42:6: first argument to copy should be slice; have [5]int
./a.go:43:6: second argument to copy should be slice or string; have [5]int
./a.go:44:6: arguments to copy must be slices; have [5]int, [5]int
```

Executing `copySlice.go` will create the next output:

```
$ go run copySlice.go
a6: [-10 1 2 3 4 5]
a4: [-1 -2 -3 -4]
a6: [-1 -2 -3 -4 4 5]
a4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-10 1 2 3]
array4: [4 -4 4 -4]
s6: [4 -4 4 -4 5 -5]
array5: [7 7 -7 -7 7]
s7: [7 7 -7 -7 7 -7 7]
```

Multi-dimensional slices

Slices can have many dimensions just like arrays. The next statement creates a slice with two dimensions:

```
s1 := make([][]int, 4)
```



If you find yourself using slices with many dimensions all the time, you might need to reconsider your approach and choose a simpler design that does not require multi-dimensional slices.

You will find a code example with a **multi-dimensional slice** in the next section.

Another example with slices

The Go code of the `slices.go` program will hopefully clarify many things about slices and will be presented in five parts.

The first part of the program contains the expected preamble as well as the definition of two slices:

```
package main

import (
    "fmt"
)

func main() {
    aSlice := []int{1, 2, 3, 4, 5}
    fmt.Println(aSlice)
    integers := make([]int, 2)
    fmt.Println(integers)
    integers = nil
    fmt.Println(integers)
```

The second part shows how to use the `[:]` notation to create a new slice that references an existing array. Remember that you are not creating a copy of the array, just a reference to it that will be verified in the output of the program:

```
anArray := [5]int{-1, -2, -3, -4, -5}
refAnArray := anArray[:]

fmt.Println(anArray)
fmt.Println(refAnArray)
anArray[4] = -100
fmt.Println(refAnArray)
```

The third code segment defines a slice with one dimension and another one with two dimensions using the `make()` function:

```
s := make([]byte, 5)
```

```
fmt.Println(s)
twoD := make([][]int, 3)
fmt.Println(twoD)
fmt.Println()
```

As slices are automatically initialized by Go, all the elements of the two preceding slices will have the zero value of the slice type, which for integers is 0 and for slices is `nil`. Keep in mind that the elements of a multi-dimensional slice are slices.

In the fourth part of `slices.go` that comes with the next Go code, you will learn how to manually initialize all the elements of a slice with two dimensions:

```
for i := 0; i < len(twoD); i++ {
    for j := 0; j < 2; j++ {
        twoD[i] = append(twoD[i], i*j)
    }
}
```

The preceding Go code shows that in order to expand an existing slice and make it grow, you will need to use the `append()` function and not just reference an index that does not exist! The latter would create a `panic: runtime error: index out of range error` message. Please note that the values of the slice elements have been chosen arbitrarily.

The last part shows how to use the `range` keyword to visit and print all the elements of a slice with two dimensions:

```
for _, x := range twoD {
    for i, y := range x {
        fmt.Println("i:", i, "value:", y)
    }
    fmt.Println()
}
```

If you execute `slices.go`, you will get the next output:

```
$ go run slices.go
[1 2 3 4 5]
[0 0]
[]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -100]
[0 0 0 0 0]
[[] [] []]
i: 0 value: 0
i: 1 value: 0
```

```
i: 0 value: 0
i: 1 value: 1
i: 0 value: 0
i: 1 value: 2
```

It should not come as a surprise to you that the objects of the slice with the two dimensions are initialized to `nil` and therefore are printed as empty; this happens because the zero value for the slice type is `nil`.

Sorting slices using `sort.Slice()`

This subsection will illustrate the use of the `sort.Slice()` function, which was first introduced in Go version 1.8. This means that the presented code, which is saved in `sortSlice.go`, will not run on older Go versions. The program will be presented in three parts. This is the first part:

```
package main

import (
    "fmt"
    "sort"
)

type aStructure struct {
    person string
    height int
    weight int
}
```

Apart from the expected preamble, you can also see the definition of a Go structure for the first time in this book. Chapter 4, *The Uses of Composite Types*, will thoroughly explore Go structures. For now, bear in mind that structures are types with multiple variables of multiple types.

The second part of `sortSlice.go` comes with the next Go code:

```
func main() {
    mySlice := make([]aStructure, 0)
    mySlice = append(mySlice, aStructure{"Mihalis", 180, 90})
    mySlice = append(mySlice, aStructure{"Bill", 134, 45})
    mySlice = append(mySlice, aStructure{"Marietta", 155, 45})
    mySlice = append(mySlice, aStructure{"Epifanios", 144, 50})
    mySlice = append(mySlice, aStructure{"Athina", 134, 40})

    fmt.Println("0:", mySlice)
```

Here, you create a new slice named `mySlice` with elements from the `aStructure` structure created earlier.

The final part of the program is the following:

```
sort.Slice(mySlice, func(i, j int) bool {
    return mySlice[i].height < mySlice[j].height
})
fmt.Println("<:", mySlice)
sort.Slice(mySlice, func(i, j int) bool {
    return mySlice[i].height > mySlice[j].height
})
fmt.Println(">:", mySlice)
}
```

Here, you sort `mySlice` two times using `sort.Slice()` and two anonymous functions. This happens one anonymous function at a time, using the `height` field of `aStructure`.



Please note that `sort.Slice()` changes the order of the elements in the slice according to the sorting function.

Executing `sortSlice.go` will create the next output:

```
$ go run sortSlice.go
0: [{Mihalis 180 90} {Bill 134 45} {Marietta 155 45} {Epifanios 144 50}
{Athina 134 40}]
<: [{Bill 134 45} {Athina 134 40} {Epifanios 144 50} {Marietta 155 45}
{Mihalis 180 90}]
>: [{Mihalis 180 90} {Marietta 155 45} {Epifanios 144 50} {Bill 134 45}
{Athina 134 40}]
```

If you try to execute `sortSlice.go` on a UNIX machine with a Go version older than 1.8, you will get the next error message:

```
$ go version
o version go1.3.3 linux/amd64
$ go run sortSlice.go
# command-line-arguments
./sortSlice.go:24: undefined: sort.Slice
./sortSlice.go:28: undefined: sort.Slice
```

Appending an array to a slice

In this subsection, you will learn how to append an existing array to an existing slice using the technique found in `appendArrayToSlice.go`. The program will be presented in two parts. The first part is the following:

```
package main

import (
    "fmt"
)

func main() {
    s := []int{1, 2, 3}
    a := [3]int{4, 5, 6}
```

So far, we have just created and initialized a slice named `s` and an array named `a`.

The second part of `appendArrayToSlice.go` is as follows:

```
    ref := a[:]
    fmt.Println("Existing array:\t", ref)
    t := append(s, ref...)
    fmt.Println("New slice:\t", t)
    s = append(s, ref...)
    fmt.Println("Existing slice:\t", s)
    s = append(s, s...)
    fmt.Println("s+s:\t\t", s)
}
```

Two important things are happening here. The first is that we create a new slice named `t` that contains the elements of `a + s`, we append the array named `a` to the slice named `s`, and we store the result to the `s` slice. So, you have a choice on whether to store the new slice in an existing slice variable or not. This mainly depends on what you want to accomplish.

The second important thing is that you have to create a reference to the existing array (`ref := a[:]`) for this to work. Please notice the way the `ref` variable is used in the two `append()` calls: the three dots (`...`) are exploding the array into arguments that are appended to the existing slice.

The last two statements of the program show how you can copy a slice to the end of itself. The three dots (`...`) are still required.

Executing `appendArrayToSlice.go` will generate the following output:

```
$ go run appendArrayToSlice.go
Existing array:    [4 5 6]
New slice:        [1 2 3 4 5 6]
Existing slice:   [1 2 3 4 5 6]
s+s:              [1 2 3 4 5 6 1 2 3 4 5 6]
```

Go maps

A Go **map** is equivalent to the well-known **hash table** found in many other programming languages. The main advantage of maps is that they can use any data type as their index, which in this case is called a **map key** or just a **key**. Although Go maps do not exclude any data types from being used as keys, for a data type to be used as a key it must be **comparable**, which means that the Go compiler must be able to differentiate one key from another or, putting it simply, that the keys of a map must support the `==` operator.

The good news is that almost all data types are comparable. However, as you can imagine, using the `bool` data type as the key to a map will definitely limit your options. Additionally, using floating-point numbers as keys might present problems caused by the precision used for different machines and operating systems.



As mentioned, a Go map is a reference to a hash table. The good thing is that Go hides the implementation of the hash table and therefore its complexity. You will learn more about implementing a hash table on your own in Go in [Chapter 5, How to Enhance Go Code with Data Structures](#).

You can create a new empty map with `string` keys and `int` values with the help of the `make()` function:

```
iMap = make(map[string]int)
```

Alternatively, you can use the next **map literal** in order to create a new map that will be populated with data:

```
anotherMap := map[string]int {
    "k1": 12
    "k2": 13
}
```

You can access the two objects of `anotherMap` as `anotherMap["k1"]` and `anotherMap["k2"]`. You can delete an object of a map using the `delete()` function:

```
delete(anotherMap, "k1")
```

You can iterate over all the elements of a map using the next technique:

```
for key, value := range iMap {  
    fmt.Println(key, value)  
}
```

The Go code of `usingMaps.go` will illustrate the use of maps in more detail. The program will be presented in three parts. The first part comes with the following Go code:

```
package main  
  
import (  
    "fmt"  
)  
  
func main() {  
  
    iMap := make(map[string]int)  
    iMap["k1"] = 12  
    iMap["k2"] = 13  
    fmt.Println("iMap:", iMap)  
  
    anotherMap := map[string]int{  
        "k1": 12,  
        "k2": 13,  
    }  
}
```

The second part of `usingMaps.go` contains the next code:

```
fmt.Println("anotherMap:", anotherMap)  
delete(anotherMap, "k1")  
delete(anotherMap, "k1")  
delete(anotherMap, "k1")  
fmt.Println("anotherMap:", anotherMap)  
  
_, ok := iMap["doesItExist"]  
if ok {  
    fmt.Println("Exists!")  
} else {  
    fmt.Println("Does NOT exist")  
}
```

Here you see a technique that allows you to determine whether a given key is in the map or not. This is a vital technique because without it you would not know whether a given map has the required information or not.



The bad thing is that if you try to get the value of a map key that does not exist in the map, you will end up getting zero, which gives you no way of determining whether the result was zero because the key you requested was not there or because the element with the corresponding key actually had a zero value. This is why we have `_`, `ok` in maps.

Additionally, you can see the `delete()` function in action. Calling the same `delete()` statement multiple times does not make any difference and does not generate any warning messages.

The last part of the program is next:

```
for key, value := range iMap {  
    fmt.Println(key, value)  
}
```

Here, you see the use of the `range` keyword on a map, which is pretty elegant and handy.

If you execute `usingMaps.go`, you will get the next output:

```
$ go run usingMaps.go  
iMap: map[k1:12 k2:13]  
anotherMap: map[k1:12 k2:13]  
anotherMap: map[k2:13]  
Does NOT exist  
k1 12  
k2 13
```



Please note that you cannot and should not make any assumptions about the order the map pairs are going to be displayed on your screen because that order is totally random.

Storing to a nil map

The following Go code will work:

```
aMap := map[string]int{}  
aMap["test"] = 1
```

However, the next Go code will not work because you have assigned the `nil` value to the map you are trying to use:

```
aMap := map[string]int{}  
// var aMap map[string]int  
aMap = nil  
fmt.Println(aMap)  
aMap["test"] = 1
```

Saving the preceding code to `failMap.go` and trying to compile it will generate the next error message:

```
$ go run failMap.go  
map[]  
panic: assignment to entry in nil map  
...
```

This means that trying to insert data to a `nil` map will fail. However, looking up, deleting, finding the length, and using `range` loops on `nil` maps will not crash your code.

When you should use a map

Maps are more versatile than both slices and arrays but this flexibility comes at a cost: the extra processing power required for the implementation of a Go map. However, built-in Go structures are very fast, so do not hesitate to use a Go map when you need to. What you should remember is that Go maps are very convenient and can store many different kinds of data, while being both easy to understand and fast to work with.

Go constants

Go supports **constants**, which are variables that cannot change their values. Constants in Go are defined with the help of the `const` keyword.



Generally speaking, constants are **global variables**, so you might rethink your approach if you find yourself defining too many constant variables with a local scope.

The main benefit you get from using constants in your programs is the guarantee that their value will not change during program execution. Strictly speaking, the value of a constant variable is defined at compile time not at run time.

Behind the scenes, Go uses Boolean, string, or number as the type for storing a constant variable because this gives Go more flexibility when dealing with constants.

You can define a new constant as follows:

```
const HEIGHT = 200
```

Please note that in Go we do not use ALL CAPS for constants; this is just a personal preference of mine.

Additionally, if you want to declare many constants at once, mainly because they are related to each other, you can use the next notation:

```
const (  
    C1 = "C1C1C1"  
    C2 = "C2C2C2"  
    C3 = "C3C3C3"  
)
```

Please note that the Go compiler considers the results of all operations applied to constants as constants. However, if a constant is part of a larger expression, this will not be the case.

Now, for something completely different, let's look at the following three variable declarations that mean exactly the same thing in Go:

```
s1 := "My String"  
var s2 = "My String"  
var s3 string = "My String"
```

However, as none of them have the `const` keyword in their declaration, none of them are constants. This does not mean that you cannot define two constants in a similar way:

```
const s1 = "My String"  
const s2 string = "My String"
```

Although both `s1` and `s2` are constants, `s2` comes with a type declaration (`string`), which makes its declaration more restrictive than the declaration of `s1`. This is because a typed Go constant must follow all the strict rules of a typed Go variable. On the other hand, a constant without a type doesn't need to follow all the strict rules of a typed variable, which means that it can be mixed with expressions more liberally. Additionally, even constants without a type have a default type that is used when, and only when, no other type information is available. The main reason for this behavior is that as you do not know how a constant is going to be used, you do not desire to use all the available Go rules.

A simple example is the definition of a numeric constant such as `const value = 123`. As you might use the `value` constant in many expressions, declaring a type would make your job much more difficult. Look at the next Go code:

```
const s1 = 123
const s2 float64 = 123

var v1 float32 = s1 * 12
var v2 float32 = s2 * 12
```

Although the compiler will not have a problem with the definition of `v1`, the code used for the definition of `v2` will not compile because `s2` and `v2` have different types:

```
$ go run a.go
# command-line-arguments
./a.go:12:6: cannot use s2 * 12 (type float64) as type float32 in
assignment
```

As general advice, if you are using lots of constants in your programs, it might be a good idea to gather all of them in a Go package or a Go structure.

The constant generator `iota`

The **constant generator `iota`** is used for declaring a sequence of related values that use incrementing numbers without the need to explicitly type each one of them.

Most of the concepts related to the `const` keyword, including the constant generator `iota`, will be illustrated in the `constants.go` file, which will be presented in four parts.

The first code segment of `constants.go` is next:

```
package main

import (
    "fmt"
)

type Digit int
type Power2 int

const PI = 3.1415926

const (
```

```
C1 = "C1C1C1"  
C2 = "C2C2C2"  
C3 = "C3C3C3"  
)
```

In this part, we declare two new types named `Digit` and `Power2`, and four new constants named `PI`, `C1`, `C2`, and `C3`.



A Go **type** is a way of defining a new **named type** that uses the same underlying type as an existing type. This is mainly used for differentiating between different types that might use the same kind of data.

The second part of `constants.go` comes with the next Go code:

```
func main() {  
    const s1 = 123  
    var v1 float32 = s1 * 12  
    fmt.Println(v1)  
    fmt.Println(PI)  
}
```

In this part, you define another constant (`s1`) that is used in an expression (`v1`).

The third part of the program is as follows:

```
const (  
    Zero Digit = iota  
    One  
    Two  
    Three  
    Four  
)  
fmt.Println(One)  
fmt.Println(Two)
```

Here you see the definition of a **constant generator** `iota` based on `Digit`, which is equivalent to the next declaration of four constants:

```
const (  
    Zero = 0  
    One = 1  
    Two = 2  
    Three = 3  
    Four = 4  
)
```

The last portion of `constants.go` is the following:

```
const (  
    p2_0 Power2 = 1 << iota  
    _  
    p2_2  
    _  
    p2_4  
    _  
    p2_6  
)  
  
fmt.Println("2^0:", p2_0)  
fmt.Println("2^2:", p2_2)  
fmt.Println("2^4:", p2_4)  
fmt.Println("2^6:", p2_6)  
  
}
```

There is another constant generator `iota` here that is a little different than the previous one. Firstly, you can see the use of the underscore character in a `const` block with a constant generator `iota`, which allows you to skip unwanted values. Secondly, the value of `iota` always increments and can be used in expressions, which is what occurred in this case.

Now let us see what really happens inside the `const` block. For `p2_0`, `iota` has the value of 0 and `p2_0` is defined as 1. For `p2_2`, `iota` has the value of 2 and `p2_2` is defined as the result of the expression `1 << 2`, which is 00000100 in binary representation. The decimal value of 00000100 is 4, which is the result and the value of `p2_2`. Analogously, the value of `p2_4` is 16 and the value of `p2_6` is 32.

As you can see, the use of `iota` can save your time when it fits your needs.

Executing `constants.go` will generate the next output:

```
$ go run constants.go  
1476  
3.1415926  
1  
2  
2^0: 1  
2^2: 4  
2^4: 16  
2^6: 64
```


Go pointers

Go supports **pointers**, which are memory addresses that offer improved speed in exchange for difficult-to-debug code and nasty bugs. Ask any C programmer you know to learn more about this.

You have already seen pointers in action in [Chapter 2, *Understanding Go Internals*](#), when we talked about unsafe code and the `unsafe` package, as well as the Go garbage collector, but this section will try to shed more light on this difficult and tricky subject. Additionally, native Go pointers are safe provided that you know what you are doing.

When working with pointers, you need `*` to get the value of a pointer, which is called **dereferencing the pointer**, and `&` to get the memory address of a non-pointer variable.



Generally speaking, amateur developers should use pointers only when the libraries they use require it because pointers can be the cause of horrible and difficult-to-discover bugs when used carelessly.

You can make a function accept a pointer parameter as follows:

```
func getPointer(n *int) {  
}
```

Similarly, a function can return a pointer as follows:

```
func returnPointer(n int) *int {  
}
```

The use of safe Go pointers is illustrated in `pointers.go`, which will be presented in four parts. The first code segment of `pointers.go` is next:

```
package main  
  
import (  
    "fmt"  
)  
  
func getPointer(n *int) {  
    *n = *n * *n  
}  
  
func returnPointer(n int) *int {  
    v := n * n  
    return &v  
}
```

```
}
```

The good thing with `getPointer()` is that it allows you to update the variable passed to it without the need to return anything to the caller function. This happens because the pointer passed as a parameter contains the memory address of the variable.

On the other hand, `returnPointer()` gets an integer parameter and returns a pointer to an integer, which is denoted by `return &v`. Although this might not look that useful, you will really appreciate this capability in Chapter 4, *The Uses of Composite Types*, when we talk about pointers to Go structures as well as in later chapters where more complex data structures will be involved.

Both the `getPointer()` and `returnPointer()` functions find the square of an integer. However, they use a totally different approach as `getPointer()` stores the result to the provided parameter whereas `returnPointer()` returns the result and requires a different variable for storing it.

The second part of the program contains the next Go code:

```
func main() {  
    i := -10  
    j := 25  
  
    pI := &i  
    pJ := &j  
  
    fmt.Println("pI memory:", pI)  
    fmt.Println("pJ memory:", pJ)  
    fmt.Println("pI value:", *pI)  
    fmt.Println("pJ value:", *pJ)  
}
```

Both `i` and `j` are normal integer variables. However, `pI` and `pJ` are both pointers pointing to `i` and `j`, respectively. `pI` is the memory address of the pointer, whereas `*pI` is the value stored to that memory address.

The third part of `pointers.go` is the following:

```
*pI = 123456  
*pI--  
fmt.Println("i:", i)
```

Here, you can see how you can change the `i` variable through the `pI` pointer that points to `i` in two different ways: firstly, by directly assigning a new value to it and secondly, by using the `--` operator.

The last code portion of `pointers.go` comes with the next Go code:

```
    getPointer(pJ)
    fmt.Println("j:", j)
    k := returnPointer(12)
    fmt.Println(*k)
    fmt.Println(k)
}
```

Here you call the `getPointer()` function using `pJ` as its parameter. As we talked about before, any changes made to the variable that `pJ` points to inside `getPointer()` will have an effect on the value of the `j` variable, which will be verified by the output of the `fmt.Println("j:", j)` statement. The call to `returnPointer()` returns a pointer that is assigned to the `k` pointer variable.

Running `pointers.go` will create the next output:

```
$ go run pointers.go
pI memory: 0xc0000160b0
pJ memory: 0xc0000160b8
pI value: -10
pJ value: 25
i: 123455
j: 625
144
0xc0000160f0
```

I recognize that you might have trouble understanding the Go code of `pointers.go` because we have not talked about functions and function definitions yet. Feel free to look at Chapter 6, *What You Might Not Know About Go Packages and Go Functions*, where things related to functions are explained in more detail.



Please note that strings in Go are value types not pointers as is the case in C.

Why use pointers?

There are two main reasons for using pointers in your programs:

- Pointers allow you to share data, especially between Go functions.
- Pointers can be extremely useful when you want to differentiate between a zero value and a value that is not set.

Times and dates

In this section, you are going to learn how to parse time and date strings in Go, how to convert between different time and date formats, and how to print times and dates in the format you desire. Although this task might look insignificant at first, it can be truly critical when you want to synchronize multiple tasks or when your application needs to read the date from one or more text files, or directly from the user.

The `time` package is the star of working with times and dates in Go; you will see some of its functions in action in this section.

Before learning how to parse a string and convert it into a time or a date, you will see a simple program named `usingTime.go` that will introduce you to the `time` package. The program will be presented in three parts. The first part is the following:

```
package main

import (
    "fmt"
    "time"
)
```

The second code segment of `usingTime.go` comes with the next Go code:

```
func main() {
    fmt.Println("Epoch time:", time.Now().Unix())
    t := time.Now()
    fmt.Println(t, t.Format(time.RFC3339))
    fmt.Println(t.Weekday(), t.Day(), t.Month(), t.Year())

    time.Sleep(time.Second)
    t1 := time.Now()
    fmt.Println("Time difference:", t1.Sub(t))
}
```

The `time.Now().Unix()` function returns the **UNIX epoch time**, which is the number of seconds that have elapsed since 00:00:00 UTC, 1 January, 1970. The `Format()` function allows you to convert a time variable to another format; in this case, the RFC3339 format.

You will see the `time.Sleep()` function many times in this book as a naive way of emulating the delay from the execution of a true function. The `time.Second` constant allows you to use a one-second duration in Go. If you want to define a duration of 10 seconds, you will need to multiply `time.Second` by 10. Other similar constants include `time.Nanosecond`, `time.Microsecond`, `time.Millisecond`, `time.Minute`, and `time.Hour`. So, the smallest amount of time that can be defined with the `time` package is the nanosecond. Lastly, the `time.Sub()` function allows you to find the time difference between two times.

The last part of the program is next:

```
formatT := t.Format("01 January 2006")
fmt.Println(formatT)
loc, _ := time.LoadLocation("Europe/Paris")
londonTime := t.In(loc)
fmt.Println("Paris:", londonTime)
}
```

Here you define a new date format using `time.Format()` in order to use it for printing out a time variable.

Executing `usingTime.go` will generate the next output:

```
$ go run usingTime.go
Epoch time: 1548753515
2019-01-29 11:18:35.01478 +0200 EET m=+0.000339641
2019-01-29T11:18:35+02:00
Tuesday 29 January 2019
Time difference: 1.000374985s
01 January 2019
Paris: 2019-01-29 10:18:35.01478 +0100 CET
```

Now that you know the basics of the `time` package, it is time to dig deeper into its functionality, starting with working with times.

Working with times

When you have a `time` variable, it is easy to convert it into anything that is related to time or date. However, the main problem is when you have a string and you want to see whether this is a valid time or not. The function used for parsing time and date strings is called `time.Parse()` and it accepts two parameters. The first one denotes the expected format of the string that you are going to parse, whereas the second parameter is the actual string that is going to be parsed. The first parameter is composed of elements from a list of Go constants related to date and time parsing.



The list of constants that can be used for creating your own parse format can be found at <https://golang.org/src/time/format.go>. Go does not define the format of a date or a time in a form like `DDYYYYMM` or `%D %Y %M` as other programming languages do but uses its own approach. Although you might find this approach strange at first, you will certainly appreciate it as it prevents the developer from making silly mistakes.

The Go constants for working with times are `15` for parsing the hour, `04` for parsing the minutes, and `05` for parsing the seconds. You can easily guess that all these numeric values must be unique. You can use `PM` for parsing the `PM` string in uppercase and `pm` for parsing the lowercase version.

Please note that you are not obligated to use every available Go constant. The main task of the developer is putting the various Go constants in the desired order to match the kind of strings that the program will have to process. You can consider the final version of the string that is passed as the first parameter to the `time.Parse()` function as a **regular expression**.

Parsing times

This section will tell you how to parse a string, which is given as a command-line argument to the `parseTime.go` utility in order to convert it into a `time` variable. However, this is not always possible because the given string might not be in the correct format or might contain invalid characters. The `parseTime.go` utility will be presented in three parts.

The first code segment of `parseTime.go` is next:

```
package main

import (
    "fmt"
    "os"
```

```
    "path/filepath"  
    "time"  
)
```

The second part comes with the next code:

```
func main() {  
    var myTime string  
    if len(os.Args) != 2 {  
        fmt.Printf("usage: %s string\n", filepath.Base(os.Args[0]))  
        os.Exit(1)  
    }  
  
    myTime = os.Args[1]
```

The last portion of `parseTime.go`, which is where the magic happens, is the following:

```
    d, err := time.Parse("15:04", myTime)  
    if err == nil {  
        fmt.Println("Full:", d)  
        fmt.Println("Time:", d.Hour(), d.Minute())  
    } else {  
        fmt.Println(err)  
    }  
}
```

In order to parse an hour and minute string, you will need to use "15:04". The value of the `err` variable tells you whether the parsing was successful or not.

Executing `parseTime.go` will create the next kind of output:

```
$ go run parseTime.go  
usage: parseTime string  
exit status 1  
$ go run parseTime.go 12:10  
Full: 0000-01-01 12:10:00 +0000 UTC  
Time: 12 10
```

As you can see here, Go prints a full date and time string because this is what is stored in a `time` variable. If you are only interested in the time and not in the date, you should print the parts of a `time` variable that you want.

If you use a wrong Go constant like 22:04 when trying to parse a string and convert it into a time, you will get the next error message:

```
$ go run parseTime.go 12:10  
parsing time "12:10" as "22:04": cannot parse ":10" as "2"
```

However, if you use a Go constant like `11` that is used for parsing months when the month is given as a number, the error message will be slightly different:

```
$ go run parseTime.go 12:10
parsing time "12:10": month out of range
```

Working with dates

In this subsection, you will learn how to parse strings that denote dates in Go, which still requires the use of the `time.Parse()` function.

The Go constants for working with dates are `Jan` for parsing the three-letter abbreviation used for describing a month, `2006` for parsing the year, and `02` for parsing the day of the month. If you use `January` instead of `Jan`, you will get the long name of the month instead of its three-letter abbreviation, which makes perfect sense.

Additionally, you can use `Monday` for parsing strings that contain a long weekday string and `Mon` for the abbreviated version of the weekday.

Parsing dates

The name of the Go program that will be developed in this subsection is `parseDate.go` and it will be presented in two parts.

The first part of `parseDate.go` is next:

```
package main

import (
    "fmt"
    "os"
    "path/filepath"
    "time"
)

func main() {

    var myDate string
    if len(os.Args) != 2 {
        fmt.Printf("usage: %s string\n",
            filepath.Base(os.Args[0]))
        return
    }
```



```
myDate = os.Args[1]
```

The second part of `parseDate.go` contains the following Go code:

```
d, err := time.Parse("02 January 2006", myDate)
if err == nil {
    fmt.Println("Full:", d)
    fmt.Println("Time:", d.Day(), d.Month(), d.Year())
} else {
    fmt.Println(err)
}
}
```

If there is a character such as `-` between the name of the month and the year, you can use `"02 January-2006"` instead of `"02 January 2006"` as the first parameter to `time.Parse()`.

Executing `parseDate.go` will generate the next output:

```
$ go run parseDate.go
usage: parseDate string
$ go run parseDate.go "20 July 2000"
Full: 2000-07-20 00:00:00 +0000 UTC
Time: 20 July 2000
```

As `parseDate.go` does not expect data about the time, the `00:00:00 +0000 UTC` string is automatically added at the end of the full date and time string.

Changing date and time formats

In this section, you will learn how to change the format of a string that contains both a date and a time. A very common place for finding such strings is the log files of web servers such as **Apache** and **Nginx**. As we do not know how to read a text file line by line, the text will be hard coded in the program; however, this fact does not change the functionality of the program.

The Go code of `timeDate.go` will be presented in four parts. The first part is the expected preamble:

```
package main

import (
    "fmt"
    "regexp"
    "time"
```

```
)
```

You need the `regexp` standard Go package for supporting regular expressions.

The second code portion of `timeDate.go` is the following:

```
func main() {

    logs := []string{"127.0.0.1 - - [16/Nov/2017:10:49:46 +0200]
325504",          "127.0.0.1 - - [16/Nov/2017:10:16:41 +0200] \"GET /CVEN
HTTP/1.1\" 200 12531 \"-\" \"Mozilla/5.0 AppleWebKit/537.36\",
"127.0.0.1 200 9412 - - [12/Nov/2017:06:26:05 +0200]
\"GET \"http://www.mtsoukalos.eu/taxonomy/term/47\" 1507",
        "[12/Nov/2017:16:27:21 +0300]",
        "[12/Nov/2017:20:88:21 +0200]",
        "[12/Nov/2017:20:21 +0200]",
    }
```

As you cannot be sure about your data and its format, the sample data used for this program tries to cover many different cases, including incomplete data like `[12/Nov/2017:20:21 +0200]` where there are no seconds in the time part, and erroneous data such as `[12/Nov/2017:20:88:21 +0200]` where the value of the minutes is 88.

The third part of `timeDate.go` contains the next Go code:

```
for _, logEntry := range logs {
    r :=
    regexp.MustCompile(`.*[ (\\d\\d\\/\\w+\\/\\d\\d\\d\\d:\\d\\d:\\d\\d:\\d\\d.*)`
    if r.MatchString(logEntry) {
        match := r.FindStringSubmatch(logEntry)
```

The main benefit you get from such a difficult-to-read regular expression in this particular program is that it allows your code to find out whether you have a date and time string somewhere in your line or not. After you get that string, you will feed `time.Parse()` with it and let `time.Parse()` do the rest of the job.

The last part of the program comes with the next Go code:

```
        dt, err := time.Parse("02/Jan/2006:15:04:05 -0700",
match[1])

        if err == nil {
            newFormat := dt.Format(time.RFC850)
            fmt.Println(newFormat)
        } else {
            fmt.Println("Not a valid date time format!")
        }
    } else {
```

```
        fmt.Println("Not a match!")
    }
}
```

Once you find a string that matches the regular expression, you parse it using `time.Parse()` to make sure that it is a valid date and time string. If yes, `timeDate.go` will print the date and time according to the **RFC850 format**.

If you execute `timeDate.go`, you will get the next kind of output:

```
$ go run timeDate.go
Thursday, 16-Nov-17 10:49:46 EET
Thursday, 16-Nov-17 10:16:41 EET
Sunday, 12-Nov-17 06:26:05 EET
Sunday, 12-Nov-17 16:27:21 +0300
Not a valid date time format!
Not a match!
```

Measuring execution time

In this section, you are going to learn how to measure the execution time of one or more commands in Go. The same technique can be applied for measuring the execution time of a function or a group of functions. The name of the Go program is `execTime.go` and it will be presented in three parts.



This is an easy-to-implement technique that is both very powerful and handy. Do not underestimate the simplicity of Go.

The first part of `execTime.go` is as follows:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    start := time.Now()
    time.Sleep(time.Second)
    duration := time.Since(start)
    fmt.Println("It took time.Sleep(1)", duration, "to finish.")
}
```

You will need the functionality offered by the `time` package in order to measure the execution time of a command. All the job is done by the `time.Since()` function that accepts a single argument, which should be a time in the past. In this case, we are measuring the time it took Go to execute a `time.Sleep(time.Second)` call as this is the only statement between `time.Now()` and `time.Since()`.

The second part of `execTime.go` has the following code:

```
start = time.Now()
time.Sleep(2 * time.Second)
duration = time.Since(start)
fmt.Println("It took time.Sleep(2)", duration, "to finish.")
```

This time we are measuring the time it took Go to execute a `time.Sleep(2 * time.Second)` call. This can be very useful for finding out how accurate the `time.Sleep()` function is, which mainly has to do with how accurate the internal Go clock is.

The last part of `execTime.go` is as follows:

```
start = time.Now()
for i := 0; i < 200000000; i++ {
    _ = i
}
duration = time.Since(start)
fmt.Println("It took the for loop", duration, "to finish.")

sum := 0
start = time.Now()
for i := 0; i < 200000000; i++ {
    sum += i
}
duration = time.Since(start)
fmt.Println("It took the for loop", duration, "to finish.")
```

In the last part, we are measuring the speed of two `for` loops. The first one does nothing, whereas the second one does some calculations. As you will see in the output of the program, the second `for` loop is faster than the first one.

Executing `execTime.go` will generate the following kind of output:

```
$ go run execTime.go
It took time.Sleep(1) 1.000768881s to finish.
It took time.Sleep(2) 2.00062487s to finish.
It took the for loop 50.497931ms to finish.
It took the for loop 47.70599ms to finish.
```

Measuring the operation of the Go garbage collector

Now we can rewrite `sliceGC.go`, `mapNoStar.go`, `mapStar.go`, and `mapSplit.go` from the previous chapter and get more accurate results from them without the need to use the `time(1)` UNIX command-line utility. Actually, the only thing that needs to be done in each one of these files is embed the call to `runtime.GC()` between `time.Now()` and `time.Since()`, and print the results. The updated versions of `sliceGC.go`, `mapNoStar.go`, `mapStar.go`, and `mapSplit.go` will be called `sliceGCTime.go`, `mapNoStarTime.go`, `mapStarTime.go`, and `mapSplitTime.go`, respectively.

Executing the updated versions will generate the following output:

```
$ go run sliceGCTime.go
It took GC() 281.563µs to finish
$ go run mapNoStarTime.go
It took GC() 9.483966ms to finish
$ go run mapStarTime.go
It took GC() 651.704424ms to finish
$ go run mapSplitTime.go
It took GC() 12.743889ms to finish
```

These results are far more accurate than before because they only show the time it took `runtime.GC()` to execute without including the time it took the program to populate the slice or the map used for storing the values. Nevertheless, the results still verify the findings about how slow it is for the Go garbage collector to deal with map variables with lots of data.

Web links and exercises

- Write a constant generator `iota` for the days of the week.
- Write a Go program that converts an existing array into a map.
- Visit the documentation page of the `time` package, which can be found at <https://golang.org/pkg/time/>.
- Can you write a constant generator `iota` for the powers of the number four?
- You can also visit the GitHub page where Go 2 and the changes to number literals are being discussed, which will help you to understand how changes in Go are happening:
<https://github.com/golang/proposal/blob/master/design/19308-number-literals.md>.

- Write your own version of `parseDate.go`.
- Write your own version of `parseTime.go`. Do not forget to test your program.
- Can you create a version of `timeDate.go` that can process two date and time formats?

Summary

In this chapter, you learned about many interesting Go topics, including numeric data types, maps, arrays, and slices, as well as Go pointers, Go constants and loops, and how Go allows you to work with dates and times. You should understand by now why slices are superior to arrays.

The next chapter will be about building and using composite types in Go, which mainly includes types that are created with the `struct` keyword and are called **structures**. After that, we will talk about `string` variables and **tuples**.

Additionally, the next chapter will talk about **regular expressions** and **pattern matching**, which are tricky subjects not only in Go but also in every other programming language. However, when used properly and carefully, regular expressions and pattern matching can make the life of a developer so much easier that it is totally worth learning more about them.

JSON is a very popular text format, so the next chapter will also discuss how you can create, import, and export JSON data in Go.

Lastly, you will learn about the `switch` keyword and the `strings` package, which allows you to manipulate **UTF-8** strings.

4

The Uses of Composite Types

In the previous chapter, we talked about many core Go topics, including numeric data types, arrays, slices, maps, pointers, constants, the `for` loop, the `range` keyword, and how to work with times and dates.

This chapter will explore more advanced Go features, such as **tuples** and **strings**, the `strings` standard Go package, and the `switch` statement, but, most importantly, it will look at **structures**, which are used extensively in Go.

The chapter will also show you how to work with **JavaScript Object Notation (JSON)** and **Extensible Markup Language (XML)** text files, how to implement a simple **key-value store**, how to define **regular expressions**, and how to perform **pattern matching** in Go.

The following topics will be covered:

- Go **structures** and the `struct` keyword
- Go **tuples**
- Go strings, runes, and string literals
- Working with the **JSON** text format
- Working with the **XML** text format
- Regular expressions in Go
- Pattern matching in Go
- The `switch` statement
- The functionality that the `strings` package offers
- Calculating **Pi** with high accuracy
- Developing a key-value store

About composite types

Although standard Go types are pretty handy, fast, and flexible, they most likely cannot cover every type of data you want to support in your Go code. Go solves this problem by supporting **structures**, which are custom types defined by the developer. Additionally, Go has its own way of supporting **tuples**, which mainly allows functions to return multiple values without the need to group them in structures as is the case in C.

Structures

Although arrays, slices, and maps are all very useful, they cannot group and hold multiple values in the same place. When you need to group various types of variables and create a new handy type, you can use a structure. The various elements of a structure are called the **fields of the structure** or just fields.

I will start this section by explaining a simple structure that was first defined in the `sortSlice.go` source file of the previous chapter:

```
type aStructure struct {  
    person string  
    height int  
    weight int  
}
```

For reasons that will become evident in Chapter 6, *What You Might Not Know About Go Packages and Go Functions*, the fields of a structure usually begin with an uppercase letter – this mainly depends on what you want to do with the fields. This structure has three fields named `person`, `height`, and `weight`, respectively. You can create a new variable of the `aStructure` type as follows:

```
var s1 aStructure
```

Additionally, you can access a specific field of a structure by its name. So, in order to get the value of the `person` field of the `s1` variable, you should type `s1.person`.

A **structure literal** can be defined as follows:

```
p1 := aStructure{"fmt", 12, -2}
```

However, since remembering the order of the fields of a structure can be pretty hard, Go allows you to use another form for defining a structure literal:

```
p1 := aStructure{weight: 12, height: -2}
```


In this case, you do not need to define an initial value for every field of the structure.

Now that you know the basics of structures, it is time to show you a more practical example. It is named `structures.go` and will be presented in four parts.

The first part of `structures.go` contains the following code:

```
package main

import (
    "fmt"
)
```



Structures, in particular, and Go types, in general, are usually defined outside the `main()` function in order that they have a global scope and are available to the entire Go package, unless you want to clarify that a type is only useful within the current scope and is not expected to be used elsewhere.

The second code segment from `structures.go` is shown in the following Go code:

```
func main() {

    type XYZ struct {
        X int
        Y int
        Z int
    }

    var s1 XYZ
    fmt.Println(s1.Y, s1.Z)
```

As you can see, there is nothing that prevents you from defining a new structure type inside a function, but you should have a reason for doing so.

The third portion of `structures.go` follows:

```
p1 := XYZ{23, 12, -2}
p2 := XYZ{Z: 12, Y: 13}
fmt.Println(p1)
fmt.Println(p2)
```

Here you define two structure literals named `p1` and `p2`, which you print afterward.

The last part of `structures.go` contains the following Go code:

```
pSlice := [4]XYZ{}
```

```
pSlice[2] = p1
pSlice[0] = p2
fmt.Println(pSlice)
p2 = XYZ{1, 2, 3}
fmt.Println(pSlice)
}
```

In this last part, we created an array of structures named `pSlice`. As you will understand from the output of `structures.go`, when you assign a structure to an array of structures, the structure is copied into the array so changing the value of the original structure will have no effect on the objects of the array.

Executing `structures.go` will generate the next output:

```
$ go run structures.go
0 0
{23 12 -2}
{0 13 12}
[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
[{0 13 12} {0 0 0} {23 12 -2} {0 0 0}]
```



Note that the order in which you put the fields in the definition of a structure type is significant for the **type identity** of the defined structure. Put simply, two structures with the same fields will not be considered identical in Go if their fields are not in exactly the same order.

The output of `structures.go` illustrates that the zero value of a `struct` variable is constructed by zeroing all the fields of the `struct` variable according to their types.

Pointers to structures

In Chapter 3, *Working with Basic Go Data Types*, we talked about pointers. In this section, we will look at an example that is related to **pointers to structures**. The name of the program will be `pointerStruct.go` and will be presented in four parts.

The first part of the program contains the next Go code:

```
package main

import (
    "fmt"
)

type myStructure struct {
    Name    string
```

```
Surname string
Height  int32
}
```

The second code segment from `pointerStruct.go` follows:

```
func createStruct(n, s string, h int32) *myStructure {
    if h > 300 {
        h = 0
    }
    return &myStructure{n, s, h}
}
```

The approach used in `createStruct()` for creating a new structure variable has many advantages over initializing structure variables on your own, including the fact that you are allowed to check whether the provided information is both correct and valid. Additionally, this approach is cleaner – there is a central point where structure variables are initialized so if there is something wrong with your `struct` variables, you know where to look and who to blame! Note that some people might prefer to name the `createStruct()` function `NewStruct()`.



For those with a C or C++ background, it is perfectly legal for a Go function to return the memory address of a local variable. Nothing gets lost, so everybody is happy.

The third portion of `pointerStruct.go` is as follows:

```
func retStructure(n, s string, h int32) myStructure {
    if h > 300 {
        h = 0
    }
    return myStructure{n, s, h}
}
```

This part presents the no-pointer version of the `createStruct()` function named `retStructure()`. Both functions work fine, so choosing between the implementation of `createStruct()` and `retStructure()` is just a matter of personal preference. More appropriate names for these two functions might have been `NewStructurePointer()` and `NewStructure()`, respectively.

The last part of `pointerStruct.go` is shown in the following Go code:

```
func main() {
    s1 := createStruct("Mihalis", "Tsoukalos", 123)
    s2 := retStructure("Mihalis", "Tsoukalos", 123)
```

```
    fmt.Println((*s1).Name)
    fmt.Println(s2.Name)
    fmt.Println(s1)
    fmt.Println(s2)
}
```

If you execute `pointerStruct.go`, you will get the next output:

```
$ go run pointerStruct.go
Mihalis
Mihalis
&{Mihalis Tsoukalos 123}
{Mihalis Tsoukalos 123}
```

Here you can see one more time that the main difference between `createStruct()` and `retStructure()` is that the former returns a pointer to a structure, which means that you will need to dereference that pointer in order to use the object it points to, whereas the latter returns an entire structure object. This can make your code a little uglier.



Structures are very important in Go and are used extensively in real-world programs because they allow you to group as many values as you want and treat those values as a single entity.

Using the new keyword

Go supports the `new` keyword, which allows you to allocate new objects. However, there is a very important detail that you need to remember about `new`: `new` returns the memory address of the allocated object. Put simply, `new` returns a pointer.

So, you can create a fresh `aStructure` variable as follows:

```
pS := new(aStructure)
```

After executing the `new` statement, you are ready to work with your fresh variable that has its allocated memory zeroed but not initialized.



The main difference between `new` and `make` is that variables created with `make` are properly initialized without just zeroing the allocated memory space. Additionally, `make` can only be applied to maps, channels, and slices, and does not return a memory address, which means that `make` does not return a pointer.

The next statement will create a slice with `new` that points to `nil`:

```
sP := new([]aStructure)
```

Tuples

Strictly speaking, a **tuple** is a finite ordered list with multiple parts. The most important thing about tuples is that Go has no support for the tuple type, which means that Go does not officially care about tuples, despite the fact that it has support for certain uses of tuples.

One interesting thing here is that we have been using Go tuples in this book since [Chapter 1, *Go and the Operating System*](#), in statements such as the next one, where a function returns two values that you get in a single statement:

```
min, _ := strconv.ParseFloat(arguments[1], 64)
```

The name of the Go program that will illustrate Go tuples is `tuples.go` and it will be presented in three code segments. Please note that the presented code uses a function that returns three values as a tuple. You will learn more about functions in [Chapter 6, *What You Might Not Know About Go Packages and Go Functions*](#).

The first part of `tuples.go` is as follows:

```
package main

import (
    "fmt"
)

func retThree(x int) (int, int, int) {
    return 2 * x, x * x, -x
}
```

You can see the implementation of a function named `retThree()` that returns a tuple containing three integer values. This capability permits the function to return multiple values, without the need to group the various return values into a structure, and return a structure variable instead.

In [Chapter 6, *What You Might Not Know About Go Packages and Go Functions*](#), you will learn how to put names to the return values of a Go function, which is a very handy feature that can save you from various types of bugs.

The second part of `tuples.go` is as follows:

```
func main() {  
    fmt.Println(retThree(10))  
    n1, n2, n3 := retThree(20)  
    fmt.Println(n1, n2, n3)  
}
```

Here, we use the `retThree()` function twice. Firstly, we do this without saving its return values. Secondly, we do this by saving the three return values of `retThree()` into three different variables using a single statement, which in Go terminology is called a **tuple assignment**, hence the confusion about Go supporting tuples.

If you do not care about one or more return values of a function, you can put an **underscore character** (`_`) in their place. Note that it is a compile time error in Go if you declare a variable and do not use it afterward.

The third part of the program is shown in the following Go code:

```
    n1, n2 = n2, n1  
    fmt.Println(n1, n2, n3)  
  
    x1, x2, x3 := n1*2, n1*n1, -n1  
    fmt.Println(x1, x2, x3)  
}
```

As you can see, tuples can do many intelligent things, such as swapping values without the need for a temporary variable, as well as evaluating expressions.

Executing `tuples.go` will create the next output:

```
$ go run tuples.go  
20 100 -10  
40 400 -20  
400 40 -20  
800 160000 -400
```

Regular expressions and pattern matching

Pattern matching, which plays a key role in Go, is a technique for searching a string for some set of characters based on a specific search pattern that is based on regular expressions and **grammars**. If pattern matching is successful, it allows you to extract the desired data from the string, replace it, or delete it.

The Go package responsible for defining regular expressions and performing pattern matching is called `regexp`. You will see it in action later in this chapter.



When using a regular expression in your code, you should consider the definition of the regular expression as the most important part of the relevant code because the functionality of that code depends on the regular expression.

Introducing some theory

Every regular expression is compiled into a recognizer by building a generalized transition diagram called a **finite automaton**. A finite automaton can be either deterministic or nondeterministic. Nondeterministic means that more than one transition out of a state can be possible for the same input. A **recognizer** is a program that takes a string x as input and is able to tell whether x is a sentence of a given language.

A **grammar** is a set of production rules for strings in a formal language. The production rules describe how to create strings from the alphabet of the language that are valid according to the syntax of the language. A grammar does not describe the meaning of a string or what can be done with it in whatever context – it only describes its form. What is important here is to realize that grammars are at the heart of regular expressions because without a grammar, you cannot define or use a regular expression.



Although regular expressions allow you to solve problems that it would be extremely difficult to solve otherwise, do not try to solve every problem you face with a regular expression. Always use the right tool for the job.

The rest of this section will present three examples of regular expressions and pattern matching.

A simple example

In this subsection, you will learn how to select a particular column from a line of text. To make things more interesting, you will also learn how to read a text file line by line. However, file I/O is the subject of [Chapter 8, *Telling a UNIX System What to Do*](#), so you should refer to that chapter in order to get more information about the relevant Go code.

The name of the Go source file is `selectColumn.go`. It will be presented in five segments. The utility needs at least two command-line arguments to operate; the first one is the required column number and the second one is the path of the text file to process. However, you can use as many text files as you want – `selectColumn.go` will process all of them one by one.

The first part of `selectColumn.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)
```

The second code portion of `selectColumn.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: selectColumn column <file1> [<file2> [...<br><fileN>]]\n")
        os.Exit(1)
    }

    temp, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Column value is not an integer:", temp)
        return
    }

    column := temp
    if column < 0 {
        fmt.Println("Invalid Column number!")
        os.Exit(1)
    }
}
```

The first test the program performs is to make sure that it has an adequate number of command-line arguments (`len(arguments) < 2`). Additionally, you need two more tests to make sure that the provided column value is actually a number and that it is bigger than 0.

The third part of `selectColumn.go` follows:

```
for _, filename := range arguments[2:] {
    fmt.Println("\t\t", filename)
    f, err := os.Open(filename)
    if err != nil {
        fmt.Printf("error opening file %s\n", err)
        continue
    }
    defer f.Close()
```

The program performs various tests to make sure that the text file does exist and that you can read it – the `os.Open()` function is used for opening the text file. Remember that the UNIX file permissions of a text file might not allow you to read it.

The fourth code piece of `selectColumn.go` is as follows:

```
r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')

    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
    }
}
```

As you will learn in Chapter 8, *Telling a UNIX System What to Do*, the `bufio.ReadString()` function reads a file until the first occurrence of its parameter. As a result, `bufio.ReadString('\n')` tells Go to read a file line by line because `\n` is the UNIX newline character. The `bufio.ReadString()` function returns a **byte slice**.

The last code fragment of `selectColumn.go` is next:

```
    data := strings.Fields(line)
    if len(data) >= column {
        fmt.Println((data[column-1]))
    }
}
```

The logic behind the program is pretty simple: you split each line of text and select the desired column. However, as you cannot be sure that the current line has the required number of fields, you check that before printing any output. This is the simplest form of pattern matching because each line is split using space characters as word separators.

If you want more information about the splitting of lines, then you will find it useful to know that the `strings.Fields()` function splits a string based on the whitespace characters that are defined in the `unicode.IsSpace()` function and returns a slice of strings.

Executing `selectColumn.go` will generate the next kind of output:

```
$ go run selectColumn.go 15 /tmp/swtag.log /tmp/adobegc.log | head
/tmp/swtag.log
/tmp/adobegc.log
AdobeGCData
Successfully
Initializing
Stream
*****AdobeGC
Perform
Perform
Trying
```

The `selectColumn.go` utility prints the name of each processed file even if you get no output from that file.



The important thing to remember is that you should never trust your data, especially when it comes from non-technical users. Put simply, always verify that the data you expect to grab is there.

A more advanced example

In this section, you will learn how to match a date and time string as found in the log files of an Apache web server. To make things even more interesting, you will also learn how to change the date and time format of the log file into a different format. Once again, this requires reading the Apache log file, which is a plain text file, line by line.

The name of the command-line utility is `changedT.go` and it will be presented in five parts. Note that `changedT.go` is an improved version of the `timeDate.go` utility presented in Chapter 3, *Working with Basic Go Data Types*, not only because it gets its data from an external file, but also because `changedT.go` uses two regular expressions and therefore it is able to match strings in two different time and date formats.



There is a very important point here: do not try to implement every possible feature in the first version of your utilities. It is a better approach to build a working version with fewer features and improve that version in small steps.

The first chunk of code from `changeDT.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
    "regexp"
    "strings"
    "time"
)
```

Lots of packages are needed because `changeDT.go` does so many fascinating things.

The second piece of code from `changeDT.go` is the following:

```
func main() {

    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one text file to process!")
        os.Exit(1)
    }

    filename := arguments[1]
    f, err := os.Open(filename)
    if err != nil {
        fmt.Printf("error opening file %s", err)
        os.Exit(1)
    }
    defer f.Close()

    notAMatch := 0
    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
        }
    }
}
```

In this part, you just try to open your input file for reading in order to read it line by line. The `notAMatch` variable holds the number of lines in the input file that did not match any one of the two regular expressions of the program.

The third code segment of `changedT.go` comes with the following Go code:

```

    r1 :=
regexp.MustCompile(`.*\[(\d\d\/\w+\/\d\d\d\d:\d\d:\d\d:\d\d.*)\]`.*`)
    if r1.MatchString(line) {
        match := r1.FindStringSubmatch(line)
        d1, err := time.Parse("02/Jan/2006:15:04:05 -0700", match[1])
        if err == nil {
            newFormat := d1.Format(time.Stamp)
            fmt.Print(strings.Replace(line, match[1], newFormat, 1))
        } else {
            notAMatch++
        }
        continue
    }

```

Here you can see that if the first date and time format is not a match, the program will continue its execution. However, if it enters the `if` block, the `continue` statement will get executed, which means that it will skip the remaining code of the surrounding `for` loop. Thus, in the first supported format, the time and date string has the 21/Nov/2017:19:28:09 +0200 format.

The `regexp.MustCompile()` function is like `regexp.Compile()` but panics if the expression cannot be parsed. The parentheses around the regular expression allows you to use the matches afterwards. In this case, you can only have one match, which you will get using the `regexp.FindStringSubmatch()` function.

The fourth part of `changedT.go` follows:

```

    r2 := regexp.MustCompile(`.*\[(\w+\/-\d\d-\d\d:\d\d:\d\d:\d\d.*)\]`.*`)
    if r2.MatchString(line) {
        match := r2.FindStringSubmatch(line)
        d1, err := time.Parse("Jan-02-06:15:04:05 -0700", match[1])
        if err == nil {
            newFormat := d1.Format(time.Stamp)
            fmt.Print(strings.Replace(line, match[1], newFormat, 1))
        } else {
            notAMatch++
        }
        continue
    }

```

The second supported time and date format is `Jun-21-17:19:28:09 +0200`. As you can appreciate, there are not many differences between the two formats. Note that although the program uses just two date and time formats, you can have as many of these types of formats as you desire.

The last code portion from `changeDT.go` contains the following Go code:

```
    }
    fmt.Println(notAMatch, "lines did not match!")
}
```

Here, you print the number of lines that did not match any one of the two formats.

The text file that will be used for testing `changeDT.go` will contain the next lines:

```
$ cat logEntries.txt
- - [21/Nov/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249
"-
- - [21/Jun/2017:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [25/Lun/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [Jun-21-17:19:28:09 +0200] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [20/Nov/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
- - [35/Nov/2017:20:05:34 +0200] "GET /MongoDjango.zip HTTP/1.1" 200 118362
```

Executing `changDT.go` will generate the next output:

```
$ go run changeDT.go logEntries.txt
- - [Nov 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200 2188249 "-"
- - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [Jun 21 19:28:09] "GET /AMEv2.tif.zip HTTP/1.1" 200
- - [Nov 20 20:05:34] "GET /MongoDjango.zip HTTP/1.1" 200 118362
2 lines did not match!
```

Matching IPv4 addresses

An **IPv4 address**, or simply an **IP address**, has four discrete parts. As an IPv4 address is stored using 8-bit binary numbers, each part can have values from 0, which is 00000000 in the binary format, to 255, which is equal to 11111111 in the binary format.



The format of an IPv6 address is much more complicated than the format of an IPv4 address so the presented program will not work with IPv6 addresses.

The name of the program will be `findIPv4.go` and it is going to be presented in five parts. The first part of `findIPv4.go` is shown here:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "net"
    "os"
    "path/filepath"
    "regexp"
)
```

As `findIPv4.go` is a pretty sophisticated utility, it needs many standard Go packages.

The second part is shown in the following Go code:

```
func findIP(input string) string {
    partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])"
    grammar := partIP + "\\." + partIP + "\\." + partIP + "\\." +
    partIP
    matchMe := regexp.MustCompile(grammar)
    return matchMe.FindString(input)
}
```

The preceding code contains the definition of the regular expression that will help you to discover an IPv4 address inside a function. This is the most critical part of the program because if you define the regular expression incorrectly, you will never be able to catch any IPv4 addresses.

Before explaining the regular expression a little further, it is important to understand that prior to defining one or more regular expressions, you should be aware of the problem you are trying to solve. In other words, if you are not aware of the fact that the decimal values of an IPv4 address cannot be larger than 255, no regular expression can save you!

Now that we are on the same page, let us talk about the next two statements:

```
partIP := "(25[0-5]|2[0-4][0-9]|1[0-9][0-9]|[1-9]?[0-9])"
grammar := partIP + "\\." + partIP + "\\." + partIP + "\\." + partIP
```

The regular expression defined in `partIP` matches each one of the four parts of an IP address. A valid IPv4 address can begin with 25 and end with 0, 1, 2, 3, 4, or 5 because that is the biggest 8-bit binary number (`25[0-5]`), or it can begin with 2 followed by 0, 1, 2, 3, or 4 and end with 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9 (`2[0-4][0-9]`).

Alternatively, it can begin with 1 followed by two more digits from 0 to 9 (`1[0-9][0-9]`). The last alternative would be a natural number that has one or two digits. The first digit, which is optional, can be from 1 to 9 and the second, which is mandatory, can be from 0 to 9 (`[1-9]?[0-9]`).

The `grammar` variable tells us that what we are looking for has four distinct parts, and each one of them must match `partIP`. That `grammar` variable is what matches the complete IPv4 address that we seek.



As `findIPv4.go` works with regular expressions to find an IPv4 address in a file, it can process any kind of text file that contains valid IPv4 addresses.

Finally, if you have any special requirements, such as excluding certain IPv4 addresses or watching for specific addresses or networks, you can easily change the Go code of `findIPv4.go` and add the extra functionality you desire, which is the kind of flexibility achieved when you develop your own tools.

The third part of the `findIPv4.go` utility contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Printf("usage: %s logFile\n", filepath.Base(arguments[0]))
        os.Exit(1)
    }

    for _, filename := range arguments[1:] {
        f, err := os.Open(filename)
        if err != nil {

            fmt.Printf("error opening file %s\n", err)
            os.Exit(-1)
        }
        defer f.Close()
    }
}
```

Firstly, you make sure that you have a sufficient number of command-line arguments by checking the length of `os.Args`. Then, you use a `for` loop to iterate over all of the command-line arguments.

The fourth code portion is as follows:

```
r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
```

```
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
}
```

As happened in `selectColumn.go`, you use `bufio.ReadString()` to read your input line by line.

The last part of `findIPv4.go` contains the following Go code:

```
    ip := findIP(line)
    trial := net.ParseIP(ip)
    if trial.To4() == nil {
        continue
    }
    fmt.Println(ip)
}
}
```

For each line of the input text file, you call the `findIP()` function. The `net.ParseIP()` function double-checks that we are dealing with a valid IPv4 address – it is never a bad thing to double-check! If the call to `net.ParseIP()` is successful, you print the IPv4 address you just found. After that, the program will deal with the next line of input.

Executing `findIPv4.go` will generate the next kind of output:

```
$ go run findIPv4.go /tmp/auth.log
116.168.10.9
192.31.20.9
10.10.16.9
10.31.160.9
192.168.68.194
```

Thus, the output of `findIPv4.go` can have lines that are displayed multiple times. Apart from that detail, the output of the utility is pretty straightforward.

Processing the preceding output with some traditional UNIX command-line utilities might help you to reveal more information about your data:

```
$ go run findIPv4.go /tmp/auth.log.1 /tmp/auth.log | sort -rn | uniq -c |
sort -rn
 38 xxx.zz.116.9
 33 x.zz.2.190
 25 xx.zzz.1.41
```



```
20 178.132.1.18
18 x.zzz.63.53
17 178.zzz.y.9
15 103.yyy.xxx.179
10 213.z.yy.194
10 yyy.zzz.110.4
9 yy.xx.65.113
```

What we did here is find out the top-10 IPv4 addresses located in the processed text files using the `sort(1)` and `uniq(1)` UNIX command-line utilities. The logic behind this pretty long `bash(1)` shell command is simple: the output of the `findIPv4.go` utility will become the input of the first `sort -rn` command in order to be sorted numerically in reverse order. Then, the `uniq -c` command removes the lines that appear multiple times by replacing them with a single line that is preceded with the count of the number of times the line occurred in the input. The output is then sorted once again so that the IPv4 addresses with the higher number of occurrences will appear first.



Once again, it is important to realize that the core functionality of `findIPv4.go` is implemented through the regular expression. If the regular expression is defined incorrectly or does not match all the cases (**false negative**) or matches things that should not be matched (**false positive**) then your program will not work correctly.

Strings

Strictly speaking, a **string** in Go is not a composite type, but there are so many Go functions that support strings that I decided to describe strings in more detail in this chapter.

As discussed in Chapter 3, *Working with Basic Go Data Types*, strings in Go are value types, not pointers as is the case with **C strings**. Additionally, Go supports UTF-8 strings by default, which means that you do not need to load any special packages or do anything tricky in order to print Unicode characters. However, there are subtle differences between a character, a rune, and a byte, as well as differences between a string and a string literal, which are going to be clarified here.

A Go string is a read-only **byte slice** that can hold any type of bytes and can have an arbitrary length.

You can define a new **string literal** as follows:

```
const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
```

You might be surprised with the look of the string literal. You can define a string variable as follows:

```
s2 := "€£³"
```

You can find the length of a string variable or a string literal using the `len()` function.

The `strings.go` file will illustrate many standard operations related to strings and will be presented in five parts. The first is shown in the following Go code:

```
package main

import (
    "fmt"
)
```

The second portion of Go code is as follows:

```
func main() {
    const sLiteral = "\x99\x42\x32\x55\x50\x35\x23\x50\x29\x9c"
    fmt.Println(sLiteral)
    fmt.Printf("x: %x\n", sLiteral)

    fmt.Printf("sLiteral length: %d\n", len(sLiteral))
}
```

Each `\xAB` sequence represents a single character of `sLiteral`. As a result, calling `len(sLiteral)` will return the number of characters of `sLiteral`. Using `%x` in `fmt.Printf()` will return the AB part of a `\xAB` sequence.

The third code segment of `strings.go` is shown in the following Go code:

```
for i := 0; i < len(sLiteral); i++ {
    fmt.Printf("%x ", sLiteral[i])
}
fmt.Println()

fmt.Printf("q: %q\n", sLiteral)
fmt.Printf("+q: %+q\n", sLiteral)
fmt.Printf(" x: % x\n", sLiteral)

fmt.Printf("s: As a string: %s\n", sLiteral)
```

Here, you can see that you can access a string literal as if it is a slice. Using `%q` in `fmt.Printf()` with a string argument will print a double-quoted string that is safely escaped with Go syntax. Using `%+q` in `fmt.Printf()` with a string argument will guarantee ASCII-only output.

Last, using `% x` (note the space between the `%` character and the `x` character) in `fmt.Printf()` will put spaces between the printed bytes. In order to print a string literal as a string, you will need to call `fmt.Printf()` with `%s`.

The fourth code portion of `strings.go` is as follows:

```
s2 := "€³"
for x, y := range s2 {
    fmt.Printf("%#U starts at byte position %d\n", y, x)
}

fmt.Printf("s2 length: %d\n", len(s2))
```

Here you define a string named `s2` with three Unicode characters. Using `fmt.Printf()` with `%#U` will print the characters in the `U+0058` format. Using the `range` keyword on a string that contains Unicode characters will allow you to process its Unicode characters one by one.

The output of `len(s2)` might surprise you a little. As the `s2` variable contains Unicode characters, its byte size is larger than the number of characters in it.

The last part of `strings.go` is as follows:

```
const s3 = "ab12AB"
fmt.Println("s3:", s3)
fmt.Printf("x: % x\n", s3)

fmt.Printf("s3 length: %d\n", len(s3))

for i := 0; i < len(s3); i++ {
    fmt.Printf("%x ", s3[i])
}
fmt.Println()
}
```

Running `strings.go` will generate the next output:

```
$ go run strings.go
💎B2UP5#P)💎
x: 9942325550352350299c
sLiteral length: 10
99 42 32 55 50 35 23 50 29 9c
q: "\x99B2UP5#P)\x9c"
+q: "\x99B2UP5#P)\x9c"
x: 99 42 32 55 50 35 23 50 29 9c
s: As a string: 💎B2UP5#P)💎
U+20AC '€' starts at byte position 0
```

```
U+00A3 '£' starts at byte position 3
U+00B3 '³' starts at byte position 5
s2 length: 7
s3: ab12AB
x: 61 62 31 32 41 42
s3 length: 6
61 62 31 32 41 42
```

It will be no surprise if you find the information presented in this section pretty strange and complex, especially if you are not familiar with Unicode and UTF-8 representations of characters and symbols. The good thing is that you will not need most of them in your everyday life as a Go developer; you will most likely get away with using simple `fmt.Println()` and `fmt.Printf()` commands in your programs to print your output. However, if you are living outside of Europe and the U.S., you might find some of the information in this section pretty handy.

What is a rune?

A **rune** is an `int32` value, and therefore a Go type, that is used for representing a **Unicode code point**. A Unicode code point, or code position, is a numerical value that is usually used for representing single Unicode characters; however, it can also have alternative meanings, such as providing formatting information.



NOTE: You can consider a string as a collection of runes.

A **rune literal** is a character in single quotes. You may also consider a rune literal as a **rune constant**. Behind the scenes, a rune literal is associated with a Unicode code point.

Runes are going to be illustrated in `runes.go`, which is going to be presented in three parts. The first part of `runes.go` follows:

```
package main

import (
    "fmt"
)
```

The second part of `runes.go` contains the following code:

```
func main() {
    const r1 = '€'
```

```
fmt.Println("(int32) r1:", r1)
fmt.Printf("(HEX) r1: %x\n", r1)
fmt.Printf("(as a String) r1: %s\n", r1)
fmt.Printf("(as a character) r1: %c\n", r1)
```

First you define a rune literal named `r1`. (Please note that the Euro sign does not belong to the ASCII table of characters.) Then, you print `r1` using various statements. Next, you print its `int32` value and its hexadecimal value. After that, you try printing it as a string. Finally, you print it as a character, which is what gives you the same output as the one used in the definition of `r1`.

The third and last code segment of `runes.go` is shown in the following Go code:

```
fmt.Println("A string is a collection of runes:", []byte("Mihalis"))
aString := []byte("Mihalis")
for x, y := range aString {
    fmt.Println(x, y)
    fmt.Printf("Char: %c\n", aString[x])
}
fmt.Printf("%s\n", aString)
}
```

Here, you see that a byte slice is a collection of runes and that printing a byte slice with `fmt.Println()` might not return what you expected. In order to convert a rune into a character, you should use `%c` in a `fmt.Printf()` statement. In order to print a byte slice as a string, you will need to use `fmt.Printf()` with `%s`.

Executing `runes.go` will create the following output:

```
$ go run runes.go
(int32) r1: 8364
(HEX) r1: 20ac
(as a String) r1: %!s(int32=8364)
(as a character) r1: €
A string is a collection of runes: [77 105 104 97 108 105 115]
0 77
Char: M
1 105
Char: i
2 104
Char: h
3 97
Char: a
4 108
Char: l
5 105
Char: i
```

```
6 115
Char: s
Mihalis
```

Finally, the easiest way to get an `illegal rune literal` error message is by using single quotes instead of double quotes when importing a package:

```
$ cat a.go
package main
import (
    'fmt'
)
func main() {
}
$ go run a.go
package main:
a.go:4:2: illegal rune literal
```

The unicode package

The `unicode` standard Go package contains various handy functions. One of them, which is called `unicode.IsPrint()`, can help you to identify the parts of a string that are printable using runes. This technique will be illustrated in the Go code of `unicode.go`, which will be presented in two parts. The first part of `unicode.go` is as follows:

```
package main

import (
    "fmt"
    "unicode"
)

func main() {
    const sL = "\x99\x00ab\x50\x00\x23\x50\x29\x9c"
```

The second code segment of `unicode.go` is shown in the following Go code:

```
    for i := 0; i < len(sL); i++ {
        if unicode.IsPrint(rune(sL[i])) {
            fmt.Printf("%c\n", sL[i])
        } else {
            fmt.Println("Not printable!")
        }
    }
}
```

As stated before, all of the dirty work is done by the `unicode.IsPrint()` function, which returns `true` when a rune is printable and `false` otherwise. If you are really into Unicode characters, you should definitely check the documentation page of the `unicode` package. Executing `unicode.go` will generate the following output:

```
$ go run unicode.go
Not printable!
Not printable!
a
b
P
Not printable!
#
P
)
Not printable!
```

The strings package

The `strings` standard Go package allows you to manipulate UTF-8 strings in Go and includes many powerful functions. Most of these functions will be illustrated in the `useStrings.go` source file, which will be presented in five parts. Note that the functions of the `strings` package that are related to file input and output will be demonstrated in Chapter 8, *Telling a UNIX System What to Do*.

The first part of `useStrings.go` follows:

```
package main

import (
    "fmt"
    s "strings"
    "unicode"
)

var f = fmt.Printf
```



There is a difference in the way the `strings` package is imported. This kind of import statement makes Go create an alias for that package. So, instead of writing `strings.FunctionName()`, you can now write `s.FunctionName()`, which is a bit shorter. Please note that you will not be able to call a function of the `strings` package as `strings.FunctionName()` anymore.

Another handy trick is that if you find yourself using the same function all the time, and you want to use something shorter instead, you can assign a variable name to that function and use that variable name instead. Here, you can see that feature applied to the `fmt.Printf()` function. Nevertheless, you should not overuse that feature because you might end up finding it difficult to read the code!

The second part of `useStrings.go` contains the following Go code:

```
func main() {
    upper := s.ToUpper("Hello there!")
    f("To Upper: %s\n", upper)
    f("To Lower: %s\n", s.ToLower("Hello THERE"))
    f("%s\n", s.Title("tHis wiLL be A title!"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALis"))
    f("EqualFold: %v\n", s.EqualFold("Mihalis", "MIHALi"))
}
```

In this code segment, you can see many functions that allow you to play with the case of a string. Additionally, you can see that the `strings.EqualFold()` function allows you to determine whether two strings are the same in spite of the differences in their letters.

The third code portion of `useStrings.go` follows:

```
f("Prefix: %v\n", s.HasPrefix("Mihalis", "Mi"))
f("Prefix: %v\n", s.HasPrefix("Mihalis", "mi"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "is"))
f("Suffix: %v\n", s.HasSuffix("Mihalis", "IS"))

f("Index: %v\n", s.Index("Mihalis", "ha"))
f("Index: %v\n", s.Index("Mihalis", "Ha"))
f("Count: %v\n", s.Count("Mihalis", "i"))
f("Count: %v\n", s.Count("Mihalis", "I"))
f("Repeat: %s\n", s.Repeat("ab", 5))

f("TrimSpace: %s\n", s.TrimSpace(" \tThis is a line. \n"))
f("TrimLeft: %s", s.TrimLeft(" \tThis is a\t line. \n", "\n\t"))
f("TrimRight: %s\n", s.TrimRight(" \tThis is a\t line. \n",
"\n\t "))
```

The `strings.Count()` function counts the number of non-overlapping times the second parameter appears in the string that is given as the first parameter. The `strings.HasPrefix()` function returns `true` when the first parameter string begins with the second parameter string, and `false` otherwise. Similarly, the `strings.HasSuffix()` function returns `true` when the first parameter, which is a string, ends with the second parameter, which is also a string, and `false` otherwise.

The fourth code segment of `useStrings.go` contains the following Go code:

```
f("Compare: %v\n", s.Compare("Mihalis", "MIHALIS"))
f("Compare: %v\n", s.Compare("Mihalis", "Mihalis"))
f("Compare: %v\n", s.Compare("MIHALIS", "MIHalis"))

f("Fields: %v\n", s.Fields("This is a string!"))
f("Fields: %v\n", s.Fields("Thisis\na\tstring!"))

f("%s\n", s.Split("abcd efg", ""))
```

This code portion contains some pretty advanced and ingenious functions. The first handy function is `strings.Split()`, which allows you to split the given string according to the desired separator string – the `strings.Split()` function returns a **string slice**. Using "" as the second parameter of `strings.Split()` will allow you to process a string character by character.

The `strings.Compare()` function compares two strings lexicographically, and it may return 0 if the two strings are identical, and -1 or +1 otherwise.

Lastly, the `strings.Fields()` function splits the string parameter using whitespace characters as separators. The whitespace characters are defined in the `unicode.IsSpace()` function.



`strings.Split()` is a powerful function that you should learn because sooner rather than later, you will have to use it in your programs.

The last part of `useStrings.go` is shown in the following Go code:

```
f("%s\n", s.Replace("abcd efg", "", "_", -1))
f("%s\n", s.Replace("abcd efg", "", "_", 4))
f("%s\n", s.Replace("abcd efg", "", "_", 2))

lines := []string{"Line 1", "Line 2", "Line 3"}
f("Join: %s\n", s.Join(lines, "+++"))

f("SplitAfter: %s\n", s.SplitAfter("123++432++", "+++"))

trimFunction := func(c rune) bool {
    return !unicode.IsLetter(c)
}
f("TrimFunc: %s\n", s.TrimFunc("123 abc ABC \t .",
trimFunction))
}
```

As in the previous part of `useStrings.go`, the last code segment contains functions that implement some very intelligent functionality in a simple-to-understand and easy-to-use way.

The `strings.Replace()` function takes four parameters. The first parameter is the string that you want to process. The second parameter contains the string that, if found, will be replaced by the third parameter of `strings.Replace()`. The last parameter is the maximum number of replacements that are allowed to happen. If that parameter has a negative value, then there is no limit to the number of replacements that can take place.

The last two statements of the program define a **trim function**, which allows you to keep the runes of a string that interest you and utilize that function as the second argument to the `strings.TrimFunc()` function.

Lastly, the `strings.SplitAfter()` function splits its first parameter string into substrings based on the separator string that is given as the second parameter to the function.



For the full list of functions similar to `unicode.IsLetter()`, you should visit the documentation page of the `unicode` standard Go package.

Executing `useStrings.go` will create the next output:

```
$ go run useStrings.go
To Upper: HELLO THERE!
To Lower: hello there
THis WiLL Be A Title!
EqualFold: true
EqualFold: false
Prefix: true
Prefix: false
Suffix: true
Suffix: false
Index: 2
Index: -1
Count: 2
Count: 0
Repeat: ababababab
TrimSpace: This is a line.
TrimLeft: This is a line.
TrimRight: This is a line.
Compare: 1
Compare: 0
Compare: -1
Fields: [This is a string!]
```

```

Fields: [This is a string!]
[a b c d   e f g]
_a_b_c_d_ _e_f_g_
_a_b_c_d_ efg
_a_bcd efg
Join: Line 1+++Line 2+++Line 3
SplitAfter: [123++ 432++ ]
TrimFunc: abc ABC

```

Please note that the list of functions presented from the `strings` package is far from complete. You should see the documentation page of the `strings` package at <https://golang.org/pkg/strings/> for the complete list of available functions.

If you are working with text and text processing, you will definitely need to learn all the gory details and functions of the `strings` package, so make sure that you experiment with all these functions and create many examples that will help you to clarify things.

The switch statement

The main reason for presenting the `switch` statement in this chapter is because a `switch` case can use regular expressions. Firstly, however, take a look at this simple `switch` block:

```

switch asString {
case "1":
    fmt.Println("One!")
case "0":
    fmt.Println("Zero!")
default:
    fmt.Println("Do not care!")
}

```

The preceding `switch` block can differentiate between the "1" string, the "0" string, and everything else (default).



Having a *match all remaining cases* case in a `switch` block is considered a very good practice. However, as the order of the cases in a `switch` block does matter, the *match all remaining cases* case should be put last. In Go, the name of the *match all remaining cases* case is `default`.

However, a `switch` statement can be much more flexible and adaptable:

```

switch {
case number < 0:
    fmt.Println("Less than zero!")
}

```

```
case number > 0:
    fmt.Println("Bigger than zero!")
default:
    fmt.Println("Zero!")
}
```

The preceding `switch` block does the job of identifying whether you are dealing with a positive integer, a negative integer, or zero. As you can see, the branches of a `switch` statement can have conditions. You will soon see that the branches of a `switch` statement can also have regular expressions in them.

All these examples, and some additional ones, can be found in `switch.go`, which will be presented in five parts.

The first part of `switch.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "regexp"
    "strconv"
)

func main() {

    arguments := os.Args
    if len(arguments) < 2 {
        fmt.Println("usage: switch number")
        os.Exit(1)
    }
```

The `regexp` package is needed for supporting regular expressions in `switch`.

The second code segment of `switch.go` is shown in the following Go code:

```
number, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println("This value is not an integer:", number)
} else {
    switch {
    case number < 0:
        fmt.Println("Less than zero!")
    case number > 0:
        fmt.Println("Bigger than zero!")
    default:
```

```
        fmt.Println("Zero!")
    }
}
```

The third part of `switch.go` is as follows:

```
asString := arguments[1]
switch asString {
case "5":
    fmt.Println("Five!")
case "0":
    fmt.Println("Zero!")
default:
    fmt.Println("Do not care!")
}
```

In this code segment, you can see that a `switch` case can also contain hardcoded values. This mainly occurs when the `switch` keyword is followed by the name of a variable.

The fourth code portion of `switch.go` contains the following Go code:

```
var negative = regexp.MustCompile(`-`)
var floatingPoint = regexp.MustCompile(`\d?\.\d`)
var email = regexp.MustCompile(`^[^@]+@[^@.]+\.[^@.]+\.`)

switch {
case negative.MatchString(asString):
    fmt.Println("Negative number")
case floatingPoint.MatchString(asString):
    fmt.Println("Floating point!")
case email.MatchString(asString):
    fmt.Println("It is an email!")
    fallthrough
default:
    fmt.Println("Something else!")
}
```

Many interesting things are happening here. Firstly, you define three regular expressions named `negative`, `floatingPoint`, and `email`, respectively. Secondly, you use all three of them in the `switch` block with the help of the `regexp.MatchString()` function, which does the actual matching.

Lastly, the `fallthrough` keyword tells Go to execute the branch that follows the current one, which, in this case, is the `default` branch. This means that when the code of the `email.MatchString(asString)` case is the one that will get executed, the `default` case will also be executed.

The last part of `switch.go` is as follows:

```
var aType error = nil
switch aType.(type) {
case nil:
    fmt.Println("It is nil interface!")
default:
    fmt.Println("Not nil interface!")
}
```

Here, you can see that `switch` can differentiate between types. You will learn more about working with `switch` and Go **interfaces** in Chapter 7, *Reflection and Interfaces for All Seasons*.

Executing `switch.go` with various input arguments will generate the next kind of output:

```
$ go run switch.go
usage: switch number.
exit status 1
$ go run switch.go mike@g.com
This value is not an integer: 0
Do not care!
It is an email!
Something else!
It is nil interface!
$ go run switch.go 5
Bigger than zero!
Five!
Something else!
It is nil interface!
$ go run switch.go 0
Zero!
Zero!
Something else!
It is nil interface!
$ go run switch.go 1.2
This value is not an integer: 0
Do not care!
Floating point!
It is nil interface!
$ go run switch.go -1.5
This value is not an integer: 0
Do not care!
Negative number
It is nil interface!
```

Calculating Pi with high accuracy

In this section, you will learn how to calculate **Pi** with high accuracy using a standard Go package named `math/big` and the special purpose types offered by that package.



This section contains the ugliest Go code that I have even seen; even **Java** code looks better than this!

The name of the program that uses Bellard's formula to calculate Pi is `calculatePi.go` and it will be presented in four parts.

The first part of `calculatePi.go` follows:

```
package main

import (
    "fmt"
    "math"
    "math/big"
    "os"
    "strconv"
)

var precision uint = 0
```

The precision variable holds the desired precision of the calculations, and it is made global in order to be accessible from everywhere in the program.

The second code segment of `calculatePi.go` is shown in the following Go code:

```
func Pi(accuracy uint) *big.Float {
    k := 0
    pi := new(big.Float).SetPrec(precision).SetFloat64(0)
    k1k2k3 := new(big.Float).SetPrec(precision).SetFloat64(0)
    k4k5k6 := new(big.Float).SetPrec(precision).SetFloat64(0)
    temp := new(big.Float).SetPrec(precision).SetFloat64(0)
    minusOne := new(big.Float).SetPrec(precision).SetFloat64(-1)
    total := new(big.Float).SetPrec(precision).SetFloat64(0)

    two2Six := math.Pow(2, 6)
    two2SixBig := new(big.Float).SetPrec(precision).SetFloat64(two2Six)
```

The `new(big.Float)` call creates a new `big.Float` variable with the required precision, which is set by `SetPrec()`.

The third part of `calculatePi.go` contains the remaining Go code of the `Pi()` function:

```

    for {
        if k > int(accuracy) {
            break
        }
        t1 := float64(float64(1) / float64(10*k+9))
        k1 := new(big.Float).SetPrec(precision).SetFloat64(t1)
        t2 := float64(float64(64) / float64(10*k+3))
        k2 := new(big.Float).SetPrec(precision).SetFloat64(t2)
        t3 := float64(float64(32) / float64(4*k+1))
        k3 := new(big.Float).SetPrec(precision).SetFloat64(t3)
        k1k2k3.Sub(k1, k2)
        k1k2k3.Sub(k1k2k3, k3)

        t4 := float64(float64(4) / float64(10*k+5))
        k4 := new(big.Float).SetPrec(precision).SetFloat64(t4)
        t5 := float64(float64(4) / float64(10*k+7))
        k5 := new(big.Float).SetPrec(precision).SetFloat64(t5)
        t6 := float64(float64(1) / float64(4*k+3))
        k6 := new(big.Float).SetPrec(precision).SetFloat64(t6)
        k4k5k6.Add(k4, k5)
        k4k5k6.Add(k4k5k6, k6)
        k4k5k6 = k4k5k6.Mul(k4k5k6, minusOne)
        temp.Add(k1k2k3, k4k5k6)

        k7temp := new(big.Int).Exp(big.NewInt(-1), big.NewInt(int64(k)),
nil)
        k8temp := new(big.Int).Exp(big.NewInt(1024), big.NewInt(int64(k)),
nil)

        k7 := new(big.Float).SetPrec(precision).SetFloat64(0)
        k7.SetInt(k7temp)
        k8 := new(big.Float).SetPrec(precision).SetFloat64(0)
        k8.SetInt(k8temp)

        t9 := float64(256) / float64(10*k+1)
        k9 := new(big.Float).SetPrec(precision).SetFloat64(t9)
        k9.Add(k9, temp)
        total.Mul(k9, k7)
        total.Quo(total, k8)
        pi.Add(pi, total)

        k = k + 1
    }
    pi.Quo(pi, twoSixBig)
    return pi
}

```


This part of the program is the Go implementation of Bellard's formula. The bad thing with `math/big` is that you need a special function of it for almost every kind of calculation, which mainly happens because those functions can keep the precision at the desired level. So, without using `big.Float` and `big.Int` variables, as well as the functions of `math/big` all the time, you cannot compute Pi with the desired precision.

The last part of `calculatePi.go` shows the implementation of the `main()` function:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide one numeric argument!")
        os.Exit(1)
    }

    temp, _ := strconv.ParseUint(arguments[1], 10, 32)
    precision = uint(temp) * 3

    PI := Pi(precision)
    fmt.Println(PI)
}
```

Executing `calculatePi.go` will generate the next kind of output:

```
$ go run calculatePi.go
Please provide one numeric argument!
exit status 1
$ go run calculatePi.go 20
3.141592653589793258
$ go run calculatePi.go 200
3.1415926535897932569603993617387624040191831562485732434931792835710464502
489134671185117843176153542820179294162928090508139378752834356105863133635
48602436768047706489838924381929
```

Developing a key-value store in Go

In this section, you will learn how to develop an unsophisticated version of a **key-value store** in Go, which means that you will learn how to implement the core functionality of a key-value store without any additional bells and whistles. The idea behind a key-value store is modest: answer queries fast and work as fast as possible. This translates into using simple algorithms and simple data structures.

The presented program will implement the four fundamental tasks of a key-value store:

1. Adding a new element
2. Deleting an existing element from the key-value store based on a key
3. Looking up the value of a specific key in the store
4. Changing the value of an existing key

These four functions allow you to have full control over the key-value store. The commands for these four functions will be named `ADD`, `DELETE`, `LOOKUP`, and `CHANGE`, respectively. This means that the program will only operate when it gets one of these four commands. Additionally, the program will stop when you enter the `STOP` word as input and will print the full contents of the key-value store when you enter the `PRINT` command.

The name of the program will be `keyValue.go` and it will be presented in five code segments.

The first code segment of `keyValue.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "strings"
)

type myElement struct {
    Name      string
    Surname   string
    Id         string
}

var DATA = make(map[string]myElement)
```

The key-value store is stored in a native Go map because using a built-in Go structure is usually faster. The map variable is defined as a global variable, where its keys are `string` variables and its values are `myElement` variables. You can also see the definition of the `myElement struct` type [here](#).

The second code segment of `keyValue.go` is as follows:

```
func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }
```

```
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}
```

This code contains the implementation of two functions that support the functionality of the `ADD` and `DELETE` commands. Note that if the user tries to add a new element to the store without giving enough values to populate the `myElement` struct, the `ADD` function will fail. For this particular program, the missing fields of the `myElement` struct will be set to the empty string. However, if you try to add a key that already exists, you will get an error message instead of modifying the value of the existing key.

The third portion of `keyValue.go` contains the following code:

```
func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}
```

In this Go code segment, you can see the implementation of the functions that support the functionality of the `LOOKUP` and `CHANGE` commands. If you try to change a key that does not exist, the program will add that key to the store without generating any error messages. In this part, you can also see the implementation of the `PRINT()` function that prints the full contents of the key-value store.

The reason for using ALL CAPS for the names of these functions is that they are really important for the program.

The fourth part of `keyValue.go` is as follows:

```
func main() {
    scanner := bufio.NewScanner(os.Stdin)
    for scanner.Scan() {
        text := scanner.Text()
        text = strings.TrimSpace(text)
        tokens := strings.Fields(text)

        switch len(tokens) {
        case 0:
            continue
        case 1:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 2:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 3:
            tokens = append(tokens, "")
            tokens = append(tokens, "")
        case 4:
            tokens = append(tokens, "")
        }
    }
}
```

In this part of `keyValue.go`, you read the input from the user. Firstly, the `for` loop makes sure that the program will keep running for as long as the user provides some input. Secondly, the program makes sure that the `tokens` slice has at least five elements, even though only the `ADD` command needs that number of elements. Thus, for an `ADD` operation to be complete and not have any missing values, you will need an input that looks like `ADD aKey Field1 Field2 Field3`.

The last part of `keyValue.go` is shown in the following Go code:

```
switch tokens[0] {
case "PRINT":
    PRINT()
case "STOP":
    return
case "DELETE":
    if !DELETE(tokens[1]) {
        fmt.Println("Delete operation failed!")
    }
case "ADD":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !ADD(tokens[1], n) {
        fmt.Println("Add operation failed!")
    }
case "LOOKUP":
    n := LOOKUP(tokens[1])
    if n != nil {
        fmt.Printf("%v\n", *n)
    }
case "CHANGE":
    n := myElement{tokens[2], tokens[3], tokens[4]}
    if !CHANGE(tokens[1], n) {
        fmt.Println("Update operation failed!")
    }
default:
    fmt.Println("Unknown command - please try again!")
}
```

In this part of the program, you process the input from the user. The `switch` statement makes the design of the program very clean and saves you from having to use multiple `if...else` statements.

Executing and using `keyValue.go` will create the following output:

```
$ go run keyValue.go
UNKNOWN
Unknown command - please try again!
ADD 123 1 2 3
ADD 234 2 3 4
ADD 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: { }
```

```
ADD 345 3 4 5
Add operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {  }
CHANGE 345 3 4 5
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
DELETE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
DELETE 345
Delete operation failed!
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
ADD 345 3 4 5
ADD 567 -5 -6 -7
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {3 4 5}
key: 567 value: {-5 -6 -7}
CHANGE 345
PRINT
key: 123 value: {1 2 3}
key: 234 value: {2 3 4}
key: 345 value: {  }
key: 567 value: {-5 -6 -7}
STOP
```

You will have to wait until Chapter 8, *Telling a UNIX System What to Do*, in order to learn how to add data persistence to the key-value store.

You can also improve `keyValue.go` by adding goroutines and channels to it. However, adding goroutines and channels to a single-user application has no practical purpose. But if you make `keyValue.go` able to operate over **Transmission Control Protocol/Internet Protocol (TCP/IP)** networks, then the use of goroutines and channels will allow it to accept multiple connections and serve multiple users.

You will learn more about routines and channels in Chapter 9, *Concurrency in Go – Goroutines, Channels, and Pipelines*, and in Chapter 10, *Concurrency in Go – Advanced Topics*.

You will then learn about creating network applications in Go in Chapter 12, *The Foundations of Network Programming in Go*, and in Chapter 13, *Network Programming – Building Your Own Servers and Clients*.

Go and the JSON format

JSON is a very popular text-based format designed to be an easy and light way to pass information between JavaScript systems. However, JSON is also being used for creating configuration files for applications and storing data in a structured format.

The `encoding/json` package offers the `Encode()` and `Decode()` functions, which allow the conversion of a Go object into a JSON document and vice versa. Additionally, the `encoding/json` package offers the `Marshal()` and `Unmarshal()` functions, which work similarly to `Encode()` and `Decode()` and are based on the `Encode()` and `Decode()` methods. The main difference between the `Marshal()` and `Unmarshal()` pair and the `Encode()` and `Decode()` pair is that the former pair works on single objects, whereas the latter pair of functions can work on multiple objects as well as streams of bytes.

Reading JSON data

In this section, you will learn how to read a JSON record from disk using the code of `readJSON.go`, which will be presented in three parts.

The first part of `readJSON.go` is shown in the following Go code:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

```
}
```

In this Go code, we define the structure variables that are going to keep the JSON data.

The second part of `readJSON.go` is as follows:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

Here, we define a new function named `loadFromJSON()` that is used for decoding the data of a JSON file according to a data structure that is given as the second argument to it. We first call the `json.NewDecoder()` function to create a new JSON decode variable that is associated with a file, and then we call the `Decode()` function for actually decoding the contents of the file and putting them into the desired variable.

The last part of `readJSON.go` is the following:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println(myRecord)
    } else {
        fmt.Println(err)
    }
}
```


The contents of `readMe.json` are the following:

```
$ cat readMe.json
{
  "Name": "Mihalis",
  "Surname": "Tsoukalos",
  "Tel": [
    {"Mobile": true, "Number": "1234-567"},
    {"Mobile": true, "Number": "1234-abcd"},
    {"Mobile": false, "Number": "abcc-567"}
  ]
}
```

Executing `readJSON.go` will generate the following output:

```
$ go run readJSON.go readMe.json
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```

Saving JSON data

In this subsection, you will learn how to write JSON data. The utility that will be presented in three parts is called `writeJSON.go` and it will write to standard output (`os.Stdout`), which means that it will write on the terminal screen.

The first part of `writeJSON.go` is as follows:

```
package main

import (
    "encoding/json"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

The second part of `writeJSON.go` is the following:

```
func saveToJSON(filename *os.File, key interface{}) {
    encodeJSON := json.NewEncoder(filename)
    err := encodeJSON.Encode(key)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `saveToJSON()` function does all the work for us as it creates a JSON encoder variable named `encodeJSON`, which is associated with a `filename`, which is where the data is going to be put. The call to `Encode()` is what saves the data into the desired file after encoding it.

The last part of `writeJSON.go` is as follows:

```
func main() {
    myRecord := Record{
        Name:      "Mihalis",
        Surname:    "Tsoukalos",
        Tel: []Telephone{Telephone{Mobile: true, Number: "1234-567"},
                    Telephone{Mobile: true, Number: "1234-abcd"},
                    Telephone{Mobile: false, Number: "abcc-567"},
                },
    }

    saveToJSON(os.Stdout, myRecord)
}
```

The previous code is all about defining a structure variable that holds the data that we want to save in the JSON format using the `saveToJSON()` function. As we are using `os.Stdout`, the data will be printed on the screen instead of being saved into a file.

Executing `writeJSON.go` will generate the following output:

```
$ go run writeJSON.go
{"Name": "Mihalis", "Surname": "Tsoukalos", "Tel": [{"Mobile": true, "Number": "1234-567"}, {"Mobile": true, "Number": "1234-abcd"}, {"Mobile": false, "Number": "abcc-567"}]}
```

Using Marshal() and Unmarshal()

In this subsection, you will see how to use the `Marshal()` and `Unmarshal()` methods in order to implement the functionality of `readJSON.go` and `writeJSON.go`. The Go code that illustrates the `Marshal()` and `Unmarshal()` functions can be found in `mUJSON.go`, and it will be presented in three parts.

The first part of `mUJSON.go` is as follows:

```
package main

import (
    "encoding/json"
    "fmt"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

In this part of the program, we are defining two structures named `Record` and `Telephone`, which will be used for storing the data that will be put into a JSON record.

The second part of `mUJSON.go` is as follows:

```
func main() {
    myRecord := Record{
        Name:      "Mihalis",
        Surname:   "Tsoukalos",
        Tel: []Telephone{Telephone{Mobile: true, Number: "1234-567"},
                    Telephone{Mobile: true, Number: "1234-abcd"},
                    Telephone{Mobile: false, Number: "abcc-567"},
                }}

    rec, err := json.Marshal(&myRecord)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(string(rec))
}
```

In this part of the utility, we define the `myRecord` variable, which holds the desired data. You can also see the use of the `json.Marshal()` function, which accepts a reference to the `myRecord` variable. Note that `json.Marshal()` requires a pointer variable that converts into the JSON format.

The last part of `mUJSON.go` contains the following code:

```
var unRec Record
err1 := json.Unmarshal(rec, &unRec)
if err1 != nil {
    fmt.Println(err1)
    return
}
fmt.Println(unRec)
}
```

The `json.Unmarshal()` function gets JSON input and converts it into a Go structure. As it happened with `json.Marshal()`, `json.Unmarshal()` also requires a pointer argument.

Executing `mUJSON.go` will generate the following output:

```
$ go run mUJSON.go
{"Name": "Mihalis", "Surname": "Tsoukalos", "Tel": [{"Mobile": true, "Number": "1234-567"}, {"Mobile": true, "Number": "1234-abcd"}, {"Mobile": false, "Number": "abcc-567"}]}
{Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false abcc-567}]}
```



The `encoding/json` Go package includes two interfaces named `Marshaler` and `Unmarshaler`. Each one of these interfaces requires the implementation of a single method, named `MarshalJSON()` and `UnmarshalJSON()`, respectively. Should you wish to perform any custom JSON marshalling and unmarshalling, these two interfaces will allow you to do so.

Parsing JSON data

So far, we have seen how to process structured JSON data with a format that is known in advance. This kind of data can be stored in Go structures using the methods that are already described in the previous subsections.

This subsection will tell you how to read and store **unstructured JSON data**. The critical thing to remember is that unstructured JSON data is put into Go maps instead of Go structures – this will be illustrated in `parsingJSON.go`, which will be presented in four parts.

The first part of `parsingJSON.go` is as follows:

```
package main

import (
    "encoding/json"
    "fmt"
    "io/ioutil"
    "os"
)
```

In this part, we just import the required Go packages.

The second part of `parsingJSON.go` is the following:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]
```

The presented code reads the command-line arguments of the program and gets the first one, which is the JSON file that is going to be read.

The third part of `parsingJSON.go` is as follows:

```
    fileData, err := ioutil.ReadFile(filename)
    if err != nil {
        fmt.Println(err)
        return
    }

    var parsedData map[string]interface{}
    json.Unmarshal([]byte(fileData), &parsedData)
```

The `ioutil.ReadFile()` function allows you to read a file all at once, which is what we want here.

In this part, there is also a definition of a map named `parsedData` that will hold the contents of the JSON file that was read. Each map key, which is a `string`, corresponds to a JSON property. The value of each map key is of the type `interface{}`, which can be of any type – this means that the value of a map key can also be another map.

The `json.Unmarshal()` function is used for putting the contents of the file into the `parsedData` map.



In Chapter 7, *Reflection and Interfaces for All Seasons*, you will learn more about interfaces and `interface{}`, as well as **reflection**, which allows you to dynamically learn the type of an arbitrary object, as well as information about its structure.

The last part of `parsingJSON.go` contains the following code:

```
for key, value := range parsedData {  
    fmt.Println("key:", key, "value:", value)  
}  
}
```

The presented code shows that you can iterate over the map and get its contents. However, interpreting these contents is a totally different story because this depends on the structure of the data, which is not known.

The JSON file with the sample data that will be used in this subsection is called `noStr.json` and it has the following contents:

```
$ cat noStr.json  
{  
  "Name": "John",  
  "Surname": "Doe",  
  "Age": 25,  
  "Parents": [  
    "Jim",  
    "Mary"  
  ],  
  "Tel": [  
    {"Mobile": true, "Number": "1234-567"},  
    {"Mobile": true, "Number": "1234-abcd"},  
    {"Mobile": false, "Number": "abcc-567"}  
  ]  
}
```

Executing `parsingJSON.go` will generate the following output:

```
$ go run parsingJSON.go noStr.json
key: Tel value: [map[Mobile:true Number:1234-567] map[Mobile:true
Number:1234-abcd] map[Mobile:false Number:abcc-567]]
key: Name value: John
key: Surname value: Doe
key: Age value: 25
key: Parents value: [Jim Mary]
```

Once again, you can see from the output that map keys are printed in random order.

Go and XML

Go has support for **XML**, which is a markup language similar to HTML but much more advanced than HTML.

The developed utility, which is called `rwXML.go`, will read a JSON record from disk, make a change to it, convert it to XML, and print it on screen. Then it will convert the XML data into JSON. The related Go code will be presented in four parts.

The first part of `rwXML.go` is as follows:

```
package main

import (
    "encoding/json"
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

The second part of `rwXML.go` is shown in the following Go code:

```
func loadFromJSON(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeJSON := json.NewDecoder(in)
    err = decodeJSON.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

The third part of `rwXML.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromJSON(filename, &myRecord)
    if err == nil {
        fmt.Println("JSON:", myRecord)
    } else {
        fmt.Println(err)
    }

    myRecord.Name = "Dimitris"

    xmlData, _ := xml.MarshalIndent(myRecord, "", " ")
    xmlData = []byte(xml.Header + string(xmlData))
    fmt.Println("\nxmlData:", string(xmlData))
}
```

After we read the input file and convert it into JSON, we put its data into a Go structure. Then, we make a change to the data of that structure (`myRecord`). After that, we convert that data into the XML format using the `MarshalIndent()` function and add a header using `xml.Header`.

The `MarshalIndent()` function, which can also be used with JSON data, works like `Marshal()`, but each XML element begins with a new line and is indented according to its nesting depth. This mainly has to do with the presentation of the XML data, not the values.

The last part of `rwXML.go` is the following:

```
data := &Record{}
err = xml.Unmarshal(xmlData, data)
if nil != err {
    fmt.Println("Unmarshalling from XML", err)
    return
}

result, err := json.Marshal(data)
if nil != err {
    fmt.Println("Error marshalling to JSON", err)
    return
}

_ = json.Unmarshal([]byte(result), &myRecord)
fmt.Println("\nJSON:", myRecord)
}
```

In this part of the program, we convert the XML data into JSON using `Marshal()` and `Unmarshal()` and print it on screen.

Executing `rwXML.go` will generate the following output:

```
$ go run rwXML.go readMe.json
JSON: {Mihalis Tsoukalos [{true 1234-567} {true 1234-abcd} {false
abcc-567}]}
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
  <Name>Dimitris</Name>
  <Surname>Tsoukalos</Surname>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-567</Number>
  </Tel>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-abcd</Number>
  </Tel>
  <Tel>
    <Mobile>false</Mobile>
    <Number>abcc-567</Number>
  </Tel>
```

```
</Record>
JSON: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false
abcc-567}]}
```

Reading an XML file

In this subsection, you will learn how to read an XML file from disk and store it into a Go structure. The name of the program is `readXML.go` and it will be presented in three parts. The first part of `readXML.go` is as follows:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

type Record struct {
    Name      string
    Surname   string
    Tel       []Telephone
}

type Telephone struct {
    Mobile bool
    Number string
}
```

The second part of `readXML.go` is the following:

```
func loadFromXML(filename string, key interface{}) error {
    in, err := os.Open(filename)
    if err != nil {
        return err
    }

    decodeXML := xml.NewDecoder(in)
    err = decodeXML.Decode(key)
    if err != nil {
        return err
    }
    in.Close()
    return nil
}
```

The presented process is very similar to the way that you read a JSON file from disk.

The last part of `readXML.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a filename!")
        return
    }

    filename := arguments[1]

    var myRecord Record
    err := loadFromXML(filename, &myRecord)

    if err == nil {
        fmt.Println("XML:", myRecord)
    } else {
        fmt.Println(err)
    }
}
```

Executing `readXML.go` will generate the following output:

```
$ go run readXML.go data.xml
XML: {Dimitris Tsoukalos [{true 1234-567} {true 1234-abcd} {false
abcc-567}]}
```

The contents of `data.xml` are the following:

```
$ cat data.xml
xmlData: <?xml version="1.0" encoding="UTF-8"?>
<Record>
  <Name>Dimitris</Name>
  <Surname>Tsoukalos</Surname>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-567</Number>
  </Tel>
  <Tel>
    <Mobile>true</Mobile>
    <Number>1234-abcd</Number>
  </Tel>
  <Tel>
    <Mobile>false</Mobile>
    <Number>abcc-567</Number>
  </Tel>
</Record>
```

```
</Tel>
</Record>
```

Customizing XML output

In this subsection, you will learn how to modify and customize the generated XML output. The name of the utility, which will be presented in three parts, is `modXML.go`. Note that the data that is going to be converted into XML is hardcoded in the program for reasons of simplicity.

The first part of `modXML.go` is as follows:

```
package main

import (
    "encoding/xml"
    "fmt"
    "os"
)

func main() {
    type Address struct {
        City, Country string
    }
    type Employee struct {
        XMLName    xml.Name `xml:"employee"`
        Id         int      `xml:"id,attr"`
        FirstName  string   `xml:"name>first"`
        LastName   string   `xml:"name>last"`
        Initials   string   `xml:"name>initials"`
        Height     float32  `xml:"height,omitempty"`
        Address
        Comment string `xml:",comment"`
    }
```

This is where the structure for the XML data is defined. However, there is additional information regarding the name and the type of the XML elements. The `XMLName` field provides the name of the XML record, which in this case will be `employee`.

A field with the tag `",comment"` is a comment and it is formatted as such in the output. A field with the tag `",attr"` appears as an attribute to the provided field name (which is `id` in this case) in the output. The `"name>first"` notation tells Go to embed the `first` tag inside a tag called `name`.

Lastly, a field with the "omitempty" option is omitted from the output if it is empty. An empty value is any of 0, false, nil pointer, or interface, and any array, slice, map, or string with a length of zero.

The second part of `modXML.go` is as follows:

```
r := &Employee{Id: 7, FirstName: "Mihalis", LastName: "Tsoukalos",
Initials: "MIT"}
r.Comment = "Technical Writer + DevOps"
r.Address = Address{"SomeWhere 12", "12312, Greece"}
```

Here we define and initialize an employee structure.

The last part of `modXML.go` is shown in the following Go code:

```
output, err := xml.MarshalIndent(r, " ", " ")
if err != nil {
    fmt.Println("Error:", err)
}
output = []byte(xml.Header + string(output))
os.Stdout.Write(output)
os.Stdout.Write([]byte("\n"))
}
```

Executing `modXML.go` will generate the following output:

```
$ go run modXML.go
<?xml version="1.0" encoding="UTF-8"?>
<employee id="7">
  <name>
    <first>Mihalis</first>
    <last>Tsoukalos</last>
    <initials>MIT</initials>
  </name>
  <City>SomeWhere 12</City>
  <Country>12312, Greece</Country>
  <!--Technical Writer + DevOps-->
</employee>
```

Go and the YAML format

YAML Ain't Markup Language (YAML) is another very popular text format. Although the standard Go library offers no support for the YAML format, you can look at

<https://github.com/go-yaml/yaml> for a Go package that offers YAML support for Go.



The YAML format is supported by the `Viper` package, which is illustrated in Chapter 8, *Telling a UNIX System What to Do*. If you want to learn more about how Viper parses YAML files, you can look at the Viper Go code at <https://github.com/spf13/viper>.

Additional resources

Take a look at the following resources:

- Read the documentation of the `regexp` standard Go package, which can be found at <https://golang.org/pkg/regexp/>.
- Visit the main page of the `grep(1)` utility and find out how it supports regular expressions.
- You can find more information about the `math/big` Go package at <https://golang.org/pkg/math/big/>.
- You can find more information about YAML at <https://yaml.org/>.
- You might find it interesting to look at the `sync.Map` type explained at <https://golang.org/pkg/sync/>.
- Please have a look at the documentation of the `unicode` standard Go package at <https://golang.org/pkg/unicode/>.
- Although you might find it hard at first, start reading *The Go Programming Language Specification* at <https://golang.org/ref/spec>.

Exercises and web links

- Try to write a Go program that prints the invalid part or parts of an IPv4 address.
- Can you state the differences between `make` and `new` without looking at the chapter text?
- Using the code of `findIPv4.go`, write a Go program that prints the most popular IPv4 addresses found in a log file without processing the output with any UNIX utilities.
- Develop a Go program that finds the IPv4 addresses in a log file that generated a 404 HTML error message.
- Read a JSON file with 10 integer values, store it in a `struct` variable, increment each integer value by one, and write the updated JSON entry on disk. Now, do the same for an XML file.

- Develop a Go program that finds all the IPv4 addresses of a log file that downloaded ZIP files.
- Using the `math/big` standard Go package, write a Go program that calculates square roots with high precision – choose the algorithm on your own.
- Write a Go utility that finds a given date and time format in its input and returns just the time part of it.
- Do you remember the differences between a character, a byte, and a rune?
- Develop a Go utility that uses a regular expression in order to match integers from 200 to 400.
- Try to improve `keyValue.go` by adding logging to it.

Summary

In this chapter, we talked about many handy Go features, including creating and using structures, tuples, strings, and runes, and the functionality of the `unicode` standard Go package. Additionally, you learned about pattern matching, regular expressions, the processing of JSON and XML files, the `switch` statement, and the `strings` standard Go package.

Finally, we developed a key-value store in Go and you learned how to use the types of the `math/big` package to calculate Pi with the desired accuracy.

In the next chapter, you will learn how you can group and manipulate data using more advanced arrangements, such as binary trees, linked lists, doubly linked lists, queues, stacks, and hash tables. You will also explore the structures that can be found in the `container` standard Go package, how to perform matrix operations in Go, and how to verify Sudoku puzzles. The last topic of the next chapter will be random numbers and generating difficult-to-guess strings that can potentially be used as secure passwords.

5

How to Enhance Go Code with Data Structures

In the previous chapter, we discussed composite data types, which are constructed using the `struct` keyword, and JSON and XML processing in Go, as well as topics such as regular expressions, pattern matching, tuples, runes, strings, and the `unicode` and `strings` standard Go packages. Finally, we developed a simple key-value store in Go.

There are times, however, when the structures offered by a programming language will not fit a particular problem. In such cases, you will need to create your own data structures to store, search, and receive your data in explicit and specialized ways.

Consequently, this chapter is all about developing and using many famous data structures in Go, including **binary trees**, **linked lists**, **hash tables**, **stacks**, and **queues**, and learning about their advantages. As nothing describes a data structure better than an image, you will see many explanatory figures in this chapter.

The last parts of the chapter will talk about verifying Sudoku puzzles and performing calculations with matrices using slices.

In this chapter, you will learn about the following topics:

- **Graphs** and **nodes**
- Measuring the complexity of an algorithm
- Binary trees
- Hash tables
- Linked lists
- Doubly linked lists
- Working with queues in Go
- Stacks
- The data structures offered by the `container` standard Go package

- Performing matrix calculations
- Working with Sudoku puzzles
- Generating random numbers in Go
- Building random strings that can be used as difficult-to-crack passwords

About graphs and nodes

A graph $G(V, E)$ is a finite, nonempty set of vertices (V) (or nodes) and a set of edges (E) . There are two main types of graphs: **cyclic graphs** and **acyclic graphs**. A cyclic graph is a graph where all or a number of its vertices are connected in a closed chain. In acyclic graphs, there are not any closed chains.

A **directed graph** is a graph with edges that have a direction associated with them, while a **directed acyclic graph** is a directed graph with no cycles in it.



As a node may contain any kind of information, nodes are usually implemented using Go structures due to their versatility.

Algorithm complexity

The efficiency of an algorithm is judged by its computational complexity, which mostly has to do with the number of times the algorithm needs to access its input data to do its job. The **big O notation** is used in computer science for describing the complexity of an algorithm. Thus, an $O(n)$ algorithm, which needs to access its input only once, is considered better than an $O(n^2)$ algorithm, which is better than an $O(n^3)$ algorithm, and so on. The worst algorithms, however, are the ones with an $O(n!)$ running time, which makes them almost unusable for inputs with more than 300 elements.

Lastly, most Go lookup operations in built-in types, such as finding the value of a map key or accessing an array element, have a constant time, which is represented by $O(1)$. This means that built-in types are faster than custom types, and that you should generally favor using them, unless you want full control over what is going on behind the scenes.

Furthermore, not all data structures are created equal. Generally speaking, array operations are faster than map operations, which is the price you have to pay for the versatility of a map.



Although every algorithm has its drawbacks, if you do not have lots of data, the algorithm is not really important as long as it performs the desired job accurately.

Binary trees in Go

A binary tree is a data structure where underneath each node there exist *at most* two other nodes. This means that a node can be connected to one, two, or no other nodes. The **root of a tree** is the first node of the tree. The **depth of a tree**, which is also called the **height of a tree**, is defined as the longest path from the root to a node, whereas the **depth of a node** is the number of edges from the node to the root of the tree. A leaf is a node with no children.

A tree is considered **balanced** when the longest length from the root node to a leaf is at most one more than the shortest such length. An **unbalanced tree** is a tree that is not balanced. Balancing a tree might be a difficult and slow operation, so it is better to keep your tree balanced from the beginning rather than trying to balance it after you have created it, especially when your tree has a large number of nodes.

The next figure shows an unbalanced binary tree. Its root node is J, whereas nodes A, G, W, and D are leaves.

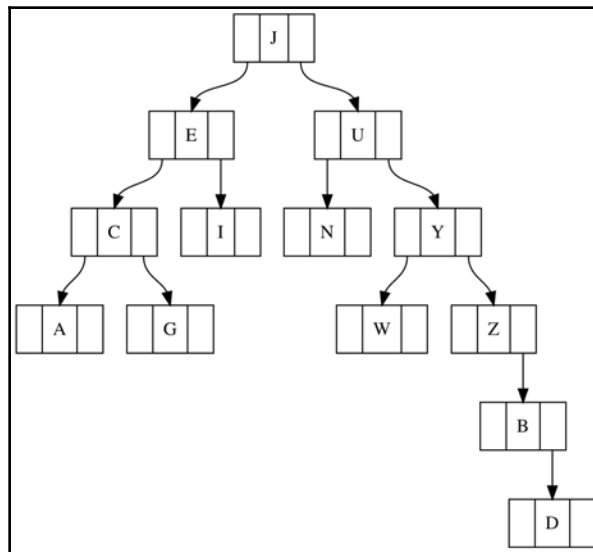


Figure 5.1: An unbalanced binary tree

Implementing a binary tree in Go

This section will illustrate how to implement a binary tree in Go using the source code found in `binTree.go` as an example. The contents of `binTree.go` will be presented in five parts. The first part is next:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)

type Tree struct {
    Left  *Tree
    Value int
    Right *Tree
}
```

What you see here is the definition of the node of the tree using a Go structure. The `math/rand` package is used for populating the tree with random numbers, as we do not have any real data.

The second code portion from `binTree.go` comes with the next Go code:

```
func traverse(t *Tree) {
    if t == nil {
        return
    }
    traverse(t.Left)
    fmt.Print(t.Value, " ")
    traverse(t.Right)
}
```

The `traverse()` function reveals how you can visit all of the nodes of a binary tree using recursion.

The third code segment is as follows:

```
func create(n int) *Tree {
    var t *Tree
    rand.Seed(time.Now().Unix())
    for i := 0; i < 2*n; i++ {
        temp := rand.Intn(n * 2)
        t = insert(t, temp)
    }
}
```

```
    return t
}
```

The `create()` function is only used for populating the binary tree with random integers.

The fourth part of the program is next:

```
func insert(t *Tree, v int) *Tree {
    if t == nil {
        return &Tree{nil, v, nil}
    }
    if v == t.Value {
        return t
    }
    if v < t.Value {
        t.Left = insert(t.Left, v)
        return t
    }
    t.Right = insert(t.Right, v)
    return t
}
```

The `insert()` function does many important things using `if` statements. The first `if` statement checks whether we are dealing with an empty tree or not. If it is indeed an empty tree, then the new node will be the root of the tree and will be created as `&Tree{nil, v, nil}`.

The second `if` statement determines whether the value you are trying to insert already exists in the binary tree or not. If it exists, the function returns without doing anything else.

The third `if` statement determines whether the value you are trying to insert will go on the left or on the right of the node that is currently being examined and acts accordingly. Please note that the presented implementation creates unbalanced binary trees.

The last part of `binTree.go` contains the following Go code:

```
func main() {
    tree := create(10)
    fmt.Println("The value of the root of the tree is",
tree.Value)
    traverse(tree)
    fmt.Println()
    tree = insert(tree, -10)
    tree = insert(tree, -2)
    traverse(tree)
    fmt.Println()
    fmt.Println("The value of the root of the tree is",
```

```
    tree.Value)
}
```

Executing `binTree.go` will generate the next kind of output:

```
$ go run binTree.go
The value of the root of the tree is 18
0 3 4 5 7 8 9 10 11 14 16 17 18 19
-10 -2 0 3 4 5 7 8 9 10 11 14 16 17 18 19
The value of the root of the tree is 18
```

Advantages of binary trees

You cannot beat a tree when you need to represent hierarchical data. For that reason, trees are extensively used when the compiler of a programming language parses a computer program.

Additionally, trees are *ordered* by design, which means that you do not have to make any special effort to order them; putting an element into its correct place keeps them ordered. However, deleting an element from a tree is not always trivial because of the way that trees are constructed.

If a binary tree is balanced, its search, insert, and delete operations take about $\log(n)$ steps, where n is the total number of elements that the tree holds. Additionally, the height of a balanced binary tree is approximately $\log_2(n)$, which means that a balanced tree with 10,000 elements has a height of about 14, and that is remarkably small.

Similarly, the height of a balanced tree with 100,000 elements will be about 17, and the height of a balanced tree with 1,000,000 elements will be about 20. In other words, putting a significantly large number of elements into a balanced binary tree does not change the speed of the tree in an extreme way. Stated differently, you can reach any node of a tree with 1,000,000 nodes in less than 20 steps!

A major disadvantage of binary trees is that the shape of the tree depends on the order in which its elements were inserted. If the keys of a tree are long and complex, then inserting or searching for an element might be slow due to the large number of comparisons required. Finally, if a tree is not balanced, then the performance of the tree will be unpredictable.



Although you can create a linked list or an array faster than a binary tree, the flexibility that a binary tree offers in search operations might be worth the extra overhead and maintenance.



When searching for an element on a binary tree, you check whether the value of the element that you are looking for is bigger or smaller than the value of the current node and use that decision to choose which part of the tree you will go down next. Doing this saves a lot of time.

Hash tables in Go

Strictly speaking, a hash table is a data structure that stores one or more key-value pairs and uses a **hash function** to compute an index into an array of buckets or slots, from which the correct value can be discovered. Ideally, the hash function should assign each key to a unique bucket provided that you have the required number of buckets, which is usually the case.

A good hash function must be able to produce a uniform distribution of the hash values, because it is inefficient to have unused buckets or big differences in the cardinalities of the buckets. Additionally, the hash function should work consistently and output the same hash value for identical keys. Otherwise, it would be impossible to locate the information you want.

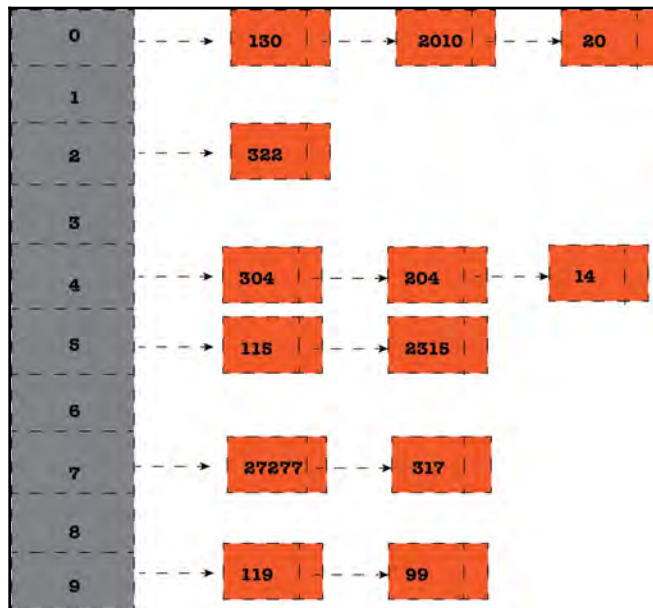


Figure 5.2: A hash table with 10 buckets

Implementing a hash table in Go

The Go code of `hashTable.go`, which will be presented in five parts, will help to clarify many things about hash tables.

The first part of `hashTable.go` is as follows:

```
package main

import (
    "fmt"
)

const SIZE = 15
type Node struct {
    Value int
    Next  *Node
}
```

In this part, you can see the definition of the node of the hash table, which, as expected, is defined using a Go structure. The `SIZE` constant variable holds the number of buckets of the hash table.

The second segment from `hashTable.go` is shown in the following Go code:

```
type HashTable struct {
    Table map[int]*Node
    Size  int
}

func hashFunction(i, size int) int {
    return (i % size)
}
```

In this code segment, you can see the implementation of the hash function used in this particular hash. The `hashFunction()` uses the **modulo operator**. The main reason for choosing the modulo operator is because this particular hash table has to cope with integer values. If you were dealing with strings or floating-point numbers, then you would use a different logic in your hash function.

The actual hash is stored in a `HashTable` structure that has two fields. The first field is a map that associates an integer with a linked list (`*Node`), and the second field is the size of the hash table. As a result, this hash table will have as many linked lists as the number of its buckets. This also means that the nodes of each bucket of the hash table will be stored in linked lists. You will learn more about linked lists in a little while.

The third code portion of `hashTable.go` is as follows:

```
func insert(hash *HashTable, value int) int {
    index := hashFunction(value, hash.Size)
    element := Node{Value: value, Next: hash.Table[index]}
    hash.Table[index] = &element
    return index
}
```

The `insert()` function is called for inserting elements into the hash table. Note that the current implementation of the `insert()` function does not check for duplicate values.

The fourth part of `hashTable.go` is next:

```
func traverse(hash *HashTable) {
    for k := range hash.Table {
        if hash.Table[k] != nil {
            t := hash.Table[k]
            for t != nil {
                fmt.Printf("%d -> ", t.Value)
                t = t.Next
            }
            fmt.Println()
        }
    }
}
```

The `traverse()` function is used for printing all of the values in the hash table. The function visits each of the linked lists of the hash table and prints the stored values, linked list by linked list.

The last code portion of `hashTable.go` is as follows:

```
func main() {
    table := make(map[int]*Node, SIZE)
    hash := &HashTable{Table: table, Size: SIZE}
    fmt.Println("Number of spaces:", hash.Size)
    for i := 0; i < 120; i++ {
        insert(hash, i)
    }
    traverse(hash)
}
```

In this part of the code, you create a new hash table named `hash` using the `table` variable, which is a map that holds the buckets of the hash table. As you already know, the slots of a hash table are implemented using linked lists.

The main reason for using a map to hold the linked lists of a hash table instead of a slice or an array is that the keys of a slice or an array can only be positive integers, while the keys of a map can be almost anything you need.

Executing `hashTable.go` will produce the following output:

```
$ go run hashTable.go
Number of spaces: 15
105 -> 90 -> 75 -> 60 -> 45 -> 30 -> 15 -> 0 ->
110 -> 95 -> 80 -> 65 -> 50 -> 35 -> 20 -> 5 ->
114 -> 99 -> 84 -> 69 -> 54 -> 39 -> 24 -> 9 ->
118 -> 103 -> 88 -> 73 -> 58 -> 43 -> 28 -> 13 ->
119 -> 104 -> 89 -> 74 -> 59 -> 44 -> 29 -> 14 ->
108 -> 93 -> 78 -> 63 -> 48 -> 33 -> 18 -> 3 ->
112 -> 97 -> 82 -> 67 -> 52 -> 37 -> 22 -> 7 ->
113 -> 98 -> 83 -> 68 -> 53 -> 38 -> 23 -> 8 ->
116 -> 101 -> 86 -> 71 -> 56 -> 41 -> 26 -> 11 ->
106 -> 91 -> 76 -> 61 -> 46 -> 31 -> 16 -> 1 ->
107 -> 92 -> 77 -> 62 -> 47 -> 32 -> 17 -> 2 ->
109 -> 94 -> 79 -> 64 -> 49 -> 34 -> 19 -> 4 ->
117 -> 102 -> 87 -> 72 -> 57 -> 42 -> 27 -> 12 ->
111 -> 96 -> 81 -> 66 -> 51 -> 36 -> 21 -> 6 ->
115 -> 100 -> 85 -> 70 -> 55 -> 40 -> 25 -> 10 ->
```

This particular hash table is perfectly balanced because it has to deal with continuous numbers that are placed in a slot according to the results of the modulo operator. Real-world problems might not generate such convenient results!



The remainder of a **Euclidean division** between two natural numbers, a and b , can be calculated according to the $a = bq + r$ formula, where q is the quotient and r is the remainder. The values allowed for the remainder can be between 0 and $b-1$, which are the possible results of the modulo operator.

Note that if you execute `hashTable.go` several times, you will most likely get an output where the lines are in a different order, because the way that Go outputs the key-value pairs of a map is deliberately totally random, and therefore cannot be relied upon.

Implementing the lookup functionality

In this section, you are going to see an implantation of the `lookup()` function that allows you to determine whether a given element already exists in the hash table or not. The code of the `lookup()` function is based on that of the `traverse()` function, as follows:

```
func lookup(hash *HashTable, value int) bool {
    index := hashFunction(value, hash.Size)
    if hash.Table[index] != nil {
        t := hash.Table[index]
        for t != nil {
            if t.Value == value {
                return true
            }
            t = t.Next
        }
    }
    return false
}
```

You can find the preceding code in the `hashTableLookup.go` source file. Executing `hashTableLookup.go` will create the following output:

```
$ go run hashTableLookup.go
120 is not in the hash table!
121 is not in the hash table!
122 is not in the hash table!
123 is not in the hash table!
124 is not in the hash table!
```

The preceding output means that the `lookup()` function does its job pretty well.

Advantages of hash tables

If you are thinking that hash tables are not that useful, handy, or smart, consider the following: when a hash table has n keys and k buckets, the search speed for the n keys goes from $O(n)$ for a linear search down to $O(n/k)$. Although the improvement might look small, for a hash array with only 20 slots, the search time will be reduced by 20 times! This makes hash tables perfect for applications such as dictionaries or any other analogous application where you have to search large amounts of data.

Linked lists in Go

A linked list is a data structure with a finite set of elements where each element uses at least two memory locations: one for storing the actual data and the other for storing a pointer that links the current element to the next one, thus creating a sequence of elements that construct the linked list.

The first element of a linked list is called the **head**, whereas the last element is often called the **tail**. The first thing that you should do when defining a linked list is to keep the head of the list in a separate variable because the head is the only thing that you have to access the entire linked list. Note that if you lose the pointer to that first node of a singly linked list, there is no way to find it again.

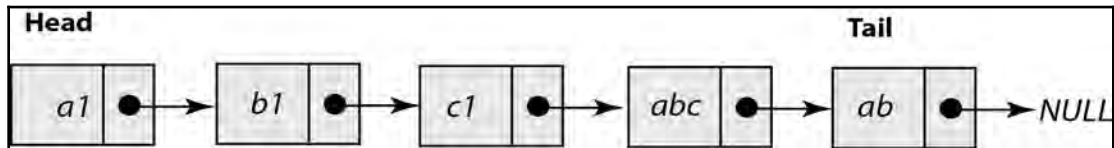


Figure 5.3: A linked list with five nodes

The next figure shows you how to remove an existing node from a linked list in order to better understand the steps that are involved in the process. The main thing that you will need to do is to arrange the pointer to the left node of the node that you are removing in order to point to the right node of the node that is being removed.

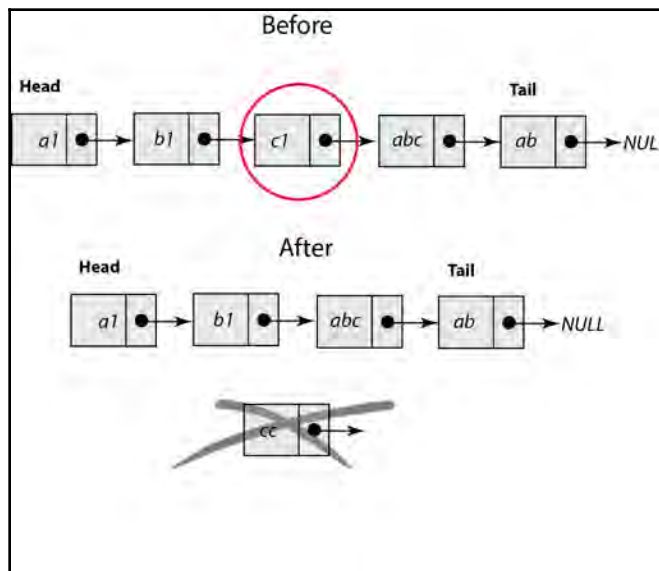


Figure 5.4: Removing a node from a linked list

The linked list implementation that follows is relatively simple and will not include the delete node functionality, which is left as an exercise for you to tackle.

Implementing a linked list in Go

The Go source file for the implementation of the linked list is called `linkedList.go`, and it will be presented in five parts.

The first code segment of `linkedList.go` is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var root = new(Node)
```

In this part of the program, you define the `Node` structure type that will be used for the nodes of the linked list, as well as the `root` global variable that holds the first element of the linked list, which will be accessible everywhere in the code. Bear in mind that although using global variables is generally fine for smaller programs and example code, it might create bugs in larger programs.

The second part of `linkedList.go` is shown in the following Go code:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        t.Next = &Node{v, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

Due to the way that linked lists work, they do not normally contain duplicate entries. Furthermore, new nodes are usually added at the end of a linked list when the linked list is not sorted. Thus, the `addNode()` function is used for adding new nodes to the linked list.

There are three distinct cases in the implementation that are examined using `if` statements. In the first case, you test whether you are dealing with an empty linked list or not. In the second case, you check whether the value that you want to add is already in the list. In the third case, you check whether you have reached the end of the linked list. In this case, you add a new node at the end of the list with the desired value using `t.Next = &Node{v, nil}`. If none of these conditions is true, you repeat the same process with the `addNode()` function for the next node of the linked list using `return addNode(t.Next, v)`.

The third code segment of the `linkedList.go` program contains the implementation of the `traverse()` function:

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

The fourth part of `linkedList.go` is as follows:

```
func lookupNode(t *Node, v int) bool {
    if root == nil {
        t = &Node{v, nil}
        root = t
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}
```

```
}

func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    i := 0
    for t != nil {
        i++
        t = t.Next
    }
    return i
}
```

In this part, you see the implementation of two very handy functions: `lookupNode()` and `size()`. The former checks whether a given element exists in the linked list, while the latter returns the size of the linked list, which is the number of nodes in the linked list.

The logic behind the implementation of the `lookupNode()` function is easy to understand: you start accessing all of the elements of the singly linked list in order to search for the value you want. If you reach the tail of the linked list without having found the desired value, then you know that the linked list does not contain that value.

The last part of `linkedList.go` contains the implementation of the `main()` function:

```
func main() {
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, -1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 45)
    addNode(root, 5)
    addNode(root, 5)
    traverse(root)
    addNode(root, 100)
    traverse(root)

    if lookupNode(root, 100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }
}
```

```
    }

    if lookupNode(root, -100) {
        fmt.Println("Node exists!")
    } else {
        fmt.Println("Node does not exist!")
    }
}
```

Executing `linkedList.go` will generate the following output:

```
$ go run linkedList.go
&{0 <nil>}
-> Empty list!
1 -> -1 ->
Node already exists: 5
Node already exists: 5
1 -> -1 -> 10 -> 5 -> 45 ->
1 -> -1 -> 10 -> 5 -> 45 -> 100 ->
Node exists!
Node does not exist!
```

Advantages of linked lists

The greatest advantages of linked lists are that they are easy to understand and implement, and they are generic enough that they can be used in many different situations. This means that they can be used to model many different kinds of data, starting from single values and going up to complex data structures with many fields. Additionally, linked lists are really fast at sequential searching when used with pointers.

Linked lists not only help you to sort your data, but they can also assist you in keeping your data sorted even after inserting or deleting elements. Deleting a node from a sorted linked list is the same as in an unsorted linked list; however, inserting a new node into a sorted linked list is different because the new node has to go to the right place in order for the list to remain sorted. In practice, this means that if you have lots of data and you know that you will need to delete data all the time, using a linked list is a better choice than using a hash table or a binary tree.

Lastly, **sorted linked lists** allow you to use various optimization techniques when searching for or inserting a node. The most common technique is keeping a pointer at the center node of the sorted linked list and starting your lookups from there. This simple optimization can reduce the time of the lookup operation by half!

Doubly linked lists in Go

A doubly linked list is one where each node keeps a pointer to the previous element on the list, as well as the next element.

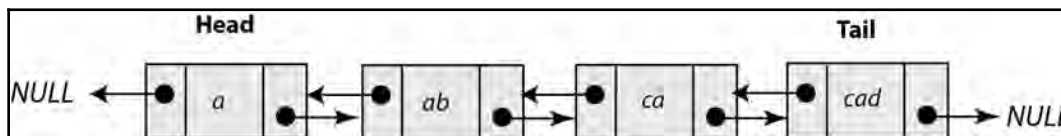


Figure 5.6: A doubly linked list

Thus, on a doubly linked list, the next link of the first node points to the second node, while its previous link points to `nil` (also called **NULL**). Analogously, the next link of the last node points to `nil`, while its previous link points to the penultimate node of the doubly linked list.

The last figure of this chapter illustrates the addition of a node in a doubly linked list. As you can imagine, the main task that needs to be accomplished is dealing with the pointers of three nodes: the new node, the node that will be on the left of the new node, and the node that will be on the right of the new node.

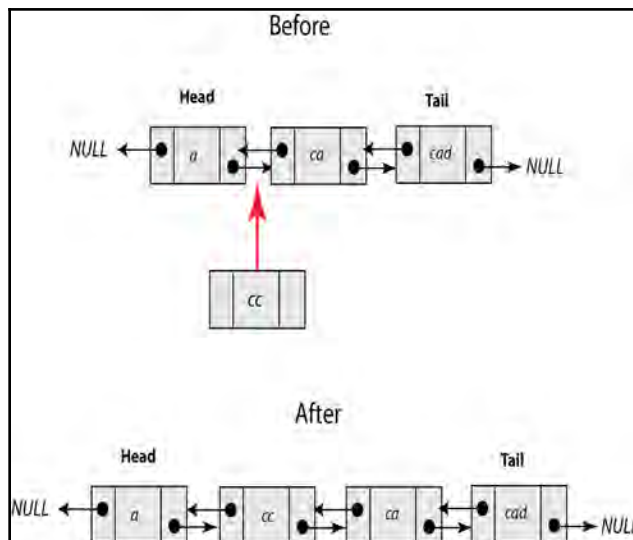


Figure 5.7: Inserting a new node into the middle of a doubly linked list

Thus, in reality, the main difference between a singly linked list and a doubly linked list is that the latter requires more housekeeping. This is the price that you will have to pay for being able to access your doubly linked list both ways.

Implementing a doubly linked list in Go

The name of the program with the Go implementation of a doubly linked list is `doublyLList.go`, and it will be offered to you in five parts. The general idea behind a doubly linked list is the same as with a singly linked list, but you just have to do more housekeeping due to the presence of two pointers in each node of the list.

The first part of `doublyLList.go` is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value      int
    Previous   *Node
    Next       *Node
}
```

In this part, you can see the definition of the node of the doubly linked list using a Go structure. However, this time, the `struct` has two pointer fields for apparent reasons.

The second code portion of `doublyLList.go` contains the following Go code:

```
func addNode(t *Node, v int) int {
    if root == nil {
        t = &Node{v, nil, nil}
        root = t
        return 0
    }

    if v == t.Value {
        fmt.Println("Node already exists:", v)
        return -1
    }

    if t.Next == nil {
        temp := t
```

```
        t.Next = &Node{v, temp, nil}
        return -2
    }

    return addNode(t.Next, v)
}
```

As happened in the case of the singly linked list, each new node is placed at the end of the current doubly linked list. However, this is not mandatory, as you can decide that you want to have a sorted doubly linked list.

The third part of `doublyLList.go` is as follows:

```
func traverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}

func reverse(t *Node) {
    if t == nil {
        fmt.Println("-> Empty list!")
        return
    }

    temp := t
    for t != nil {
        temp = t
        t = t.Next
    }

    for temp.Previous != nil {
        fmt.Printf("%d -> ", temp.Value)
        temp = temp.Previous
    }
    fmt.Printf("%d -> ", temp.Value)
    fmt.Println()
}
```

Here you see the Go code for the `traverse()` and `reverse()` functions. The implementation of the `traverse()` function is the same as in the `LinkedList.go` program. However, the logic behind the `reverse()` function is very interesting. As we do not keep a pointer to the tail of the doubly linked list, we need to go to the end of the doubly linked list before being able to access its nodes in reverse order.

Notice that Go allows you to write code such as `a, b = b, a` in order to swap the values of two variables without the need for a temporary variable.

The fourth part of `doublyLinkedList.go` contains the following Go code:

```
func size(t *Node) int {
    if t == nil {
        fmt.Println("-> Empty list!")
        return 0
    }

    n := 0
    for t != nil {
        n++
        t = t.Next
    }
    return n
}

func lookupNode(t *Node, v int) bool {
    if root == nil {
        return false
    }

    if v == t.Value {
        return true
    }

    if t.Next == nil {
        return false
    }

    return lookupNode(t.Next, v)
}
```

The last code segment of `doublyLinkedList.go` contains the following Go code:

```
var root = new(Node)

func main() {
```

```
    fmt.Println(root)
    root = nil
    traverse(root)
    addNode(root, 1)
    addNode(root, 1)
    traverse(root)
    addNode(root, 10)
    addNode(root, 5)
    addNode(root, 0)
    addNode(root, 0)
    traverse(root)
    addNode(root, 100)
    fmt.Println("Size:", size(root))
    traverse(root)
    reverse(root)
}
```

If you execute `doublyLList.go`, you will get the following output:

```
$ go run doublyLList.go
&{0 <nil> <nil>}
-> Empty list!
Node already exists: 1
1 ->
Node already exists: 0
1 -> 10 -> 5 -> 0 ->
Size: 5
1 -> 10 -> 5 -> 0 -> 100 ->
100 -> 0 -> 5 -> 10 -> 1 ->
```

As you can see, the `reverse()` function works just fine!

Advantages of doubly linked lists

Doubly linked lists are more versatile than singly linked lists because you can traverse them in any direction you want and also you can insert and delete elements from them more easily. Additionally, even if you lose the pointer to the head of a doubly linked list, you can still find the head node of that list. However, this versatility comes at a price: maintaining two pointers for each node. It is up to the developer to decide whether that extra complexity is justified or not. After all, your music player might be using a doubly linked list to represent your current list of songs and be able to go to the previous song as well as the next one.

Queues in Go

A queue is a special kind of linked list where each new element is inserted to the head and removed from the tail of the linked list. I do not need a figure to describe a queue; imagine going to a bank and waiting for the people that came before you to finish their transactions before you can talk to a bank teller.

The main advantage of queues is simplicity. You only need two functions to access a queue, which means that you have to worry about fewer things going wrong and you can implement a queue any way you want as long as you can offer support for those two functions.

Implementing a queue in Go

The program that will illustrate the Go implementation of a queue is called `queue.go`, and it will be presented in five parts. Note that a linked list is going to be used for the implementation of the queue. The `Push()` and `Pop()` functions are used for adding and removing nodes from the queue, respectively.

The first part of the code for `queue.go` is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var size = 0
var queue = new(Node)
```

Having a variable (`size`) for keeping the number of nodes that you have on the queue is handy but not compulsory. However, the implementation presented here supports this functionality because it makes things simpler. In practice, you will probably want to keep these fields in your own structure.

The second code portion of `queue.go` contains the following Go code:

```
func Push(t *Node, v int) bool {
    if queue == nil {
```

```
        queue = &Node{v, nil}
        size++
        return true
    }

    t = &Node{v, nil}
    t.Next = queue
    queue = t
    size++

    return true
}
```

This part displays the implementation of the `Push()` function, which is straightforward. If the queue is empty, then the new node will become the queue. If the queue is not empty, then you create a new node that is placed in front of the current queue. After that, the head of the queue becomes the node that was just created.

The third part of `queue.go` contains the following Go code:

```
func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }

    if size == 1 {
        queue = nil
        size--
        return t.Value, true
    }

    temp := t
    for (t.Next) != nil {
        temp = t
        t = t.Next
    }

    v := (temp.Next).Value
    temp.Next = nil

    size--
    return v, true
}
```

The preceding code shows the implementation of the `Pop()` function, which removes the oldest element of the queue. If the queue is empty (`size == 0`), there is nothing to extract.

If the queue has only one node, then you extract the value of that node and the queue becomes empty. Otherwise, you extract the last element of the queue, remove the last node of the queue, and fix the required pointers before returning the desired value.

The fourth part of `queue.go` contains the following Go code:

```
func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Queue!")
        return
    }
    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

Strictly speaking, the `traverse()` function is not necessary for the operation of a queue, but it gives you a practical way of looking at all of the nodes of the queue.

The last code segment of `queue.go` is shown in the following Go code:

```
func main() {
    queue = nil
    Push(queue, 10)
    fmt.Println("Size:", size)
    traverse(queue)

    v, b := Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    for i := 0; i < 5; i++ {
        Push(queue, i)
    }
    traverse(queue)
    fmt.Println("Size:", size)

    v, b = Pop(queue)
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)

    v, b = Pop(queue)
```

```
    if b {
        fmt.Println("Pop:", v)
    }
    fmt.Println("Size:", size)
    traverse(queue)
}
```

Almost all of the Go code in `main()` is for checking the operation of the queue. The most important code in here is the two `if` statements, which let you know whether the `Pop()` function returned an actual value, or if the queue was empty and there is nothing to return.

Executing `queue.go` will produce the following type of output:

```
$ go run queue.go
Size: 1
10 ->
Pop: 10
Size: 0
4 -> 3 -> 2 -> 1 -> 0 ->
Size: 5
Pop: 0
Size: 4
Pop: 1
Size: 3
4 -> 3 -> 2 ->
```

Stacks in Go

A stack is a data structure that looks like a pile of plates. The last plate that goes on the top of the pile is the one that will be used first when you need to use a new plate.

Like a queue, the main advantage of a stack is its simplicity because you only have to worry about implementing two functions in order to be able to work with a stack: adding a new node to the stack and removing a node from the stack.

Implementing a stack in Go

It is now time to look at the implementation of a stack in Go. This will be illustrated in the `stack.go` source file. Once again, a linked list will be used for implementing the stack. As you know, you will need two functions: one function named `Push()` for putting things on the stack and another one named `Pop()` for removing things from the stack.

Although it is not necessary, it is useful to keep the number of elements that you have on the stack on a separate variable in order to be able to tell whether you are dealing with an empty stack or not, without having to access the linked list itself.

The source code of `stack.go` will be presented in four parts. The first part is as follows:

```
package main

import (
    "fmt"
)

type Node struct {
    Value int
    Next  *Node
}

var size = 0
var stack = new(Node)
```

The second part of `stack.go` contains the implementation of the `Push()` function:

```
func Push(v int) bool {
    if stack == nil {
        stack = &Node{v, nil}
        size = 1
        return true
    }

    temp := &Node{v, nil}
    temp.Next = stack
    stack = temp
    size++
    return true
}
```

If the stack is not empty, then you create a new node (`temp`), which is placed in front of the current stack. After that, this new node becomes the head of the stack. The current version of the `Push()` function always returns `true`, but if your stack does not have unlimited space, you might want to modify it and return `false` when you are about to exceed its capacity.

The third part contains the implementation of the `Pop()` function:

```
func Pop(t *Node) (int, bool) {
    if size == 0 {
        return 0, false
    }
```

```
    }

    if size == 1 {
        size = 0
        stack = nil
        return t.Value, true
    }

    stack = stack.Next
    size--
    return t.Value, true
}
```

The fourth code segment of `stack.go` is as follows:

```
func traverse(t *Node) {
    if size == 0 {
        fmt.Println("Empty Stack!")
        return
    }

    for t != nil {
        fmt.Printf("%d -> ", t.Value)
        t = t.Next
    }
    fmt.Println()
}
```

As the stack is implemented using a linked list, it is traversed as such.

The last part of `stack.go` is shown in the following Go code:

```
func main() {
    stack = nil
    v, b := Pop(stack)
    if b {
        fmt.Print(v, " ")
    } else {
        fmt.Println("Pop() failed!")
    }

    Push(100)
    traverse(stack)
    Push(200)
    traverse(stack)

    for i := 0; i < 10; i++ {
        Push(i)
    }
}
```

```

    }

    for i := 0; i < 15; i++ {
        v, b := Pop(stack)
        if b {
            fmt.Print(v, " ")
        } else {
            break
        }
    }
    fmt.Println()
    traverse(stack)
}

```

As you just saw, the source code of `stack.go` is a little shorter than the Go code of `queue.go`, primarily because the idea behind a stack is simpler than the idea behind a queue.

Executing `stack.go` will generate the following type of output:

```

$ go run stack.go
Pop() failed!
100 ->
200 -> 100 ->
9 8 7 6 5 4 3 2 1 0 200 100
Empty Stack!

```



NOTE: So far, you have seen how a linked list is used in the implementation of a hash table, a queue, and a stack. These examples should help you to realize the usefulness and the importance of linked lists in programming and computer science in general.

The container package

In this section, I will explain the use of the `container` standard Go package. The `container` package supports three data structures: a **heap**, **list**, and **ring**. These data structures are implemented in `container/heap`, `container/list`, and `container/ring`, respectively.

If you are unfamiliar with rings, a ring is a **circular list**, which means that the last element of a ring points to its first element. In essence, this means that all of the nodes of a ring are equivalent and that a ring does not have a beginning and an end. As a result, each element of a ring can help you to traverse the entire ring.

The next three subsections will illustrate each one of the packages contained in the `container` package. The rational advice is that if the functionality of the `container` standard Go package suits your needs, use it; otherwise, you should implement and use your own data structures.

Using container/heap

In this subsection, you will see the functionality that the `container/heap` package offers. First of all, you should know that the `container/heap` package implements a heap, which is a tree where the value of each node of the tree is the smallest element in its subtree. Note that I am using the phrase *smallest element* instead of *minimum value* in order to make it clear that a heap does not only support numerical values.

However, as you can guess, in order to implement a heap tree in Go, you will have to develop a way to tell which of two elements is smaller than the other on your own. In such cases, Go uses **interfaces** because they allow you to define such a behavior.

This means that the `container/heap` package is more advanced than the other two packages found in `container`, and that you will have to define some things before being able to use the functionality of the `container/heap` package. Strictly speaking, the `container/heap` package requires that you implement `container/heap.Interface`, which is defined as follows:

```
type Interface interface {
    sort.Interface
    Push(x interface{}) // add x as element Len()
    Pop() interface{}    // remove and return element Len() - 1.
}
```

You will learn more about interfaces in Chapter 7, *Reflection and Interfaces for All Seasons*. For now, just remember that compliance with a Go interface requires the implementation of one or more functions or other interfaces, which in this case is `sort.Interface`, as well as the `Push()` and `Pop()` functions.

`sort.Interface` requires that you implement the `Len()`, `Less()`, and `Swap()` functions, which makes perfect sense because you cannot perform any kind of sorting without being able to swap two elements, being able to calculate a value for the things that you want to sort, and being able to tell which element between two elements is bigger than the other based on the value that you calculated previously. Although you might think that this is a lot of work, keep in mind that most of the time, the implementation of these functions is either trivial or rather simple.

Since the purpose of this section is to illustrate the use of `container/heap` and not to make your life difficult, the data type for the elements in this example will be `float32`.

The Go code of `conHeap.go` will be presented in five parts. The first part is as follows:

```
package main

import (
    "container/heap"
    "fmt"
)

type heapFloat32 []float32
```

The second part of `conHeap.go` is shown in the following Go code:

```
func (n *heapFloat32) Pop() interface{} {
    old := *n
    x := old[len(old)-1]
    new := old[0 : len(old)-1]
    *n = new
    return x
}

func (n *heapFloat32) Push(x interface{}) {
    *n = append(*n, x.(float32))
}
```

Although you define two functions named `Pop()` and `Push()` here, these two functions are used for interface compliance. In order to add and remove elements from the heap, you should call `heap.Push()` and `heap.Pop()`, respectively.

The third code segment of `conHeap.go` contains the following Go code:

```
func (n heapFloat32) Len() int {
    return len(n)
}

func (n heapFloat32) Less(a, b int) bool {
    return n[a] < n[b]
}

func (n heapFloat32) Swap(a, b int) {
    n[a], n[b] = n[b], n[a]
}
```

This part implements the three functions needed by the `sort.Interface` interface.

The fourth part of `conHeap.go` is as follows:

```
func main() {
    myHeap := &heapFloat32{1.2, 2.1, 3.1, -100.1}
    heap.Init(myHeap)
    size := len(*myHeap)
    fmt.Printf("Heap size: %d\n", size)
    fmt.Printf("%v\n", myHeap)
```

The last code portion of `conHeap.go` is as follows:

```
    myHeap.Push(float32(-100.2))
    myHeap.Push(float32(0.2))
    fmt.Printf("Heap size: %d\n", len(*myHeap))
    fmt.Printf("%v\n", myHeap)
    heap.Init(myHeap)
    fmt.Printf("%v\n", myHeap)
}
```

In this last part of `conHeap.go`, you add two new elements to `myHeap` using `heap.Push()`. However, in order for the heap to get properly resorted, you will need to make another call to `heap.Init()`.

Executing `conHeap.go` will generate the following type of output:

```
$ go run conHeap.go
Heap size: 4
&[-100.1 1.2 3.1 2.1]
Heap size: 6
&[-100.1 1.2 3.1 2.1 -100.2 0.2]
&[-100.2 -100.1 0.2 2.1 1.2 3.1]
```

If you find it strange that the `2.1 1.2 3.1` triplet in the last line of the output is not sorted in the linear logic, remember that a heap is a tree-not a linear structure like an array or a slice.

Using container/list

This subsection will illustrate the operation of the `container/list` package using the Go code of `conList.go`, which will be presented in three parts.



NOTE: The `container/list` package implements a linked list.

The first part of `conList.go` contains the following Go code:

```
package main

import (
    "container/list"
    "fmt"
    "strconv"
)

func printList(l *list.List) {
    for t := l.Back(); t != nil; t = t.Prev() {
        fmt.Print(t.Value, " ")
    }
    fmt.Println()

    for t := l.Front(); t != nil; t = t.Next() {
        fmt.Print(t.Value, " ")
    }

    fmt.Println()
}
```

Here, you see a function named `printList()`, which allows you to print the contents of a `list.List` variable passed as a pointer. The Go code shows you how to print the elements of `list.List` starting from the first element and going to the last element, and vice versa. Usually, you will need to use only one of the two methods in your programs. The `Prev()` and `Next()` functions allow you to iterate over the elements of a list backward and forward.

The second code segment of `conList.go` is as follows:

```
func main() {
    values := list.New()

    e1 := values.PushBack("One")
    e2 := values.PushBack("Two")
    values.PushFront("Three")
    values.InsertBefore("Four", e1)
    values.InsertAfter("Five", e2)
    values.Remove(e2)
```

```
values.Remove(e2)
values.InsertAfter("FiveFive", e2)
values.PushBackList(values)

printList(values)

values.Init()
```

The `list.PushBack()` function allows you to insert an object at the back of a linked list, whereas the `list.PushFront()` function allows you to insert an object at the front of a list. The return value of both functions is the element inserted in the list.

If you want to insert a new element after a specific element, then you should use `list.InsertAfter()`. Similarly, if you want to insert a new element before a specific element, you should use the `list.InsertBefore()` function. If the element does not exist, then the list will not change. `list.PushBackList()` inserts a copy of an existing list at the end of another list, whereas the `list.PushFrontList()` function puts a copy of an existing list at the front of another list. The `list.Remove()` function removes a specific element from a list.

Note the use of the `values.Init()` function, which either empties an existing list or initializes a new list.

The last portion of the code of `conList.go` is shown in the following Go code:

```
fmt.Printf("After Init(): %v\n", values)

for i := 0; i < 20; i++ {
    values.PushFront(strconv.Itoa(i))
}

printList(values)
}
```

Here, you create a new list using a `for` loop. The `strconv.Itoa()` function converts an integer value into a string.

In summary, the use of the functions of the `container/list` package is straightforward and comes with no surprises.

Executing `conList.go` will generate the following type of output:

```
$ go run conList.go
Five One Four Three Five One Four Three
Three Four One Five Three Four One Five
After Init(): &{{0xc420074180 0xc420074180 <nil> <nil>} 0}
```



```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
```

Using container/ring

This section will illustrate the use of the `container/ring` package using the Go code of `conRing.go`, which will be presented in four parts. Note that the `container/ring` package is much simpler than both `container/list` and `container/heap`, which means that it contains fewer functions than the other two packages.

The first code segment of `conRing.go` follows:

```
package main

import (
    "container/ring"
    "fmt"
)

var size int = 10
```

The `size` variable holds the size of the ring that is going to be created.

The second part of `conRing.go` contains the following Go code:

```
func main() {
    myRing := ring.New(size + 1)
    fmt.Println("Empty ring:", *myRing)

    for i := 0; i < myRing.Len()-1; i++ {
        myRing.Value = i
        myRing = myRing.Next()
    }

    myRing.Value = 2
```

Thus, a new ring is created with the help of the `ring.New()` function, which requires a single parameter: the size of the ring. The `myRing.Value = 2` statement at the end adds the value 2 to the ring. That value, however, already exists in the ring as it was added in the `for` loop. Lastly, the zero value of a ring is a ring with a single element whose value is `nil`.

The third part of `conRing.go` is shown in the following Go code:

```
sum := 0
myRing.Do(func(x interface{}) {
```

```
t := x.(int)
sum = sum + t
})
fmt.Println("Sum:", sum)
```

The `ring.Do()` function allows you to call a function for each element of a ring in chronological order. However, if that function makes any changes to the ring, then the behavior of `ring.Do()` is undefined.

The `x.(int)` statement is called **type assertion**. You will learn more about type assertions in Chapter 7, *Reflection and Interfaces for All Seasons*. For now, just know that it shows that `x` is of type `int`.

The last part of the `conRing.go` program is as follows:

```
for i := 0; i < myRing.Len()+2; i++ {
    myRing = myRing.Next()
    fmt.Print(myRing.Value, " ")
}
fmt.Println()
}
```

The only problem with rings is that you can keep calling `ring.Next()` indefinitely, so you will need to find a way to put a stop to that. In this case, this is accomplished with the help of the `ring.Len()` function. Personally, I prefer to use the `ring.Do()` function for iterating over all of the elements of a ring because it generates cleaner code, but using a `for` loop is just as good.

Executing `conRing.go` will generate the following type of output:

```
$ go run conRing.go
Empty ring: {0xc42000a080 0xc42000a1a0 <nil>}
Sum: 47
0 1 2 3 4 5 6 7 8 9 2 0 1
```

The output verifies that a ring can contain duplicate values, which means that unless you use the `ring.Len()` function, you have no safe way of knowing the size of a ring.

Generating random numbers

Random number generation is an art as well as a research area in computer science. This is because computers are purely logical machines, and it turns out that using them to generate random numbers is extremely difficult!

Go uses the `math/rand` package for generating pseudo-random numbers. It needs a **seed** to start producing the numbers. The seed is used for initializing the entire process, and it is extremely important because if you always start with the same seed, you will always get the same sequence of pseudo-random numbers. This means that everybody can regenerate that sequence, and that particular sequence will not be random after all.

The name of the utility that will help us to generate pseudo-random numbers is `randomNumbers.go`, and it will be presented in four parts. The utility takes various parameters, which are the lower and upper limits of the numbers that will be generated, as well as the amount of numbers to generate. If you use a fourth command parameter, the program will use that as the seed of the pseudo-random number generator, which will help you to regenerate the same number sequence – the main reason for doing so is for testing your code.

The first part of the utility is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The `random()` function does all of the job, which is generating pseudo-random numbers in the given range by calling `rand.Intn()`.

The second part of the command-line utility is as follows:

```
func main() {
    MIN := 0
    MAX := 100
    TOTAL := 100
    SEED := time.Now().Unix()

    arguments := os.Args
```

In this part, you initialize the variables that will be used in the program.

The third part of `randomNumbers.go` contains the following Go code:

```
switch len(arguments) {
case 2:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX = MIN + 100
case 3:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
case 4:
    fmt.Println("Usage: ./randomNumbers MIN MAX TOTAL SEED")
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
    TOTAL, _ = strconv.Atoi(arguments[3])
case 5:
    MIN, _ = strconv.Atoi(arguments[1])
    MAX, _ = strconv.Atoi(arguments[2])
    TOTAL, _ = strconv.Atoi(arguments[3])
    SEED, _ = strconv.ParseInt(arguments[4], 10, 64)
default:
    fmt.Println("Using default values!")
}
```

The logic behind this `switch` block is relatively simple: depending on the number of command-line arguments you have, you use either the initial values of the missing arguments or the values given by the user. For reasons of simplicity, the error variables of the `strconv.Atoi()` and `strconv.ParseInt()` functions are being ignored using underscore characters. If this was a commercial program, the error variables of the `strconv.Atoi()` and `strconv.ParseInt()` functions would not have been ignored.

Lastly, the reason for using `strconv.ParseInt()` for setting a new value to the `SEED` variable is that the `rand.Seed()` function requires an `int64` parameter. The first parameter of `strconv.ParseInt()` is the string to parse, the second parameter is the base of the generated number, and the third parameter is the bit size of the generated number.

As we want to create a decimal integer that uses 64 bits, we are using 10 as the base and 64 as the bit size. Please note that had we wanted to parse an unsigned integer, we would have used the `strconv.ParseUint()` function instead.

The last part of `randomNumbers.go` is shown in the following Go code:

```
rand.Seed(SEED)
for i := 0; i < TOTAL; i++ {
    myrand := random(MIN, MAX)
```

```

        fmt.Print(myrand)
        fmt.Print(" ")
    }
    fmt.Println()
}

```



Instead of using the UNIX epoch time as the seed for the pseudo-random number generator, you can use the `/dev/random` system device. You will learn about reading from `/dev/random` in Chapter 8, *Telling a UNIX System What to Do*.

Executing `randomNumbers.go` will create the following type of output:

```

$ go run randomNumbers.go
Using default values!
75 69 15 75 62 67 64 8 73 1 83 92 7 34 8 70 22 58 38 8 54 34 91 65 1 50 76
5 82 61 90 10 38 40 63 6 28 51 54 49 27 52 92 76 35 44 9 66 76 90 10 29 22
20 83 33 92 80 50 62 26 19 45 56 75 40 30 97 23 87 10 43 11 42 65 80 82 25
53 27 51 99 88 53 36 37 73 52 61 4 81 71 57 30 72 51 55 62 63 79
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
1 1
$ go run randomNumbers.go 1 3 2
Usage: ./randomNumbers MIN MAX TOTAL SEED
2 2
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3
$ go run randomNumbers.go 1 5 10 10
3 1 4 4 1 1 4 4 4 3

```

If you are really interested in random number generation, you should start by reading the second volume of *The Art of Computer Programming* by Donald E. Knuth (Addison-Wesley Professional, 2011).

If you intend to use these pseudo-random numbers for security-related reasons, it is important that you use the `crypto/rand` package. This package implements a cryptographically secure pseudo-random number generator. It will be presented later in this chapter.

Generating random strings

Once you know how a computer represents single characters, it is not difficult to go from pseudo-random numbers to random strings. This section will present a technique for creating difficult-to-guess passwords based on the Go code of `randomNumbers.go`, which was presented in the previous section. The name of the Go program for this task will be `generatePassword.go`, and it is going to be presented in four parts. The utility requires just one command-line parameter, which is the length of the password that you want to generate.

The first part of `generatePassword.go` contains the following Go code:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second code portion of `generatePassword.go` contains the following Go code:

```
func main() {
    MIN := 0
    MAX := 94
    var LENGTH int64 = 8

    arguments := os.Args
```

As we only want to get printable ASCII characters, we limit the range of pseudo-random numbers that can be generated. The number of printable characters in the ASCII table is 94. This means that the range of the pseudo-random numbers that the program can generate should be from 0 to 94, not including 94.

The third code segment of `generatePassword.go` is shown in the following Go code:

```
switch len(arguments) {
case 2:
    LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
default:
    fmt.Println("Using default values!")
}

SEED := time.Now().Unix()
rand.Seed(SEED)
```

The last part of `generatePassword.go` is as follows:

```
startChar := "!"
var i int64 = 1
for {
    myRand := random(MIN, MAX)
    newChar := string(startChar[0] + byte(myRand))
    fmt.Print(newChar)
    if i == LENGTH {
        break
    }
    i++
}
fmt.Println()
}
```

The `startChar` variable holds the first ASCII character that can be generated by the utility, which, in this case, is the exclamation mark, which has a decimal ASCII value of 33. Given that the program can generate pseudo-random numbers up to 94, the maximum ASCII value that can be generated is $93 + 33$, which is equal to 126, which is the ASCII value of `~`. The following output shows the ASCII table with the corresponding decimal values for each character:

0 nul	1 soh	2 st	3 etx	4 eot	5 enq	6 ack	7 bel
8 bs	9 ht	10 nl	11 vt	12 np	13 cr	14 so	15 si
16 dle	17 dc1	18 dc	19 dc3	20 dc4	21 nak	22 syn	23 etb
24 can	25 em	26 su	27 esc	28 fs	29 gs	30 rs	31 us
32 sp	33 !	34 "	35 #	36 \$	37 %	38 &	39 '
40 (	41)	42 *	43 +	44 ,	45 -	46 .	47 /
48 0	49 1	50 2	51 3	52 4	53 5	54 6	55 7
56 8	57 9	58 :	59 ;	60 <	61 =	62 >	63 ?
64 @	65 A	66 B	67 C	68 D	69 E	70 F	71 G
72 H	73 I	74 J	75 K	76 L	77 M	78 N	79 O
80 P	81 Q	82 R	83 S	84 T	85 U	86 V	87 W
88 x	89 Y	90 Z	91 [	92 \	93]	94 ^	95 _

96	`	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124		125	}	126	~	127	del



Typing `man ascii` on your favorite UNIX shell will also generate the ASCII table in a readable form.

Executing `generatePassword.go` with the appropriate command-line parameters will create the following type of output:

```
$ go run generatePassword.go
Using default values!
ugs$5mvl
$ go run generatePassword.go
Using default values!
PA/8hA@?
$ go run generatePassword.go 20
HBR+=3\UA'B@ExT4QG|o
$ go run generatePassword.go 20
XLcr|R{*pX/::'t2u^T'
```

Generating secure random numbers

Should you wish to generate more secure pseudo-random numbers in Go, you should use the `crypto/rand` package, which implements a cryptographically secure pseudo-random number generator and is the subject of this section.

The use of the `crypto/rand` package will be illustrated using the Go code of `cryptoRand.go`, which is going to be presented in three parts.

The first part of `cryptoRand.go` is as follows:

```
package main

import (
    "crypto/rand"
    "encoding/base64"
    "fmt"
    "os"
    "strconv"
)
```



```
func generateBytes(n int64) ([]byte, error) {
    b := make([]byte, n)
    _, err := rand.Read(b)
    if err != nil {
        return nil, err
    }

    return b, nil
}
```

The second part of `cryptoRand.go` contains the following Go code:

```
func generatePass(s int64) (string, error) {
    b, err := generateBytes(s)
    return base64.URLEncoding.EncodeToString(b), err
}
```

The last part of `cryptoRand.go` is as follows:

```
func main() {
    var LENGTH int64 = 8
    arguments := os.Args
    switch len(arguments) {
    case 2:
        LENGTH, _ = strconv.ParseInt(os.Args[1], 10, 64)
        if LENGTH <= 0 {
            LENGTH = 8
        }
    default:
        fmt.Println("Using default values!")
    }

    myPass, err := generatePass(LENGTH)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println(myPass[0:LENGTH])
}
```

Executing `cryptoRand.go` will generate the following kind of output:

```
$ go run cryptoRand.go
Using default values!
hIAFYuvW
$ go run cryptoRand.go 120
WTR15SIcjYQmaMKds0lDfFturG27ovH_HZ6iAi_kOnJC88EDLdvNPcv1JjOd9DcF0r0S3q2itXZ
801TNaNFpHkT-aMrsjeue6kUyHnx_EaL_vJHy9wL5RTr8
```

You can find more information about the `crypto/rand` package by visiting its documentation page at <https://golang.org/pkg/crypto/rand/>.

Performing matrix calculations

A **matrix** is an array with two dimensions. The easiest way to represent a matrix in Go is using a slice. However, if you know the dimensions of your array in advance, an array will also do the job just fine. If both dimensions of a matrix are the same, then the matrix is called a **square matrix**.

There are some rules that can tell you whether you can perform a calculation between two matrices or not. The rules are the following:

- In order to add or subtract two matrices, they should have exactly the same dimensions.
- In order to multiply matrix A with matrix B, the number of columns of matrix A should be equal to the number of rows of matrix B. Otherwise, the multiplication of matrices A and B is impossible.
- In order to divide matrix A with matrix B, two conditions must be met. Firstly, you will need to be able to calculate the inverse of matrix B and secondly, you should be able to multiply matrix A with the inverse of matrix B according to the previous rule. Only square matrices can have an inverse.

Adding and subtracting matrices

In this section, you are going to learn how to add and subtract matrices with the help of the `addMat.go` utility, which is going to be presented in three parts. The program uses slices to implement the required matrices.

The first part of `addMat.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)

func random(min, max int) int {
```

```

        return rand.Intn(max-min) + min
    }

    func negativeMatrix(s [][]int) [][]int {
        for i, x := range s {
            for j, _ := range x {
                s[i][j] = -s[i][j]
            }
        }
        return s
    }

```

The `negativeMatrix()` function gets a slice input and returns a new slice where each of the integer elements of the original size is replaced with its opposite integer. As you will soon see, the elements of the two initial matrices are generated using pseudo-random numbers, hence the need for the `random()` function.

The second part of `addMat.go` contains the following Go code:

```

func addMatrices(m1 [][]int, m2 [][]int) [][]int {
    result := make([][]int, len(m1))
    for i, x := range m1 {
        for j, _ := range x {
            result[i] = append(result[i], m1[i][j]+m2[i][j])
        }
    }
    return result
}

```

The `addMatrices()` function accesses the elements of both matrices in order to add them and create the result matrix.

The last part of `addMat.go` is as follows:

```

func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row, col int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[1])
        return
    }
}

```

```

    col, err = strconv.Atoi(arguments[2])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[2])
        return
    }
    fmt.Printf("Using %dx%d arrays\n", row, col)

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    m1 := make([][]int, row)
    m2 := make([][]int, row)

    rand.Seed(time.Now().Unix())
    // Initialize m1 and m2 with random numbers
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            m1[i] = append(m1[i], random(-1, i*j+rand.Intn(10)))
            m2[i] = append(m2[i], random(-1, i*j+rand.Intn(10)))
        }
    }
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    // Adding
    r1 := addMatrices(m1, m2)
    // Subtracting
    r2 := addMatrices(m1, negativeMatrix(m2))
    fmt.Println("r1:", r1)
    fmt.Println("r2:", r2)
}

```

The `main()` function is the controller of the program. Among other things, it makes sure that the user has given the correct type of input, creates the desired matrices, and populates them with pseudo-randomly generated numbers.

Executing `addMat.go` will create the following output:

```

$ go run addMat.go 2 3
Using 2x3 arrays
m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 2 3
Using 2x3 arrays

```

```

m1: [[0 -1 0] [1 1 1]]
m2: [[2 1 0] [7 4 9]]
r1: [[2 0 0] [8 5 10]]
r2: [[-2 -2 0] [-6 -3 -8]]
$ go run addMat.go 3 2
Using 3x2 arrays
m1: [[0 -1] [0 0] [0 1]]
m2: [[2 1] [0 3] [1 9]]
r1: [[2 0] [0 3] [1 10]]
r2: [[-2 -2] [0 -3] [-1 -8]]

```

Multiplying matrices

As you already know, multiplying matrices is much more complex than adding or subtracting matrices, which is also shown in the number of command-line arguments that the presented utility, which is named `mulMat.go`, requires. The `mulMat.go` utility, which is going to be presented in four parts, requires four command-line arguments, which are the dimensions of the first and the second matrices, respectively.

The first part of `mulMat.go` is as follows:

```

package main

import (
    "errors"
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

```

The second part of `mulMat.go` contains the following Go code:

```

func multiplyMatrices(m1 [][]int, m2 [][]int) ([][]int, error) {
    if len(m1[0]) != len(m2) {
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]int, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]int, len(m2[0]))
    }
}

```

```

        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}

```

The way matrices are multiplied is totally different from the way they are added, hence the way `multiplyMatrices()` is implemented. The `multiplyMatrices()` function also returns its own custom error message in case the input matrices do not have the right dimensions that will allow them to get multiplied.

The third part of `mulMat.go` is as follows:

```

func createMatrix(row, col int) [][]int {
    r := make([][]int, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 5 {
        fmt.Println("Wrong number of arguments!")
        return
    }
}

```

The `createMatrix()` function creates a slice with the desired dimensions and populates it with integers that are randomly generated.

The last part of `mulMat.go` is as follows:

```

var row, col int
row, err := strconv.Atoi(arguments[1])
if err != nil {
    fmt.Println("Need an integer: ", arguments[1])
    return
}

col, err = strconv.Atoi(arguments[2])

```

```
    if err != nil {
        fmt.Println("Need an integer: ", arguments[2])
        return
    }

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }
    fmt.Printf("m1 is a %dx%d matrix\n", row, col)
    // Initialize m1 with random numbers
    m1 := createMatrix(row, col)

    row, err = strconv.Atoi(arguments[3])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[3])
        return
    }

    col, err = strconv.Atoi(arguments[4])
    if err != nil {
        fmt.Println("Need an integer: ", arguments[4])
        return
    }

    if col <= 0 || row <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }
    fmt.Printf("m2 is a %dx%d matrix\n", row, col)
    // Initialize m2 with random numbers
    m2 := createMatrix(row, col)
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    // Multiply
    r1, err := multiplyMatrices(m1, m2)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("r1:", r1)
}
```

The `main()` function is the controller of the program that defines the way it operates and makes sure that the correct kind of command-line arguments were given.

Executing `mulMat.go` will create the following output:

```
$ go run mulMat.go 1 2 2 1
m1 is a 1x2 matrix
m2 is a 2x1 matrix
m1: [[-3 -1]]
m2: [[-2] [-1]]
r1: [[7]]
$ go run mulMat.go 5 2 2 1
m1 is a 5x2 matrix
m2 is a 2x1 matrix
m1: [[-1 -2] [-4 -4] [-4 -1] [-2 2] [-5 -5]]
m2: [[-5] [-3]]
r1: [[11] [32] [23] [4] [40]]
$ go run mulMat.go 1 2 3 4
m1 is a 1x2 matrix
m2 is a 3x4 matrix
m1: [[-3 -4]]
m2: [[-5 -2 -2 -3] [-1 -4 -3 -5] [-5 -2 3 3]]
Cannot multiply the given matrices!
```

Dividing matrices

In this subsection, you will learn how to divide two matrices using the Go code of `divMat.go`, which is going to be presented in five parts. The core function of `divMat.go` is called `inverseMatrix()`. What `inverseMatrix()` mainly implements is the calculation of the inverse matrix of a given matrix, which is a pretty complex task. There are ready-to-use Go packages that allow you to inverse a matrix, but I have decided to implement it from scratch.



Not all matrices are invertible. Non-square matrices are not invertible. A **square matrix** that is not invertible is called **singular** or **degenerate** – this happens when the determinant of the square matrix is 0. Singular matrices are very rare.

The `divMat.go` utility requires a single command-line argument that defines the dimensions of the used square matrices.

The first part of `divMat.go` is as follows:

```
package main

import (
    "errors"
    "fmt"
```



```

    "math/rand"
    "os"
    "strconv"
    "time"
)

func random(min, max int) float64 {
    return float64(rand.Intn(max-min) + min)
}

```

This time the `random()` function generates a `float64`. The entire `divMat.go` operates using floating-point numbers, mainly because the inverse of a matrix with integer elements will most likely not be a matrix with integer elements.

The second part of `divMat.go` contains the following Go code:

```

func getCofactor(A [][]float64, temp [][]float64, p int, q int, n int) {
    i := 0
    j := 0

    for row := 0; row < n; row++ {
        for col := 0; col < n; col++ {
            if row != p && col != q {
                temp[i][j] = A[row][col]
                j++
                if j == n-1 {
                    j = 0
                    i++
                }
            }
        }
    }
}

func determinant(A [][]float64, n int) float64 {
    D := float64(0)
    if n == 1 {
        return A[0][0]
    }

    temp := createMatrix(n, n)
    sign := 1

    for f := 0; f < n; f++ {
        getCofactor(A, temp, 0, f, n)
        D += float64(sign) * A[0][f] * determinant(temp, n-1)
        sign = -sign
    }
}

```

```
    return D
}
```

The `getCofactor()` and `determinant()` functions calculate things that are necessary for inverting a matrix. If the determinant of a matrix is 0, then the matrix is singular.

The third part of `divMat.go` is as follows:

```
func adjoint(A [][]float64) ([][]float64, error) {
    N := len(A)
    adj := createMatrix(N, N)
    if N == 1 {
        adj[0][0] = 1
        return adj, nil
    }
    sign := 1
    var temp = createMatrix(N, N)

    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            getCofactor(A, temp, i, j, N)
            if (i+j)%2 == 0 {
                sign = 1
            } else {
                sign = -1
            }
            adj[j][i] = float64(sign) * (determinant(temp, N-1))
        }
    }
    return adj, nil
}

func inverseMatrix(A [][]float64) ([][]float64, error) {
    N := len(A)
    var inverse = createMatrix(N, N)
    det := determinant(A, N)
    if det == 0 {
        fmt.Println("Singular matrix, cannot find its inverse!")
        return nil, nil
    }

    adj, err := adjoint(A)
    if err != nil {
        fmt.Println(err)
        return nil, nil
    }

    fmt.Println("Determinant:", det)
```

```

    for i := 0; i < N; i++ {
        for j := 0; j < N; j++ {
            inverse[i][j] = float64(adj[i][j]) / float64(det)
        }
    }

    return inverse, nil
}

```

The `adjoint()` function calculates the adjoint matrix of the given matrix. The `inverseMatrix()` function is what calculates the inverse of the given matrix.



The `divMat.go` program is a great example of Go code that needs to be extensively tested before putting it into production. You will learn more about testing in Chapter 11, *Code Testing, Optimization, and Profiling*.

The fourth part of `divMat.go` contains the following Go code:

```

func multiplyMatrices(m1 [][]float64, m2 [][]float64) ([][]float64, error)
{
    if len(m1[0]) != len(m2) {
        return nil, errors.New("Cannot multiply the given matrices!")
    }

    result := make([][]float64, len(m1))
    for i := 0; i < len(m1); i++ {
        result[i] = make([]float64, len(m2[0]))
        for j := 0; j < len(m2[0]); j++ {
            for k := 0; k < len(m2); k++ {
                result[i][j] += m1[i][k] * m2[k][j]
            }
        }
    }
    return result, nil
}

func createMatrix(row, col int) [][]float64 {
    r := make([][]float64, row)
    for i := 0; i < row; i++ {
        for j := 0; j < col; j++ {
            r[i] = append(r[i], random(-5, i*j))
        }
    }
    return r
}

```

The `multiplyMatrices()` function is needed because the division of a matrix with another is equal to the multiplication of the first matrix with the inverse of the second one.

The last part of `divMat.go` is as follows:

```
func main() {
    rand.Seed(time.Now().Unix())
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Wrong number of arguments!")
        return
    }

    var row int
    row, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println("Need an integer:", arguments[1])
        return
    }
    col := row
    if col <= 0 {
        fmt.Println("Need positive matrix dimensions!")
        return
    }

    m1 := createMatrix(row, col)
    m2 := createMatrix(row, col)
    fmt.Println("m1:", m1)
    fmt.Println("m2:", m2)

    inverse, err := inverseMatrix(m2)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println("\t\t\tPrinting inverse matrix!")
    for i := 0; i < len(inverse); i++ {
        for j := 0; j < len(inverse[0]); j++ {
            fmt.Printf("%.2f\t", inverse[i][j])
        }
        fmt.Println()
    }

    fmt.Println("\t\t\tPrinting result!")
    r1, err := multiplyMatrices(m1, inverse)
    if err != nil {
        fmt.Println(err)
    }
}
```

```

        return
    }

    for i := 0; i < len(r1); i++ {
        for j := 0; j < len(r1[0]); j++ {
            fmt.Printf("%.3f\t", r1[i][j])
        }
        fmt.Println()
    }
}

```

Once again, the `main()` function orchestrates the flow of the program and makes the necessary checks on the user input before proceeding.

Executing `divMat.go` will create the following output:

```

$ go run divMat.go 2
m1: [[-3 -3] [-4 -4]]
m2: [[-3 -5] [-4 -1]]
Determinant: -17
        Printing inverse matrix!
0.06      -0.29
-0.24      0.18
        Printing result!
0.529      0.353
0.706      0.471
$ go run divMat.go 3
m1: [[-3 -5 -2] [-1 -4 1] [-2 -5 -1]]
m2: [[-2 -4 -5] [-1 0 -2] [-2 -2 1]]
Determinant: -22
        Printing inverse matrix!
0.18      -0.64      -0.36
-0.23      0.55      -0.05
-0.09      -0.18      0.18
        Printing result!
0.773      -0.455      0.955
0.636      -1.727      0.727
0.864      -1.273      0.773
$ go run divMat.go 2
m1: [[-3 -5] [-5 -5]]
m2: [[-5 -3] [-5 -3]]
Singular matrix, cannot find its inverse!
        Printing inverse matrix!
        Printing result!
Cannot multiply the given matrices!

```

A tip on finding out the dimensions of an array

In this section, I am going to present you with a way to find the dimensions of an array using the Go code found in `dimensions.go`. The same technique can be used for finding out the dimensions of a slice.

The Go code of `dimensions.go` is as follows:

```
package main

import (
    "fmt"
)

func main() {
    array := [12][4][7][10]float64{}
    x := len(array)
    y := len(array[0])
    z := len(array[0][0])
    w := len(array[0][0][0])
    fmt.Println("x:", x, "y:", y, "z:", z, "w:", w)
}
```

There is an array called `array` with four dimensions. The `len()` function allows you to find its dimensions when provided with the right arguments. Finding the first dimension requires a call to `len(array)`, whereas finding the second dimension requires a call to `len(array[0])`, and so on.

Executing `dimensions.go` will generate the following output:

```
$ go run dimensions.go
x: 12 y: 4 z: 7 w: 10
```

Solving Sudoku puzzles

The main purpose of this section is to help you to understand that you should always use the simplest data structure that does the job you want it to. In this case, that data structure will be a humble slice, which will be used for representing and verifying a Sudoku puzzle. Alternatively, we could have used an array because Sudoku puzzles have a predefined size.

A Sudoku is a logic-based, combinatorial, number-placement puzzle. Verifying a Sudoku puzzle means making sure that the Sudoku puzzle is correctly solved – this is a task that can be easily done by a computer program.

In order to be as generic as possible, the presented utility, which is named `sudoku.go` and will be presented in four parts, will load Sudoku puzzles from external files.

The first part of `sudoku.go` is as follows:

```
package main

import (
    "bufio"
    "errors"
    "fmt"
    "io"
    "os"
    "strconv"
    "strings"
)

func importFile(file string) ([][]int, error) {
    var err error
    var mySlice = make([][]int, 0)

    f, err := os.Open(file)
    if err != nil {
        return nil, err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        fields := strings.Fields(line)
        temp := make([]int, 0)
        for _, v := range fields {
            n, err := strconv.Atoi(v)
            if err != nil {
                return nil, err
            }
            temp = append(temp, n)
        }
        if len(temp) != 0 {
            mySlice = append(mySlice, temp)
        }

        if err == io.EOF {
            break
        } else if err != nil {
            return nil, err
        }
    }
}
```

```

        if len(temp) != len(mySlice[0]) {
            return nil, errors.New("Wrong number of elements!")
        }
    }
    return mySlice, nil
}

```

The `importFile()` function only checks whether it reads valid integer numbers or not. Put simply, `importFile()` will accept negative integers or integers bigger than 9 but it will not accept the value `a`, which is not a number or a float. The only other test that `importFile()` will perform is making sure that all the lines in the input file have the same number of integers. The first line of the input text file is the one that specifies the number of columns that should exist in each input line.

The second code portion of `sudoku.go` contains the following Go code:

```

func validPuzzle(sl [][]int) bool {
    for i := 0; i <= 2; i++ {
        for j := 0; j <= 2; j++ {
            iEl := i * 3
            jEl := j * 3
            mySlice := []int{0, 0, 0, 0, 0, 0, 0, 0, 0}
            for k := 0; k <= 2; k++ {
                for m := 0; m <= 2; m++ {
                    bigI := iEl + k
                    bigJ := jEl + m
                    val := sl[bigI][bigJ]
                    if val > 0 && val < 10 {
                        if mySlice[val-1] == 1 {
                            fmt.Println("Appeared 2 times:", val)
                            return false
                        } else {
                            mySlice[val-1] = 1
                        }
                    } else {
                        fmt.Println("Invalid value:", val)
                        return false
                    }
                }
            }
        }
    }
}

```

In order to access all the elements of a Sudoku puzzle, `sudoku.go` uses four `for` loops. Although using four `for` loops for a 9x9 array might not be a performance issue, it would definitely be a problem if we had to work with much bigger arrays.

The third part of `sudoku.go` is as follows:

```
// Testing columns
for i := 0; i <= 8; i++ {
    sum := 0
    for j := 0; j <= 8; j++ {
        sum = sum + sl[i][j]
    }
    if sum != 45 {
        return false
    }
    sum = 0
}

// Testing rows
for i := 0; i <= 8; i++ {
    sum := 0
    for j := 0; j <= 8; j++ {
        sum = sum + sl[j][i]
    }
    if sum != 45 {
        return false
    }
    sum = 0
}

return true
}
```

The presented code makes sure that each column and row of the Sudoku puzzle contains all numbers.

The last part of `sudoku.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Printf("usage: loadFile textFile size\n")
        return
    }

    file := arguments[1]

    mySlice, err := importFile(file)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

```
    if validPuzzle(mySlice) {  
        fmt.Println("Correct Sudoku puzzle!")  
    } else {  
        fmt.Println("Incorrect Sudoku puzzle!")  
    }  
}
```

The `main()` function orchestrates the entire program.

Executing `sudoku.go` with various input files will generate the following kind of output:

```
$ go run sudoku.go OK.txt  
Correct Sudoku puzzle!  
$ go run sudoku.go noOK1.txt  
Incorrect Sudoku puzzle!
```

Additional resources

You should find the following resources very useful:

- Examine the **Graphviz** utility website. This utility lets you draw graphs using its own language: <http://graphviz.org/>.
- Read the documentation page of the sub-packages of the container standard Go package by visiting <https://golang.org/pkg/container/>.
- Should you wish to learn more about data structures, you should read *The Design and Analysis of Computer Algorithms* by Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman (Addison-Wesley, 1974). It is an excellent book!
- You can learn more about hash functions by visiting https://en.wikipedia.org/wiki/Hash_function.
- Another really interesting book about algorithms and data structures is *Programming Pearls*, written by Jon Bentley (Addison-Wesley Professional, 1999) as well as *More Programming Pearls: Confessions of a Coder*, also by John Bentley (Addison-Wesley Professional, 1988). Reading both books will make you a better programmer.

Exercises

- Try to change the logic behind `generatePassword.go` by picking the password from a list of passwords found in a Go slice combined with the current system time or date.

- Make the necessary changes to the code of `queue.go` in order to store floating-point numbers instead of integers.
- Change the Go code of `stack.go` so that its nodes have three data fields of integer type, named `Value`, `Number`, and `Seed`. Apart from the apparent changes to the definition of the `Node` struct, what is the main change that you will need to make to the rest of the program?
- Can you change the code in `linkedList.go` in order to keep the nodes of the linked list sorted?
- Similarly, can you change the Go code of `doublyLList.go` in order to keep the nodes of the list sorted? Can you develop a function for deleting existing nodes?
- Change the code of `hashTableLookup.go` so that you do not have duplicate values in your hash table. Use the `lookup()` function for that.
- Rewrite `sudoku.go` in order to use a Go map instead of a slice.
- Rewrite `sudoku.go` in order to use a linked list instead of a slice. Why is this difficult?
- Create a Go program that can calculate the powers of matrices. Are there any conditions that need to be met in order to find the power of a matrix?
- Implement the addition and subtraction of arrays with three dimensions.
- Try to represent matrices using Go structures. What are the main challenges?
- Can you modify the Go code of `generatePassword.go` in order to generate passwords that only contain uppercase letters?
- Try to change the code of `conHeap.go` in order to support a custom and more complex structure instead of just `float32` elements.
- Implement the delete node functionality that is missing from `linkedList.go`.
- Do you think that a doubly linked list would make the code of the `queue.go` program better? Try to implement a queue using a doubly linked list instead of a singly linked list.

Summary

This chapter talked about many interesting and practical topics, including implementing linked lists, doubly linked lists, hash tables, queues, and stacks in Go, as well as using the functionality of the `container` standard Go package, verifying Sudoku puzzles, and generating pseudo-random numbers, along with difficult-to-guess passwords, in Go.

What you should remember from this chapter is that the foundation of every data structure is the definition and the implementation of its node. Lastly, we talked about performing matrix calculations.

I am sure that you will find the next chapter to be one of the most interesting and valuable chapters of this book. The main topic is Go packages, along with information about how to define and use the various types of Go functions in your programs. Additionally, the chapter will talk about modules, which, put simply, are packages with versions.

6

What You Might Not Know About Go Packages and Functions

The previous chapter talked about developing and using custom data structures like linked lists, binary trees, and hash tables, as well as generating random numbers and difficult-to-guess passwords in Go and performing matrix operations.

The main focus of this chapter is Go **packages**, which are the Go way of organizing, delivering, and using code. The most common components of a Go package are **functions**, which are pretty flexible in Go. Additionally, this chapter will talk about Go **modules**, which are packages with versions. In the last part of this chapter, you will see some advanced packages that belong to the Go standard library in order to better understand that not all Go packages are created equal.

In this chapter, you will learn about the following topics:

- Developing functions in Go
- Anonymous functions
- Functions that return multiple values
- Giving names to the return values of a function
- Functions that return other functions
- Functions that get other functions as parameters
- Variadic functions
- Developing Go packages
- Developing and working with Go modules
- Private and public package objects
- The use of the `init()` function in packages

- The sophisticated `html/template` standard Go package
- The `text/template` standard package, which is another truly sophisticated Go package that has its own language
- The `go/scanner`, `go/parser`, and `go/token` advanced packages
- The `syscall` standard Go package, which is a low-level package that, although you might not use it directly, is extensively used by other Go packages

About Go packages

Everything in Go is delivered in the form of packages. A Go package is a Go source file that begins with the `package` keyword followed by the name of the package. Some packages have a structure. For example, the `net` package has several subdirectories, named `http`, `mail`, `rpc`, `smtp`, `textproto`, and `url`, which should be imported as `net/http`, `net/mail`, `net/rpc`, `net/smtp`, `net/textproto`, and `net/url`, respectively.

Apart from the packages of the Go standard library, there exist external packages that can be imported using their full address and that should be downloaded before their first use. One such example is `github.com/matryer/is`, which is stored in GitHub.

Packages are mainly used for grouping related functions, variables, and constants so that you can transfer them easily and use them in your own Go programs. Note that apart from the `main` package, Go packages are not autonomous programs and cannot be compiled into executable files. This means that they need to be called directly or indirectly from a `main` package in order to be used. As a result, if you try to execute a Go package as if it is an autonomous program, you will be disappointed:

```
$ go run aPackage.go
go run: cannot run non-main package
```

About Go functions

Functions are an important element of every programming language because they allow you to break big programs into smaller and more manageable parts. Functions must be as independent from each other as possible and must do one job and only one job well. So, if you find yourself writing functions that do multiple things, you might consider replacing them with multiple functions instead.

The single most popular Go function is `main()`, which is used in every independent Go program. You should already know that all function definitions begin with the `func` keyword.

Anonymous functions

Anonymous functions can be defined inline without the need for a name and they are usually used for implementing things that require a small amount of code. In Go, a function can return an anonymous function or take an anonymous function as one of its arguments. Additionally, anonymous functions can be attached to Go variables. Note that anonymous functions are also called **closures**, especially in functional programming terminology.



It is considered good practice for anonymous functions to have a small implementation and a local focus. If an anonymous function does not have a local focus, then you might need to consider making it a regular function.

When an anonymous function is suitable for a job it is extremely convenient and makes your life easier; just do not use too many anonymous functions in your programs without having a good reason. You will see anonymous functions in action in a while.

Functions that return multiple values

As you already know from functions such as `strconv.Atoi()`, Go functions can return multiple distinct values, which saves you from having to create a dedicated structure for returning and receiving multiple values from a function. You can declare a function that returns four values (two `int` values, one `float64` value, and one `string`) as follows:

```
func aFunction() (int, int, float64, string) {  
}
```

It is now time to illustrate anonymous functions and functions that return multiple values in more detail using the Go code of `functions.go` as an example. The relevant code will be presented in five parts.

The first code portion of `functions.go` is as follows:

```
package main  
  
import (  
    "fmt"  
    "os"
```

```
    "strconv"  
)
```

The second code segment from `functions.go` is shown in the following Go code:

```
func doubleSquare(x int) (int, int) {  
    return x * 2, x * x  
}
```

Here you can see the definition and implementation of a function named `doubleSquare()`, which requires a single `int` parameter and returns two `int` values.

The third part of the `functions.go` program is as follows:

```
func main() {  
    arguments := os.Args  
    if len(arguments) != 2 {  
        fmt.Println("The program needs 1 argument!")  
        return  
    }  
  
    y, err := strconv.Atoi(arguments[1])  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
}
```

The preceding code deals with the command-line arguments of the program.

The fourth portion of the `functions.go` program contains the following Go code:

```
square := func(s int) int {  
    return s * s  
}  
fmt.Println("The square of", y, "is", square(y))  
  
double := func(s int) int {  
    return s + s  
}  
fmt.Println("The double of", y, "is", double(y))
```

Each of the `square` and `double` variables holds an anonymous function. The bad part is that you are allowed to change the value of `square`, `double`, or any other variable that holds an anonymous function afterward, which means that the meaning of those variables can change and calculate something else instead.



It is not considered good programming practice to alter the code of variables that hold anonymous functions because this might be the root cause of nasty bugs.

The last part of `functions.go` is as follows:

```
    fmt.Println(doubleSquare(y))
    d, s := doubleSquare(y)
    fmt.Println(d, s)
}
```

So, you can either print the return values of a function, such as `doubleSquare()`, or assign them to distinct variables.

Executing `functions.go` will generate the following kind of output:

```
$ go run functions.go 1 21
The program needs 1 argument!
$ go run functions.go 10.2
strconv.Atoi: parsing "10.2": invalid syntax
$ go run functions.go 10
The square of 10 is 100
The double of 10 is 20
20 100
20 100
```

The return values of a function can be named

Unlike C, Go allows you to name the return values of a Go function. Additionally, when such a function has a `return` statement without any arguments, then the function automatically returns the current value of each named return value in the order in which it was declared in the definition of the function.

The source code that illustrates Go functions that have named return values is `returnNames.go`, and it will be presented in three parts.

The first part of the `returnNames.go` program is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
```

```
)

func namedMinMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return
}
```

In this code segment, you can see the implementation of the `namedMinMax()` function, which uses named return parameters. However, there is a tricky point here: the `namedMinMax()` function does not explicitly return any variables or values in its `return` statement. Nevertheless, as this function has named return values in its signature, the `min` and `max` parameters are automatically returned in the order in which they were put into the function definition.

The second code segment from `returnNames.go` is as follows:

```
func minMax(x, y int) (min, max int) {
    if x > y {
        min = y
        max = x
    } else {
        min = x
        max = y
    }
    return min, max
}
```

The `minMax()` function also uses named return values, but its `return` statement specifically defines the order and the variables that are going to be returned.

The last code portion from `returnNames.go` is shown in the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) < 3 {
        fmt.Println("The program needs at least 2 arguments!")
        return
    }

    a1, _ := strconv.Atoi(arguments[1])
    a2, _ := strconv.Atoi(arguments[2])
}
```

```
    fmt.Println(minMax(a1, a2))
    min, max := minMax(a1, a2)
    fmt.Println(min, max)

    fmt.Println(namedMinMax(a1, a2))
    min, max = namedMinMax(a1, a2)
    fmt.Println(min, max)
}
```

The purpose of the Go code in the `main()` function is to verify that all methods generate the same results.

Executing `returnNames.go` will produce the following output:

```
$ go run returnNames.go -20 1
-20 1
-20 1
-20 1
-20 1
```

Functions with pointer parameters

A function can take pointer parameters provided that its signature allows it. The Go code of `ptrFun.go` will illustrate the use of pointers as function parameters.

The first part of `ptrFun.go` is as follows:

```
package main

import (
    "fmt"
)

func getPtr(v *float64) float64 {
    return *v * *v
}
```

So, the `getPtr()` function accepts a pointer parameter that points to a `float64` value.

The second part of the program is shown in the following Go code:

```
func main() {
    x := 12.2
    fmt.Println(getPtr(&x))
    x = 12
    fmt.Println(getPtr(&x))
}
```

```
}
```

The tricky part here is that you need to pass the address of the variable to the `getPtr()` function because it requires a pointer parameter, which can be done by putting an ampersand in front of a variable (`&x`).

Executing `ptrFun.go` will generate the following kind of output:

```
$ go run ptrFun.go
148.83999999999997
144
```

If you try to pass a plain value, such as `12.12`, to `getPtr()` and call it, such as `getPtr(12.12)`, the compilation of the program will fail, as shown in the following error message:

```
$ go run ptrFun.go
# command-line-arguments
./ptrFun.go:15:21: cannot use 12.12 (type float64) as type *float64 in
argument to getPtr
```

Functions that return pointers

As illustrated in `pointerStruct.go` from Chapter 4, *The Uses of Composite Types*, it is considered good practice to create new structure variables using a separate function and return a pointer to them from that function. So, the scenario of functions returning pointers is very common. Generally speaking, such a function simplifies the structure of a program and allows the developer to concentrate on more important things instead of copying the same Go code all the time. This section will use a much simpler example, as found in the Go code of `returnPtr.go`.

The first part of `returnPtr.go` contains the following Go code:

```
package main

import (
    "fmt"
)

func returnPtr(x int) *int {
    y := x * x
    return &y
}
```

Apart from the expected preamble, this code portion defines a new function that returns a pointer to an `int` variable. The only thing to remember is to use `&y` in the `return` statement in order to return the memory address of the `y` variable.

The second part of `returnPtr.go` is as follows:

```
func main() {
    sq := returnPtr(10)
    fmt.Println("sq value:", *sq)
```

The `*` character dereferences a pointer variable, which means that it returns the actual value stored at the memory address instead of the memory address itself.

The last code segment from `returnPtr.go` is shown in the following Go code:

```
    fmt.Println("sq memory address:", sq)
}
```

The preceding code will return the memory address of the `sq` variable, not the `int` value stored in it.

If you execute `returnPtr.go`, you will see the following output (the memory address will differ):

```
$ go run returnPtr.go
sq value: 100
sq memory address: 0xc00009a000
```

Functions that return other functions

In this section, you are going to learn how to implement a Go function that returns another function using the Go code of `returnFunction.go`, which will be presented in three segments. The first code segment of `returnFunction.go` is as follows:

```
package main

import (
    "fmt"
)

func funReturnFun() func() int {
    i := 0
    return func() int {
        i++
        return i * i
    }
}
```

```
    }  
}
```

As you can see from the implementation of `funReturnFun()`, its return value is an anonymous function (`func() int`).

The second code segment from `returnFunction.go` contains the following code:

```
func main() {  
    i := funReturnFun()  
    j := funReturnFun()  
}
```

In this code, you call `funReturnFun()` two times and assign its return value, which is a function, to two separate variables named `i` and `j`. As you will see in the output of the program, the two variables are completely unrelated to each other.

The last code section of `returnFunction.go` is as follows:

```
    fmt.Println("1:", i())  
    fmt.Println("2:", i())  
    fmt.Println("j1:", j())  
    fmt.Println("j2:", j())  
    fmt.Println("3:", i())  
}
```

So, in this Go code, you use the `i` variable three times as `i()` and the `j` variable two times as `j()`. The important thing here is that although both `i` and `j` were created by calling `funReturnFun()`, they are totally independent from each other and have nothing in common.

Executing `returnFunction.go` will produce the following output:

```
$ go run returnFunction.go  
1: 1  
2: 4  
j1: 1  
j2: 4  
3: 9
```

As you can see from the output of `returnFunction.go`, the value of `i` in `funReturnFun()` keeps increasing and does not become 0 after each call either to `i()` or `j()`.

Functions that accept other functions as parameters

Go functions can accept other Go functions as parameters, which is a feature that adds versatility to what you can do with a Go function. The two most common uses of this functionality are functions for sorting elements and the `filepath.Walk()` function. However, in the example presented here, which is named `funFun.go`, we will implement a much simpler case that deals with integer values. The relevant code will be presented in three parts.

The first code segment of `funFun.go` is shown in the following Go code:

```
package main

import "fmt"

func function1(i int) int {
    return i + i
}

func function2(i int) int {
    return i * i
}
```

What we have here is two functions that both accept an `int` and return an `int`. These functions will be used as parameters to another function in a short while.

The second code segment of `funFun.go` contains the following code:

```
func funFun(f func(int) int, v int) int {
    return f(v)
}
```

The `funFun()` function accepts two parameters, a function parameter named `f` and an `int` value. The `f` parameter should be a function that takes one `int` argument and returns an `int` value.

The last code segment of `funFun.go` follows:

```
func main() {
    fmt.Println("function1:", funFun(function1, 123))
    fmt.Println("function2:", funFun(function2, 123))
    fmt.Println("Inline:", funFun(func(i int) int {return i * i
    *i}, 123))
}
```

The first `fmt.Println()` call uses `funFun()` with `function1`, without any parentheses, as its first parameter, whereas the second `fmt.Println()` call uses `funFun()` with `function2` as its first parameter.

In the last `fmt.Println()` statement, something magical happens: the implementation of the function parameter is defined inside the call to `funFun()`. Although this method works fine for simple and small function parameters, it might not work that well for functions with many lines of Go code.

Executing `funFun.go` will produce the next output:

```
$ go run funFun.go
function1: 246
function2: 15129
Inline: 1860867
```

Variadic functions

Go also supports **variadic functions**, which are functions that accept a variable number of arguments. The most popular variadic functions can be found in the `fmt` package. Variadic functions will be illustrated in `variadic.go`, which will be presented in three parts.

The first part of `variadic.go` is as follows:

```
package main

import (
    "fmt"
    "os"
)

func varFunc(input ...string) {
    fmt.Println(input)
}
```

This code part presents the implementation of a variadic function named `varFunc()` that accepts string arguments. The `input` function argument is a slice and will be handled as a slice inside the `varFunc()` function. The `...` operator used as `...Type` is called the **pack operator**, whereas the **unpack operator** ends with `...` and begins with a slice. A variadic function cannot use the pack operator more than once.

The second part of `variadic.go` contains the following Go code:

```
func oneByOne(message string, s ...int) int {
    fmt.Println(message)
    sum := 0
    for i, a := range s {
        fmt.Println(i, a)
        sum = sum + a
    }
    s[0] = -1000
    return sum
}
```

Here you can see another variadic function named `oneByOne()` that accepts a single string and a variable number of integer arguments. The `s` function argument is a slice.

The last part of `variadic.go` is as follows:

```
func main() {
    arguments := os.Args
    varFunc(arguments...)
    sum := oneByOne("Adding numbers...", 1, 2, 3, 4, 5, -1, 10)
    fmt.Println("Sum:", sum)
    s := []int{1, 2, 3}
    sum = oneByOne("Adding numbers...", s...)
    fmt.Println(s)
}
```

The `main()` function is what calls and uses the two variadic functions. As the second call to `oneByOne()` uses a slice, any changes you make to the slice inside the variadic function will remain after the function exits.

Building and executing `variadic.go` will generate the following output:

```
$ ./variadic 1 2
[./variadic 1 2]
Adding numbers...
0 1
1 2
2 3
3 4
4 5
5 -1
6 10
Sum: 24
Adding numbers...
0 1
```

```
1 2
2 3
[-1000 2 3]
```

Developing your own Go packages

The source code of a Go package, which can contain multiple files and multiple directories, can be found within a single directory that is named after the package name, with the obvious exception of the `main` package, which can be located anywhere.

For the purposes of this section, a simple Go package named `aPackage` will be developed. The source file of the package is called `aPackage.go`, and its source code will be presented in two parts.

The first part of `aPackage.go` is shown in the following Go code:

```
package aPackage

import (
    "fmt"
)

func A() {
    fmt.Println("This is function A!")
}
```

Notice that using capital letters in Go package names is not considered a good practice – `aPackage` is only used here as an example.

The second code segment of `aPackage.go` follows:

```
func B() {
    fmt.Println("privateConstant:", privateConstant)
}

const MyConstant = 123
const privateConstant = 21
```

As you can see, developing a new Go package is pretty easy. Right now, you cannot use that package on its own and you need to create a package named `main` with a `main()` function in it in order to create an executable file. In this case, the name of the program that will use `aPackage` is `useAPackage.go`, and it is included in the following Go code:

```
package main
```

```
import (  
    "aPackage"  
    "fmt"  
)  
  
func main() {  
    fmt.Println("Using aPackage!")  
    aPackage.A()  
    aPackage.B()  
    fmt.Println(aPackage.MyConstant)  
}
```

If you try to execute `useAPackage.go` right now, however, you will get an error message, which means that we are not done yet:

```
$ go run useAPackage.go  
useAPackage.go:4:2: cannot find package "aPackage" in any of:  
    /usr/local/Cellar/go/1.9.2/libexec/src/aPackage (from $GOROOT)  
    /Users/mtsouk/go/src/aPackage (from $GOPATH)
```

There is another thing that you will need to handle. As you already know from Chapter 1, *Go and the Operating System*, Go requires the execution of specific commands from the UNIX shell in order to install all external packages, which also includes packages that you have developed locally. Therefore, you will need to put the preceding package in the appropriate directory and make it available to the current UNIX user. Thus, installing one of your own packages involves the execution of the following commands from your favorite UNIX shell:

```
$ mkdir ~/go/src/aPackage  
$ cp aPackage.go ~/go/src/aPackage/  
$ go install aPackage  
$ cd ~/go/pkg/darwin_amd64/  
$ ls -l aPackage.a  
-rw-r--r--  1 mtsouk  staff   4980 Dec 22 06:12 aPackage.a
```



If the `~/go` directory does not already exist, you will need to create it with the help of the `mkdir(1)` command. In that case, you will also need to do the same for the `~/go/src` directory.

Executing `useAPackage.go` will create the following output:

```
$ go run useAPackage.go  
Using aPackage!  
This is function A!  
privateConstant: 21  
123
```

Compiling a Go package

Although you cannot execute a Go package if it does not include a `main()` function, you are still allowed to compile it and create an object file, as follows:

```
$ go tool compile aPackage.go
$ ls -l aPackage.*
-rw-r--r--@ 1 mtsouk  staff    201 Jan 10 22:08 aPackage.go
-rw-r--r--  1 mtsouk  staff  16316 Mar  4 20:01 aPackage.o
```

Private variables and functions

What differentiates private variables and functions from public ones is that private ones can be strictly used and called internally in a package. Controlling which functions, constants, and variables are public or not is also known as **encapsulation**.

Go follows a simple rule that states that functions, variables, types, and so forth that begin with an uppercase letter are public, whereas functions, variables, types, and so on that begin with a lowercase letter are private. This is the reason that `fmt.Println()` is named `Println()` instead of `println()`. However, this rule does not affect package names that are allowed to begin with uppercase and lowercase letters.

The `init()` function

Every Go package can optionally have a private function named `init()` that is automatically executed at the beginning of the execution time.



The `init()` function is a private function by design, which means that it cannot be called from outside the package in which it is contained. Additionally, as the user of a package has no control over the `init()` function, you should think carefully before using an `init()` function in public packages or changing any global state in `init()`.

I will now present a code example with multiple `init()` functions from multiple Go packages. Examine the code of the following basic Go package, which is simply called `a`:

```
package a

import (
    "fmt"
)
```

```
func init() {  
    fmt.Println("init() a")  
}  
  
func FromA() {  
    fmt.Println("fromA()")  
}
```

The `a` package implements an `init()` function and a public one named `FromA()`.

After that, you will need to execute the following commands from your UNIX shell so that the package becomes available to the current UNIX user:

```
$ mkdir ~/go/src/a  
$ cp a.go ~/go/src/a/  
$ go install a
```

Now, look at the code of the next Go code package, which is named `b`:

```
package b  
  
import (  
    "a"  
    "fmt"  
)  
  
func init() {  
    fmt.Println("init() b")  
}  
  
func FromB() {  
    fmt.Println("fromB()")  
    a.FromA()  
}
```

What is happening here? Package `a` uses the `fmt` standard Go package. However, package `b` needs to import package `a`, as it uses `a.FromA()`. Both `a` and `b` have an `init()` function.

As before, you will need to install that package and make it available to the current UNIX user by executing the following commands from your UNIX shell:

```
$ mkdir ~/go/src/b  
$ cp b.go ~/go/src/b  
$ go install b
```

Thus, we currently have two Go packages that both have an `init()` function. Now try to guess the output that you will get from executing `manyInit.go`, which comes with the following code:

```
package main

import (
    "a"
    "b"
    "fmt"
)

func init() {
    fmt.Println("init() manyInit")
}

func main() {
    a.FromA()
    b.FromB()
}
```

The actual question could have been: how many times is the `init()` function of package `a` going to be executed? Executing `manyInit.go` will generate the following output and shed some light on this question:

```
$ go run manyInit.go
init() a
init() b
init() manyInit
fromA()
fromB()
fromA()
```

The preceding output shows that the `init()` function of `a` is executed only once, despite the fact that the `a` package is imported two times by two different packages. Additionally, as the `import` block from `manyInit.go` is executed first, the `init()` functions of package `a` and package `b` are executed before the `init()` function of `manyInit.go`, which makes perfect sense. The main reason for this is that the `init()` function of `manyInit.go` is allowed to use an element from either `a` or `b`.

Notice that `init()` can be very useful when you want to set up some unexported internal variables. As an example, you might find the current time zone in `init()`. Finally, bear in mind that you can have many `init()` functions within one file; however, this Go feature is rarely used.

Go modules

Go modules were first introduced in Go version 1.11. At the time of writing, the latest Go version is 1.13. Although the general idea behind Go modules will remain the same, some of the presented details might change in future versions of Go.

A Go module is like a Go package with a version. Go uses **semantic versioning** for versioning modules. This means that versions begin with the letter `v` followed by the version number. Therefore, you can have versions such as `v1.0.0`, `v1.0.5`, and `v2.0.2`. The `v1`, `v2`, or `v3` part signifies the major version of a Go package that is usually not backwards compatible. This means that if your Go program works with `v1`, it will not necessarily work with `v2` or `v3` – it might work, but you cannot count on it.

The second number in a version is about features. Usually `v1.1.0` has more features than `v1.0.2` or `v1.0.0`, while being compatible with all older versions.

Lastly, the third number is just about bug fixes without having any new features. Note that semantic versioning is also used for Go versions.

Note that Go modules allow you to write things outside of `GOPATH`.

Creating and using a Go module

In this subsection, we are going to create the first version of a basic module. You will need to have a GitHub repository for storing your Go code. In my case, the GitHub repository will be `https://github.com/mactsouk/myModule`. We will begin with an empty GitHub repository – only `README.md` will be there. So, first we will need to execute the following command to get the contents of the GitHub repository:

```
$ git clone git@github.com:mactsouk/myModule.git
Cloning into 'myModule'...
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (6/6), done.
remote: Total 7 (delta 1), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (7/7), done.
Resolving deltas: 100% (1/1), done.
```

If you execute the same command on your computer, you will get my GitHub repository, which will not be empty at the time you read this. If you want to create your own Go module from scratch, you will need to create your own empty GitHub repository.

Creating version v1.0.0

We will need to execute the following commands to create v1.0.0 of our basic Go module:

```
$ go mod init
go: creating new go.mod: module github.com/mactsouk/myModule
$ touch myModule.go
$ vi myModule.go
$ git add .
$ git commit -a -m "Initial version 1.0.0"
$ git push
$ git tag v1.0.0
$ git push -q origin v1.0.0
$ go list
github.com/mactsouk/myModule
$ go list -m
github.com/mactsouk/myModule
```

The contents of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.0.0")
}
```

The contents of `go.mod`, which was created previously, will be as follows:

```
$ cat go.mod
module github.com/mactsouk/myModule
go 1.12
```

Using version v1.0.0

In this section, you are going to learn how to use v1.0.0 of the Go module we created earlier. In order to use our Go modules, we will need to create a Go program, which in this case will be called `useModule.go` and will contain the following Go code:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
)
```



```
func main() {
    v1.Version()
}
```

You will need to include the path of the Go module (`github.com/mactsouk/myModule`) – in this case the Go module also has an alias (`v1`). Using aliases for packages is highly irregular in Go; however, in this example it makes the code easier to read. Nevertheless, this feature should not be used on production code without a really good reason.

If you just try to execute `useModule.go`, which in this case is going to be put in `/tmp`, it will fail because the required module is not present on your system:

```
$ pwd
/tmp
$ go run useModule.go
useModule.go:4:2: cannot find package "github.com/mactsouk/myModule" in any
of:
    /usr/local/Cellar/go/1.12/libexec/src/github.com/mactsouk/myModule
    (from $GOROOT)
    /Users/mtsouk/go/src/github.com/mactsouk/myModule (from $GOPATH)
```

Therefore, you will need to execute the following commands to get the required Go modules and to successfully execute `useModule.go`:

```
$ export GO111MODULE=on
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.0.0
go: downloading github.com/mactsouk/myModule v1.0.0
go: extracting github.com/mactsouk/myModule v1.0.0
Version 1.0.0
```

So, `useModule.go` is correct and can be executed. Now it is time to make things even more official by giving `useModule.go` a name and building it:

```
$ go mod init hello
go: creating new go.mod: module hello
$ go build
```

The last command generates an executable file inside `/tmp`, as well as two additional files named `go.sum` and `go.mod`. The contents of `go.sum` will be as follows:

```
$ cat go.sum
github.com/mactsouk/myModule v1.0.0
h1:eTCn2JewnaJw0REKONrVhHmeDEJ0Q5TAZ0xsSbh8kFs=
github.com/mactsouk/myModule v1.0.0/go.mod
h1:s3ziarTDDvaXaHWYYOf/ULi97aoBd6JfnvAkM8rSuzg=
```

The `go.sum` file keeps a checksum of all the modules it has downloaded.

The contents of `go.mod` will be as follows:

```
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.0.0
```



Please note that if the `go.mod` file found in your project specifies the use of version `v1.3.0` of a Go module, Go will use version `v1.3.0` even if a newer version of the Go module is available.

Creating version v1.1.0

In this subsection, we are going to create a new version of `myModule` using a different **tag**. However, this time there is no need to execute `go mod init`, as this was done before. You will just need to execute the following commands:

```
$ vi myModule.go
$ git commit -a -m "v1.1.0"
[master ddd0742] v1.1.0
 1 file changed, 1 insertion(+), 1 deletion(-)
$ git push
$ git tag v1.1.0
$ git push -q origin v1.1.0
```

The contents of this version of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 1.1.0")
}
```

Using version v1.1.0

In this section, you are going to learn how to use v1.1.0 of the Go module we created. This time we are going to use a **Docker image** in order to be as independent from the machine we used for developing the module as possible. The command we will use to get the Docker image and going into its UNIX shell is the following:

```
$ docker run --rm -it golang:latest
root@884c0d188694:/go# cd /tmp
root@58c5688e3ee0:/tmp# go version
go version go1.13 linux/amd64
```

As you can see, the Docker image uses the latest version of Go, which at the time of writing is 1.13. In order to use one or more Go modules, you will need to create a Go program, which is called `useUpdatedModule.go` and contains the following Go code:

```
package main

import (
    v1 "github.com/mactsouk/myModule"
)

func main() {
    v1.Version()
}
```

The Go code of `useUpdatedModule.go` is the same as the Go code of `useModule.go`. The good thing is that you will automatically get the latest update of version v1.

After writing the program in the Docker image, you will need to do the following:

```
root@58c5688e3ee0:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 91 Mar  2 19:59 useUpdatedModule.go
root@58c5688e3ee0:/tmp# export GO111MODULE=on
root@58c5688e3ee0:/tmp# go run useUpdatedModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0
```

This means that `useUpdatedModule.go` is automatically using the latest v1 version of the Go module. It is critical that you execute `export GO111MODULE=on` to turn on module support.

If you try to execute `useModule.go`, which is located in the `/tmp` directory of your local machine, you will get the following:

```
$ ls -l go.mod go.sum useModule.go
-rw----- 1 mtsouk wheel 67 Mar 2 21:29 go.mod
-rw----- 1 mtsouk wheel 175 Mar 2 21:29 go.sum
-rw-r--r-- 1 mtsouk wheel 92 Mar 2 21:12 useModule.go
$ go run useModule.go
Version 1.0.0
```

This means that `useModule.go` is still using the older version of the Go module. If you want `useModule.go` to use the latest version of the Go module, you can do the following:

```
$ rm go.mod go.sum
$ go run useModule.go
go: finding github.com/mactsouk/myModule v1.1.0
go: downloading github.com/mactsouk/myModule v1.1.0
go: extracting github.com/mactsouk/myModule v1.1.0
Version 1.1.0
```

If you want to go back to using `v1.0.0` of the module, you can do the following:

```
$ go mod init hello
go: creating new go.mod: module hello
$ go build
$ go run useModule.go
Version 1.1.0
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.1.0
$ vi go.mod
$ cat go.mod
module hello
go 1.12
require github.com/mactsouk/myModule v1.0.0
$ go run useModule.go
Version 1.0.0
```

The next subsection will create a new major version of the Go module, which means that instead of using a different tag, we will need to use a different GitHub branch.

Creating version v2.0.0

In this subsection, we are going to create the second major version of `myModule`. Note that for major versions, you will need to be explicit in your `import` statements.

So `github.com/mactsouk/myModule` will become
`github.com/mactsouk/myModule/v2` for version v2 and
`github.com/mactsouk/myModule/v3` for v3.

The first thing to do is create a new GitHub branch:

```
$ git checkout -b v2
Switched to a new branch 'v2'
$ git push --set-upstream origin v2
```

Then, you should do the following:

```
$ vi go.mod
$ cat go.mod
module github.com/mactsouk/myModule/v2
go 1.12
$ git commit -a -m "Using 2.0.0"
[v2 5af2269] Using 2.0.0
 2 files changed, 2 insertions(+), 2 deletions(-)
$ git tag v2.0.0
$ git push --tags origin v2
Counting objects: 4, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 441 bytes | 441.00 KiB/s, done.
Total 4 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To github.com:mactsouk/myModule.git
 * [new branch]      v2 -> v2
 * [new tag]         v2.0.0 -> v2.0.0
$ git --no-pager branch -a
master
* v2
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/v2
```

The contents of this major version of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.0.0")
}
```

Using version v2.0.0

Once again, in order to use our Go modules, we will need to create a Go program, which is called `useV2.go` and contains the following Go code:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

We are going to use a Docker image. This is the most convenient way of playing with Go modules because we are starting with a clean Go installation:

```
$ docker run --rm -it golang:latest
root@191d84fc5571:/go# cd /tmp
root@191d84fc5571:/tmp# cat > useV2.go
package main
import (
    v "github.com/mactsouk/myModule/v2"
)
func main() {
    v.Version()
}
root@191d84fc5571:/tmp# export GO111MODULE=on
root@191d84fc5571:/tmp# go run useV2.go
go: finding github.com/mactsouk/myModule/v2 v2.0.0
go: downloading github.com/mactsouk/myModule/v2 v2.0.0
go: extracting github.com/mactsouk/myModule/v2 v2.0.0
Version 2.0.0
```

Everything is working fine as the Docker image is using version `v2.0.0` of `myModule`.

Creating version v2.1.0

We are now going to create an updated version of `myModule.go` that has to do with using a different GitHub tag. So, execute the following commands:

```
$ vi myModule.go
$ git commit -a -m "v2.1.0"
$ git push
$ git tag v2.1.0
```

```
$ git push -q origin v2.1.0
```

The updated contents of `myModule.go` will be as follows:

```
package myModule

import (
    "fmt"
)

func Version() {
    fmt.Println("Version 2.1.0")
}
```

Using version v2.1.0

As you already know, in order to use our Go modules, we will need to create a Go program, which will be called `useUpdatedV2.go` and contains the following Go code:

```
package main

import (
    v "github.com/mactsouk/myModule/v2"
)

func main() {
    v.Version()
}
```

Still, there is no need to declare that you want to use the latest `v2` version of the Go module because this is handled by Go, which is the main reason that `useUpdatedV2.go` and `useV2.go` are exactly the same.

Once again, a Docker image will be used for reasons of simplicity – the reason for using the `cat (1)` command to create `useUpdatedV2.go` is because that particular Docker image comes without `vi (1)` installed.

```
$ docker run --rm -it golang:1.12
root@ccfcd675e333:/go# cd /tmp/
root@ccfcd675e333:/tmp# cat > useUpdatedV2.go
package main
import (
    v "github.com/mactsouk/myModule/v2"
)
func main() {
    v.Version()
}
```

```

}
root@ccfcd675e333:/tmp# ls -l
total 4
-rw-r--r-- 1 root root 92 Mar  2 20:34 useUpdatedV2.go
root@ccfcd675e333:/tmp# go run useUpdatedV2.go
useUpdatedV2.go:4:2: cannot find package "github.com/mactsouk/myModule/v2"
in any of:
    /usr/local/go/src/github.com/mactsouk/myModule/v2 (from $GOROOT)
    /go/src/github.com/mactsouk/myModule/v2 (from $GOPATH)
root@ccfcd675e333:/tmp# export GO111MODULE=on
root@ccfcd675e333:/tmp# go run useUpdatedV2.go
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 2.1.0

```



You will learn more about `git` (1) and GitHub in Chapter 7, *Reflection and Interfaces for All Seasons*.

Using two different versions of the same Go module

In this subsection, you will see how to use two different major versions of the same Go module in a single Go program. The same technique can be used if you want to use more than two major versions of a Go module at the same time.

The name of the Go source file will be `useTwo.go` and it is as follows:

```

package main

import (
    v1 "github.com/mactsouk/myModule"
    v2 "github.com/mactsouk/myModule/v2"
)

func main() {
    v1.Version()
    v2.Version()
}

```

So, you just need to explicitly import the major versions of the Go module you want to use and give them different aliases.

Executing `useTwo.go` will generate the following kind of output:

```
$ export GO111MODULE=on
$ go run useTwo.go
go: creating new go.mod: module github.com/PacktPublishing/Mastering-Go-Second-Edition
go: finding github.com/mactsouk/myModule/v2 v2.1.0
go: downloading github.com/mactsouk/myModule/v2 v2.1.0
go: extracting github.com/mactsouk/myModule/v2 v2.1.0
Version 1.1.0
Version 2.1.0
```

Where Go stores Go modules

In this section, we are going to see where and how Go stores the code and the information about the Go modules we are using, using our Go module as an example. Here are the contents of the `~/go/pkg/mod/github.com/mactsouk` directory after using the presented Go module on my local macOS Mojave machine:

```
$ ls -lR ~/go/pkg/mod/github.com/mactsouk
total 0
drwxr-xr-x  3 mtsouk  staff    96B Mar  2 22:38 my!module
dr-x-----  6 mtsouk  staff   192B Mar  2 21:18 my!module@v1.0.0
dr-x-----  6 mtsouk  staff   192B Mar  2 22:07 my!module@v1.1.0
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module:
total 0
dr-x-----  6 mtsouk  staff   192B Mar  2 22:38 v2@v2.1.0
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module/v2@v2.1.0:
total 24
-r--r--r--  1 mtsouk  staff    28B Mar  2 22:38 README.md
-r--r--r--  1 mtsouk  staff    48B Mar  2 22:38 go.mod
-r--r--r--  1 mtsouk  staff    86B Mar  2 22:38 myModule.go
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.0.0:
total 24
-r--r--r--  1 mtsouk  staff    28B Mar  2 21:18 README.md
-r--r--r--  1 mtsouk  staff    45B Mar  2 21:18 go.mod
-r--r--r--  1 mtsouk  staff    86B Mar  2 21:18 myModule.go
/Users/mtsouk/go/pkg/mod/github.com/mactsouk/my!module@v1.1.0:
total 24
-r--r--r--  1 mtsouk  staff    28B Mar  2 22:07 README.md
-r--r--r--  1 mtsouk  staff    45B Mar  2 22:07 go.mod
-r--r--r--  1 mtsouk  staff    86B Mar  2 22:07 myModule.go
```



The best way to learn how to develop and use Go modules is to experiment. Go modules are here to stay, so start using them.

The go mod vendor command

There are times when you need to store all your dependencies in the same place and keep them close to the files of your project. In these situations, the `go mod vendor` command can help you to do exactly this:

```
$ cd useTwoVersions
$ go mod init useV1V2
go: creating new go.mod: module useV1V2
$ go mod vendor
$ ls -l
total 24
-rw----- 1 mtsouk staff 114B Mar  2 22:43 go.mod
-rw----- 1 mtsouk staff 356B Mar  2 22:43 go.sum
-rw-r--r--@ 1 mtsouk staff 143B Mar  2 19:36 useTwo.go
drwxr-xr-x  4 mtsouk staff 128B Mar  2 22:43 vendor
$ ls -l vendor/github.com/mactsouk/myModule
total 24
-rw-r--r-- 1 mtsouk staff  28B Mar  2 22:43 README.md
-rw-r--r-- 1 mtsouk staff  45B Mar  2 22:43 go.mod
-rw-r--r-- 1 mtsouk staff  86B Mar  2 22:43 myModule.go
drwxr-xr-x  6 mtsouk staff 192B Mar  2 22:43 v2
$ ls -l vendor/github.com/mactsouk/myModule/v2
total 24
-rw-r--r-- 1 mtsouk staff  28B Mar  2 22:43 README.md
-rw-r--r-- 1 mtsouk staff  48B Mar  2 22:43 go.mod
-rw-r--r-- 1 mtsouk staff  86B Mar  2 22:43 myModule.go
```

The key point here is to execute `go mod init <package name>` before executing the `go mod vendor` command.

Creating good Go packages

This section will provide some handy advice that will help you to develop better Go packages. We have covered that Go packages are organized in directories and can contain public and private elements. Public elements can be used both internally and externally from other packages, whereas private elements can only be used internally in a package.

Here are several good rules to follow to create superior Go packages:

- The first unofficial rule of a successful package is that its elements must be related in some way. Thus, you can create a package for supporting cars, but it would not be a good idea to create a single package for supporting both cars and bicycles. Put simply, it is better to split the functionality of a package unnecessarily into multiple packages than to add too much functionality to a single Go package. Additionally, packages should be made simple and stylish—but not too simplistic and fragile.
- A second practical rule is that you should use your own packages first for a reasonable amount of time before giving them to the public. This will help you to discover silly bugs and make sure that your packages operate as expected. After that, give them to some fellow developers for additional testing before making them publicly available.
- Next, try to imagine the kinds of users who will use your packages happily and make sure that your packages will not create more problems to them than they can solve.
- Unless there is a very good reason for doing so, your packages should not export an endless list of functions. Packages that export a short list of functions are understood better and used more easily. After that, try to title your functions using descriptive but not very long names.
- **Interfaces** can improve the usefulness of your functions, so when you think it is appropriate, use an interface instead of a single type as a function parameter or return type.
- When updating one of your packages, try not to break things and create incompatibilities with older versions unless it is absolutely necessary.
- When developing a new Go package, try to use multiple files in order to group similar tasks or concepts.
- Additionally, try to follow the rules that exist in the Go packages of the standard library. Reading the code of a Go package that belongs to the standard library will help you on this.
- Do not create a package that already exists from scratch. Make changes to the existing package and maybe create your own version of it.
- Nobody wants a Go package that prints logging information on the screen. It would be more professional to have a flag for turning on logging when needed.
- The Go code of your packages should be in harmony with the Go code of your programs. This means that if you look at a program that uses your packages and your function names stand out in the code in a bad way, it would be better to change the names of your functions. As the name of a package is used almost everywhere, try to use a concise and expressive package name.

- It is more convenient if you put new Go type definitions near the place that they will be used for the first time because nobody, including you, wants to search source files for definitions of new data types.
- Try to create test files for your packages, because packages with test files are considered more professional than ones without them; small details make all the difference and give people confidence that you are a serious developer! Notice that writing tests for your packages is not optional and that you should avoid using packages that do not include tests. You will learn more about testing in Chapter 11, *Code Testing, Optimization, and Profiling*.
- Finally, do not write a Go package because you do not have anything better to do – in that case, find something better to do and do not waste your time!



Always remember that apart from the fact that the actual Go code in a package should be bug-free, the next most important element of a successful package is its **documentation**, as well as some code examples that clarify its use and showcase the idiosyncrasies of the functions of the package.

The syscall package

This section will present a small portion of the functionality of the `syscall` standard Go package. Note that the `syscall` package offers a plethora of functions and types related to low-level operating system primitives. Additionally, the `syscall` package is extensively used by other Go packages, such as `os`, `net`, and `time`, which all provide a portable interface to the operating system. This means that the `syscall` package is not the most portable package in the Go library – that is not its job.

Although UNIX systems have many similarities, they also exhibit various differences, especially when we talk about their system internals. The job of the `syscall` package is to deal with all of these incompatibilities as gently as possible. The fact that this is not a secret and is well documented makes `syscall` a successful package.

Strictly speaking, a **system call** is a programmatic way for an application to request something from the kernel of an operating system. As a consequence, system calls are responsible for accessing and working with most UNIX low-level elements such as processes, storage devices, printing data, network interfaces, and all kinds of files. Put simply, you cannot work on a UNIX system without using system calls. You can inspect the system calls of a UNIX process using utilities such as `strace(1)` and `dtrace(1)`, which were presented in Chapter 2, *Understanding Go Internals*.

The use of the `syscall` package will be illustrated in the `useSyscall.go` program, which will be presented in four parts.



You might not directly need to use the `syscall` package unless you are working on pretty low-level stuff. Not all Go packages are for everyone!

The first code portion of `useSyscall.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "syscall"
)
```

This is the easy part of the program, where you just import the required Go packages.

The second part of `useSyscall.go` is shown in the following Go code:

```
func main() {
    pid, _, _ := syscall.Syscall(39, 0, 0, 0)
    fmt.Println("My pid is", pid)
    uid, _, _ := syscall.Syscall(24, 0, 0, 0)
    fmt.Println("User ID:", uid)
}
```

In this part, you find out information about the process ID and the user ID using two `syscall.Syscall()` calls. The first parameter of the `syscall.Syscall()` call determines the information that you request.

The third code segment of `useSyscall.go` contains the following Go code:

```
message := []byte{'H', 'e', 'l', 'l', 'o', '!', '\n'}
fd := 1
syscall.Write(fd, message)
```

In this part, you print a message on the screen using `syscall.Write()`. The first parameter is the file descriptor to which you will write, and the second parameter is a byte slice that holds the actual message. The `syscall.Write()` function is portable.

The last part of the `useSyscall.go` program is as follows:

```
fmt.Println("Using syscall.Exec()")
command := "/bin/ls"
env := os.Environ()
```

```

        syscall.Exec(command, []string{"ls", "-a", "-x"}, env)
    }

```

In the last part of the program, you can see how to use the `syscall.Exec()` function to execute an external command. However, you have no control over the output of the command, which is automatically printed on the screen.

Executing `useSyscall.go` on macOS Mojave will generate the following output:

```

$ go run useSyscall.go
My pid is 14602
User ID: 501
Hello!
Using syscall.Exec()
.
funFun.go      functions.go    a.go
htmlT.db       htmlT.go       html.gohtml
ptrFun.go      returnFunction.go manyInit.go
returnPtr.go   text.go        returnNames.go
useAPackage.go useSyscall.go  textT.go

```

Executing the same program on a Debian Linux machine will generate the following output:

```

$ go run useSyscall.go
My pid is 20853
User ID: 0
Hello!
Using syscall.Exec()
.
funFun.go      functions.go    a.go
htmlT.db       htmlT.go       html.gohtml
ptrFun.go      returnFunction.go manyInit.go
returnPtr.go   text.go        returnNames.go
useAPackage.go useSyscall.go  textT.go

```

So, although most of the output is the same as before, the `syscall.Syscall(39, 0, 0, 0)` call does not work on Linux because the user ID of the Linux user is not 0, which means that this command is not portable.

If you want to discover which standard Go packages use the `syscall` package, you can execute the next command from your UNIX shell:

```

$ grep \"syscall\" `find /usr/local/Cellar/go/1.12/libexec/src -name
\"*.go\"`

```

Please replace `/usr/local/Cellar/go/1.12/libexec/src` with the appropriate directory path.

Finding out how `fmt.Println()` really works

If you really want to grasp the usefulness of the `syscall` package, start by reading this subsection. The implementation of the `fmt.Println()` function, as found in <https://golang.org/src/fmt/print.go>, is as follows:

```
func Println(a ...interface{}) (n int, err error) {
    return Fprintln(os.Stdout, a...)
}
```

This means that the `fmt.Println()` function calls `fmt.Fprintln()` to do its job. The implementation of `fmt.Fprintln()`, as found in the same file, is as follows:

```
func Fprintln(w io.Writer, a ...interface{}) (n int, err error) {
    p := newPrinter()
    p.doPrintln(a)
    n, err = w.Write(p.buf)
    p.free()
    return
}
```

This means that the actual writing in `fmt.Fprintln()` is done by the `Write()` function of the `io.Writer` interface. In this case, the `io.Writer` interface is `os.Stdout`, which is defined as follows in <https://golang.org/src/os/file.go>:

```
var (
    Stdin  = NewFile(uintptr(syscall.Stdin), "/dev/stdin")
    Stdout = NewFile(uintptr(syscall.Stdout), "/dev/stdout")
    Stderr = NewFile(uintptr(syscall.Stderr), "/dev/stderr")
)
```

Now look at the implementation of `NewFile()`, which can be found inside https://golang.org/src/os/file_plan9.go:

```
func NewFile(fd uintptr, name string) *File {
    fdi := int(fd)
    if fdi < 0 {
        return nil
    }
    f := &File{&file{fd: fdi, name: name}}
    runtime.SetFinalizer(f.file, (*file).close)
    return f
}
```

```
}
```

When you see a Go source file named `file_plan9.go`, you should suspect that it contains commands specific to a UNIX variant, which means that it contains code that is not portable.

What we have here is the `file` structure type that is embedded in the `File` type, which is the one that is being exported due to its name. So, start looking for functions inside https://golang.org/src/os/file_plan9.go that are applied to a `File` structure or to a pointer to a `File` structure, and that allow you to write data. As the function we are seeking is named `Write()` – look at the implementation of `Fprintln()` – we will have to search all of the source files of the `os` package to find it:

```
$ grep "func (f \*File) Write(" *.go
file.go:func (f \*File) Write(b []byte) (n int, err error) {
```

The implementation of `Write()` as found in <https://golang.org/src/os/file.go> is as follows:

```
func (f \*File) Write(b []byte) (n int, err error) {
    if err := f.checkValid("write"); err != nil {
        return 0, err
    }
    n, e := f.write(b)
    if n < 0 {
        n = 0
    }
    if n != len(b) {
        err = io.ErrShortWrite
    }

    epipecheck(f, e)

    if e != nil {
        err = f.wrapErr("write", e)
    }

    return n, err
}
```

This means that we now have to search for the `write()` function. Searching for the `write` string in https://golang.org/src/os/file_plan9.go reveals the following function inside https://golang.org/src/os/file_plan9.go:

```
func (f \*File) write(b []byte) (n int, err error) {
    if len(b) == 0 {
```



```
        return 0, nil
    }
    return fixCount(syscall.Write(f.fd, b))
}
```

This tells us that a call to the `fmt.Println()` function is implemented using a call to `syscall.Write()`. This underscores how useful and necessary the `syscall` package is.

The go/scanner, go/parser, and go/token packages

This section will talk about the `go/scanner`, `go/parser`, and `go/token` packages, as well as the `go/ast` package. This is low-level information about how Go scans and parses Go code that will help you to understand how Go works. However, you might want to skip this section if low-level things frustrate you.

Parsing a language requires two phases. The first one is about breaking up the input into tokens (**lexical analysis**) and the second one is about feeding the parser with all these tokens in order to make sure that these tokens make sense and are in the right order (**semantic analysis**). Just combining English words does not always create valid sentences.

The go/ast package

An **abstract syntax tree** (AST) is a structured representation of the source code of a Go program. This tree is constructed according to some rules that are specified in the language specification. The `go/ast` package is used for declaring the data types required to represent ASTs in Go. If you want to find out more about an `ast.*` type, the `go/ast` package should be the best place for this kind of information.

The go/scanner package

A **scanner** is something, which in this case will be some Go code, that reads a program written in a programming language, which in this case is Go, and generates tokens.

The `go/scanner` package is used for reading Go programs and generating a series of tokens. The use of the `go/scanner` package will be illustrated in `goScanner.go`, which will be presented in three parts.

The first part of `goScanner.go` is as follows:

```
package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
}
```

The `go/token` package defines constants that represent the lexical tokens of the Go programming language.

The second part of `goScanner.go` contains the following Go code:

```
for _, file := range os.Args[1:] {
    fmt.Println("Processing:", file)
    f, err := ioutil.ReadFile(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    one := token.NewFileSet()
    files := one.AddFile(file, one.Base(), len(f))
}
```

The source file that is going to be tokenized is stored in the `file` variable, whereas its contents are stored in `f`.

The last part of `goScanner.go` is as follows:

```
var myScanner scanner.Scanner
myScanner.Init(files, f, nil, scanner.ScanComments)

for {
    pos, tok, lit := myScanner.Scan()
    if tok == token.EOF {
        break
    }
    fmt.Printf("%s\t%s\t%q\n", one.Position(pos), tok, lit)
```

```

    }
}
}

```

The `for` loop is used for traversing the input file. The end of the source code file is indicated by `token.EOF` – this will exit the `for` loop. The `scanner.Scan()` method returns the current file position, the token, and the literal. The use of `scanner.ScanComments` in `scanner.Init()` tells the scanner to return comments as `COMMENT` tokens. You can use 1 instead of `scanner.ScanComments` and you can put 0 if you do not want to see any `COMMENT` tokens in the output.

Building and executing `goScanner.go` will create the following output:

```

$ ./goScanner a.go
Processing: a.go
a.go:1:1    package "package"
a.go:1:9    IDENT   "a"
a.go:1:10   ;       "\n"
a.go:3:1    import  "import"
a.go:3:8    (       ""
a.go:4:2    STRING  "\"fmt\""
a.go:4:7    ;       "\n"
a.go:5:1    )       ""
a.go:5:2    ;       "\n"
a.go:7:1    func    "func"
a.go:7:6    IDENT   "init"
a.go:7:10   (       ""
a.go:7:11   )       ""
a.go:7:13   {       ""
a.go:8:2    IDENT   "fmt"
a.go:8:5    .       ""
a.go:8:6    IDENT   "Println"
a.go:8:13   (       ""
a.go:8:14   STRING  "\"init() a\""
a.go:8:24   )       ""
a.go:8:25   ;       "\n"
a.go:9:1    }       ""
a.go:9:2    ;       "\n"
a.go:11:1   func    "func"
a.go:11:6   IDENT   "FromA"
a.go:11:11  (       ""
a.go:11:12  )       ""
a.go:11:14  {       ""
a.go:12:2   IDENT   "fmt"
a.go:12:5   .       ""
a.go:12:6   IDENT   "Println"
a.go:12:13  (       ""

```

```

a.go:12:14  STRING  "\"fromA()\""
a.go:12:23  )       ""
a.go:12:24  ;       "\n"
a.go:13:1   }       ""
a.go:13:2   ;       "\n"

```

The output of `goScanner.go` is as simple as it can be. Note that `goScanner.go` can scan any type of file, even binary files. However, if you scan a binary file, you might get less readable output. As you can see from the output, the Go scanner adds semicolons automatically. Note that `IDENT` notifies an identifier, which is the most popular type of token.

The next subsection will deal with the parsing process.

The go/parser package

A parser reads the output of a scanner (tokens) in order to generate a structure from those tokens. Parsers use a grammar that describes a programming language to make sure that the given tokens compose a valid program. That structure is represented as a tree, which is the AST.

The use of the `go/parser` package that processes the output of `go/token` is illustrated in `goParser.go`, which is going to be presented in four parts.

The first part of `goParser.go` is as follows:

```

package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strings"
)

type visitor int

```

The second part of `goParser.go` contains the following Go code:

```

func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }
}

```

```

    fmt.Printf("%sT\n", strings.Repeat("\t", int(v)), n)
    return v + 1
}

```

The `Visit()` method will be called for every node of the AST.

The third part of `goParser.go` is as follows:

```

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }
}

```

The last part of `goParser.go` is as follows:

```

    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        one := token.NewFileSet()
        var v visitor
        f, err := parser.ParseFile(one, file, nil,
parser.AllErrors)
        if err != nil {
            fmt.Println(err)
            return
        }
        ast.Walk(v, f)
    }
}

```

The `Walk()` function, which is called recursively, traverses an AST in depth-first order in order to visit all of its nodes.

Building and executing `goParser.go` to find the AST of a simple and small Go module will generate the following kind of output:

```

$ ./goParser a.go
Processing: a.go
*ast.File
  *ast.Ident
  *ast.GenDecl
    *ast.ImportSpec
      *ast.BasicLit
  *ast.FuncDecl
    *ast.Ident
    *ast.FuncType
      *ast.FieldList
    *ast.BlockStmt

```

```

    *ast.ExprStmt
    *ast.CallExpr
    *ast.SelectorExpr
    *ast.Ident
    *ast.Ident
    *ast.BasicLit
*ast.FuncDecl
*ast.Ident
*ast.FuncType
*ast.FieldList
*ast.BlockStmt
*ast.ExprStmt
*ast.CallExpr
*ast.SelectorExpr
*ast.Ident
*ast.Ident
*ast.BasicLit

```

The output of `goParser.go` is as simple as it gets. However, it is totally different from the output of `goScanner.go`.

Now that you know how the outputs of the Go scanner and the Go parser look, you are ready to see some more practical examples.

A practical example

In this subsection, we are going to write a Go program that counts the number of times a keyword appears in the input files. In this case, the keyword that is going to be counted is `var`. The name of the utility will be `varTimes.go` and it is going to be presented in four parts. The first part of `varTimes.go` is as follows:

```

package main

import (
    "fmt"
    "go/scanner"
    "go/token"
    "io/ioutil"
    "os"
)

var KEYWORD = "var"
var COUNT = 0

```

You can search for any Go keyword you want – you can even set the value of the `KEYWORD` global variable at runtime if you modify `varTimes.go`.

The second part of `varTimes.go` contains the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Not enough arguments!")
        return
    }

    for _, file := range os.Args[1:] {
        fmt.Println("Processing:", file)
        f, err := ioutil.ReadFile(file)
        if err != nil {
            fmt.Println(err)
            return
        }
        one := token.NewFileSet()
        files := one.AddFile(file, one.Base(), len(f))
```

The third part of `varTimes.go` is as follows:

```
var myScanner scanner.Scanner
myScanner.Init(files, f, nil, scanner.ScanComments)

localCount := 0
for {
    _, tok, lit := myScanner.Scan()
    if tok == token.EOF {
        break
    }
```

In this case, the position the token was found in is ignored as it does not matter. However, the `tok` variable is needed for finding out the end of the file.

The last part of `varTimes.go` is as follows:

```
    if lit == KEYWORD {
        COUNT++
        localCount++
    }
    fmt.Printf("Found %s_ %d times\n", KEYWORD, localCount)
}
fmt.Printf("Found %s_ %d times in total\n", KEYWORD, COUNT)
}
```

Compiling and executing `varTimes.go` will create the following kind of output:

```
$ go build varTimes.go
$ ./varTimes varTimes.go variadic.go a.go
Processing: varTimes.go
Found _var_ 3 times
Processing: variadic.go
Found _var_ 0 times
Processing: a.go
Found _var_ 0 times
Found _var_ 3 times in total
```

Finding variable names with a given string length

This subsection will present another practical example that will be more advanced than the one presented in `varTimes.go`. You will see how to find the variable names with a given string length – you can use any string length you want. Additionally, the program will be able to differentiate between **global variables** and **local variables**.



A local variable is defined inside a function whereas a global variable is defined outside of a function. Global variables are also called package variables.

The name of the utility is `varSize.go` and it will be presented in four parts. The first part of `varSize.go` is as follows:

```
package main

import (
    "fmt"
    "go/ast"
    "go/parser"
    "go/token"
    "os"
    "strconv"
)

var SIZE = 2
var GLOBAL = 0
var LOCAL = 0

type visitor struct {
    Package map[*ast.GenDecl]bool
}
```



```

func makeVisitor(f *ast.File) visitor {
    k1 := make(map[*ast.GenDecl]bool)
    for _, aa := range f.Decls {
        v, ok := aa.(*ast.GenDecl)
        if ok {
            k1[v] = true
        }
    }

    return visitor{k1}
}

```

As we want to differentiate between local and global variables, we define two global variables named `GLOBAL` and `LOCAL` to keep these two counts. The use of the `visitor` structure will help us to differentiate between local and global variables, hence the `map` field defined in the `visitor` structure. The `makeVisitor()` method is used to initialize the active `visitor` structure according to the values of its parameter, which is a `File` node representing an entire file.

The second part of `varSize.go` contains the implementation of the `Visit()` method:

```

func (v visitor) Visit(n ast.Node) ast.Visitor {
    if n == nil {
        return nil
    }

    switch d := n.(type) {
    case *ast.AssignStmt:
        if d.Tok != token.DEFINE {
            return v
        }

        for _, name := range d.Lhs {
            v.isItLocal(name)
        }
    case *ast.RangeStmt:
        v.isItLocal(d.Key)
        v.isItLocal(d.Value)
    case *ast.FuncDecl:
        if d.Recv != nil {
            v.CheckAll(d.Recv.List)
        }

        v.CheckAll(d.Type.Params.List)
        if d.Type.Results != nil {
            v.CheckAll(d.Type.Results.List)
        }
    }
}

```

```

    case *ast.GenDecl:
        if d.Tok != token.VAR {
            return v
        }
        for _, spec := range d.Specs {
            value, ok := spec.(*ast.ValueSpec)
            if ok {
                for _, name := range value.Names {
                    if name.Name == "_" {
                        continue
                    }
                    if v.Package[d] {
                        if len(name.Name) == SIZE {
                            fmt.Printf("** %s\n", name.Name)
                            GLOBAL++
                        }
                    } else {
                        if len(name.Name) == SIZE {
                            fmt.Printf("* %s\n", name.Name)
                            LOCAL++
                        }
                    }
                }
            }
        }
    }
    return v
}

```

The main job of the `Visit()` function is to determine the type of the node that it works with in order to act accordingly. This happens with the help of a `switch` statement.

The `ast.AssignStmt` node represents assignments or short variable declarations. The `ast.RangeStmt` node is a structure type for representing a `for` statement with a `range` clause – this is another place where new local variables are declared.

The `ast.FuncDecl` node is a structure type for representing function declarations – every variable that is defined inside a function is a local variable. Lastly, the `ast.GenDecl` node is a structure type for representing an import, constant, type, or variable declaration. However, we are only interested in `token.VAR` tokens.

The third part of `varSize.go` is as follows:

```

func (v visitor) isItLocal(n ast.Node) {
    identifier, ok := n.(*ast.Ident)
    if ok == false {
        return
    }
}

```

```

    }

    if identifier.Name == "_" || identifier.Name == "" {
        return
    }

    if identifier.Obj != nil && identifier.Obj.Pos() == identifier.Pos() {
        if len(identifier.Name) == SIZE {
            fmt.Printf("* %s\n", identifier.Name)
            LOCAL++
        }
    }
}

func (v visitor) CheckAll(fs []*ast.Field) {
    for _, f := range fs {
        for _, name := range f.Names {
            v.isItLocal(name)
        }
    }
}

```

These two functions are helper methods. The first one decides whether an identifier node is a local variable or not, and the second one visits an `ast.Field` node in order to examine its contents for local variables.

The last part of `varSize.go` is as follows:

```

func main() {
    if len(os.Args) <= 2 {
        fmt.Println("Not enough arguments!")
        return
    }

    temp, err := strconv.Atoi(os.Args[1])
    if err != nil {
        SIZE = 2
        fmt.Println("Using default SIZE:", SIZE)
    } else {
        SIZE = temp
    }

    var v visitor
    all := token.NewFileSet()
    for _, file := range os.Args[2:] {
        fmt.Println("Processing:", file)
        f, err := parser.ParseFile(all, file, nil, parser.AllErrors)
        if err != nil {

```

```

        fmt.Println(err)
        continue
    }

    v = makeVisitor(f)
    ast.Walk(v, f)
}
fmt.Printf("Local: %d, Global:%d with a length of %d.\n", LOCAL,
GLOBAL, SIZE)
}

```

The program generates the AST of its input and processes that in order to extract the desired information. Apart from the `Visit()` method, which is part of the interface, the rest of the magic in the `main()` function happens with the help of `ast.Walk()`, which automatically visits all the AST nodes of each file that is being processed.

Building and executing `varSize.go` will generate the following kind of output:

```

$ go build varSize.go
$ ./varSize
Not enough arguments!
$ ./varSize 2 varSize.go variadic.go
Processing: varSize.go
* k1
* aa
* ok
* ok
* ok
* fs
Processing: variadic.go
Local: 6, Global:0 with a length of 2.
$ ./varSize 3 varSize.go variadic.go
Processing: varSize.go
* err
* all
* err
Processing: variadic.go
* sum
* sum
Local: 5, Global:0 with a length of 3.
$ ./varSize 7 varSize.go variadic.go
Processing: varSize.go
Processing: variadic.go
* message
Local: 1, Global:0 with a length of 7.

```

You can remove the various `fmt.Println()` calls in `varSize.go` and have a less cluttered output.

You can do pretty ingenious things in Go once you know how to parse a Go program – you can even write your own parser for your own programming language if you want! If you are really into parsing, you should have a look at the documentation page of the `go/ast` package, as well as its source code, which can be found at

<https://golang.org/pkg/go/ast/> and at

<https://github.com/golang/go/tree/master/src/go/ast> respectively.

Text and HTML templates

The subject of this section will probably surprise you in a good way, because both presented packages give you so much flexibility that I am sure you will find many creative ways to use them. **Templates** are mainly used for separating the formatting part and the data part of the output. Please note that a Go template can be either a file or a string – the general idea is to use inline strings for smaller templates and external files for bigger ones.



You cannot import both `text/template` and `html/template` on the same Go program because these two packages share the same package name (`template`). If absolutely necessary, you should define an alias for one of them. See the `useStrings.go` code in Chapter 4, *The Uses of Composite Types*.

Text output is usually presented on your screen, whereas HTML output is seen with the help of a web browser. However, as text output is usually better than HTML output, if you think that you will need to process the output of a Go utility using other UNIX command-line utilities, you should use `text/template` instead of `html/template`.

Note that both the `text/template` and `html/template` packages are good examples of how sophisticated a Go package can be. As you will see shortly, both packages support their own kind of programming language – good software makes complex things look simple and elegant.

Generating text output

If you need to create plain text output, then using the `text/template` package is a good choice. The use of the `text/template` package will be illustrated in `textT.go`, which will be presented in five parts.

As templates are usually stored in external files, the example presented will use the `text.gotext` template file, which will be analyzed in three parts. Data is typically read from text files or from the Internet. However, for reasons of simplicity, the data for `textT.go` will be hardcoded in the program using a slice.

We will start by looking at the Go code of `textT.go`. The first code portion of `textT.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "text/template"
)
```

The second code segment from `textT.go` is as follows:

```
type Entry struct {
    Number int
    Square int
}
```

You will need to define a new data type for storing your data unless you are dealing with very simplistic data.

The third part of `textT.go` is shown in the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need the template file!")
        return
    }

    tFile := arguments[1]
    DATA := [][]int{{-1, 1}, {-2, 4}, {-3, 9}, {-4, 16}}
```

The `DATA` variable, which is a slice with two dimensions, holds the initial version of your data.

The fourth part of `textT.go` contains the following Go code:

```
var Entries []Entry

for _, i := range DATA {
    if len(i) == 2 {
        temp := Entry{Number: i[0], Square: i[1]}
        Entries = append(Entries, temp)
    }
}
```

The preceding code creates a **slice of structures** from the `DATA` variable.

The last code segment from `textT.go` is the following:

```
t := template.Must(template.ParseGlob(tFile))
t.Execute(os.Stdout, Entries)
}
```

The `template.Must()` function is used for making the required initializations. Its return data type is `Template`, which is a structure that holds the representation of a parsed template. `template.ParseGlob()` reads the external template file. Note that I prefer to use the `gohtml` extension for the template files, but you can use any extension that you want – just be consistent.

Lastly, the `template.Execute()` function does all the work, which includes processing the data and printing the output to the desired file, which in this case is `os.Stdout`.

Now it is time to look at the code of the template file. The first part of the text template file is as follows:

```
Calculating the squares of some integers
```

Note that empty lines in a text template file are significant and will be shown as empty lines in the final output.

The second part of the template is as follows:

```
{{ range . }} The square of {{ printf "%d" .Number }} is {{ printf
"%d" .Square }}
```

There are many interesting things happening here. The `range` keyword allows you to iterate over the lines of the input, which is given as a slice of structures. Plain text is printed as such, whereas variables and dynamic text must begin with `{{` and end with `}}`. The fields of the structure are accessed as `.Number` and `.Square`. Note the dot character in front of the field name of the `Entry` data type. Lastly, the `printf` command is used to format the final output.

The third part of the `text.gotext` file is as follows:

```
{{ end }}
```

A `{{ range }}` command is ended with `{{ end }}`. Accidentally putting `{{ end }}` in the wrong place will affect your output. Once again, keep in mind that empty lines in text template files are significant and will be shown in the final output.

Executing `textT.go` will generate the following type of output:

```
$ go run textT.go text.gotext
Calculating the squares of some integers
The square of -1 is 1
The square of -2 is 4
The square of -3 is 9
The square of -4 is 16
```

Constructing HTML output

This section illustrates the use of the `html/template` package with an example named `htmlT.go`. It will be presented in six parts. The philosophy of the `html/template` package is the same as the `text/template` package. The main difference between these two packages is that `html/template` generates HTML output that is safe against code injection.



Although you can create HTML output with the `text/template` package – after all, HTML is just plain text – if you want to create HTML output, then you should use the `html/template` package instead.

For reasons of simplicity, the following presented will read data from an SQLite database, but you can use any database that you want, provided that you have or you can write the appropriate Go drivers. To make things even easier, the example will populate a database table before reading from it.

The first code portion from `htmlT.go` is as follows:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/mattn/go-sqlite3"
    "html/template"
    "net/http"
    "os"
)

type Entry struct {
    Number int
    Double int
    Square int
}

var DATA []Entry
var tFile string
```

You can see a new package named `net/http` in the `import` block, which is used for creating HTTP servers and clients in Go. You will learn more about network programming in Go and the use of the `net` and `net/http` standard Go packages in Chapter 12, *The Foundations of Network Programming in Go*, and Chapter 13, *Network Programming – Building Your Own Servers and Clients*.

Apart from `net/http`, you can also see the definition of the `Entry` data type that will hold the records read from the SQLite3 table, as well as two global variables named `DATA` and `tFile`, which hold the data that is going to be passed to the template file and the filename of the template file, respectively.

Lastly, you can see the use of the `https://github.com/mattn/go-sqlite3` package for communicating with the SQLite3 database with the help of the `database/sql` interface.

The second part of `htmlT.go` is as follows:

```
func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Host: %s Path: %s\n", r.Host, r.URL.Path)
    myT := template.Must(template.ParseGlob(tFile))
    myT.ExecuteTemplate(w, tFile, DATA)
}
```

The simplicity and the effectiveness of the `myHandler()` function is phenomenal, especially if you consider the size of the function! The `template.ExecuteTemplate()` function does all the work for us. Its first parameter is the variable that holds the connection with the HTTP client, its second parameter is the template file that will be used for formatting the data, and its third parameter is the slice of structures with the data.

The third code segment from `htmlT.go` is shown in the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) != 3 {
        fmt.Println("Need Database File + Template File!")
        return
    }

    database := arguments[1]
    tFile = arguments[2]
```

The fourth code portion from `htmlT.go` is where you start dealing with the database:

```
db, err := sql.Open("sqlite3", database)
if err != nil {
    fmt.Println(nil)
    return
}

fmt.Println("Emptying database table.")
_, err = db.Exec("DELETE FROM data")
if err != nil {
    fmt.Println(nil)
    return
}

fmt.Println("Populating", database)
stmt, _ := db.Prepare("INSERT INTO data(number, double,
square) values(?,?,?)")
for i := 20; i < 50; i++ {
    _, _ = stmt.Exec(i, 2*i, i*i)
}
```

The `sql.Open()` function opens the connection with the desired database. With `db.Exec()`, you can execute database commands without expecting any feedback from them. Lastly, the `db.Prepare()` function allows you to execute a database command multiple times by changing only its parameters and calling `Exec()` afterward.

The fifth part of `htmlT.go` contains the following Go code:

```
rows, err := db.Query("SELECT * FROM data")
if err != nil {
    fmt.Println(nil)
    return
}

var n int
var d int
var s int
for rows.Next() {
    err = rows.Scan(&n, &d, &s)
    temp := Entry{Number: n, Double: d, Square: s}
    DATA = append(DATA, temp)
}
```

In this part of the program, we read the data from the desired table using `db.Query()` and multiple calls to `Next()` and `Scan()`. While reading the data, you put it into a slice of structures and you are done dealing with the database.

The last part of the program is all about setting up the web server, and it contains the following Go code:

```
http.HandleFunc("/", myHandler)
err = http.ListenAndServe(":8080", nil)
if err != nil {
    fmt.Println(err)
    return
}
}
```

Here, the `http.HandleFunc()` function tells the web server embedded in the program which URLs will be supported and by which handler function (`myHandler()`). The current handler supports the `/` URL, which in Go matches all URLs. This saves you from having to create any extra static or dynamic pages.

The code of the `htmlT.go` program is divided into two virtual parts. The first part is about getting the data from the database and putting it into a slice of structures, whereas the second part, which is similar to `textT.go`, is about displaying your data in a web browser.



The two biggest advantages of SQLite are that you do not need to run a server process for the database server and that SQLite databases are stored in self-contained files, which means that single files hold entire SQLite databases.

Note that in order to reduce the Go code and be able to run the `htmlT.go` program multiple times, you will need to create the database table and the SQLite3 database manually, which is as simple as executing the following commands:

```
$ sqlite3 htmlT.db
SQLite version 3.19.3 2017-06-27 16:48:08
Enter ".help" for usage hints.
sqlite> CREATE TABLE data (
...> number INTEGER PRIMARY KEY,
...> double INTEGER,
...> square INTEGER );
sqlite> ^D
$ ls -l htmlT.db
-rw-r--r--  1 mtsouk  staff  8192 Dec 26 22:46 htmlT.db
```

The first command is executed from the UNIX shell and it is needed for creating the database file. The second command is executed from the SQLite3 shell and it has to do with creating a database table named `data`, which has three fields named `number`, `double`, and `square`.

Additionally, you are going to need an external template file, which will be named `html.gohtml`. It is going to be used for the generation of the output of the program.

The first part of `html.gohtml` is as follows:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Doing Maths in Go!</title>
  <style>
    html {
      font-size: 14px;
    }
    table, th, td {
      border: 2px solid blue;
    }
  </style>
</head>
<body>
```

The HTML code that a web browser will get is based on the contents of `html.gohtml`. This means that you will need to create proper HTML output, hence the preceding HTML code, which also includes some inline CSS code for formatting the generated HTML table.

The second part of `html.gohtml` contains the following code:

```
<table>
  <thead>
    <tr>
      <th>Number</th>
      <th>Double</th>
      <th>Square</th>
    </tr>
  </thead>
  <tbody>
    {{ range . }}
      <tr>
        <td> {{ .Number }} </td>
        <td> {{ .Double }} </td>
        <td> {{ .Square }} </td>
      </tr>
    {{ end }}
  </tbody>
</table>
```

As you can see from the preceding code, you still have to use `{{ range }}` and `{{ end }}` in order to iterate over the elements of the slice of structures that was passed to `template.ExecuteTemplate()`. However, this time the `html.gohtml` template file contains lots of HTML code in order to format the data in the slice of structures better.

The last part of the HTML template file is as follows:

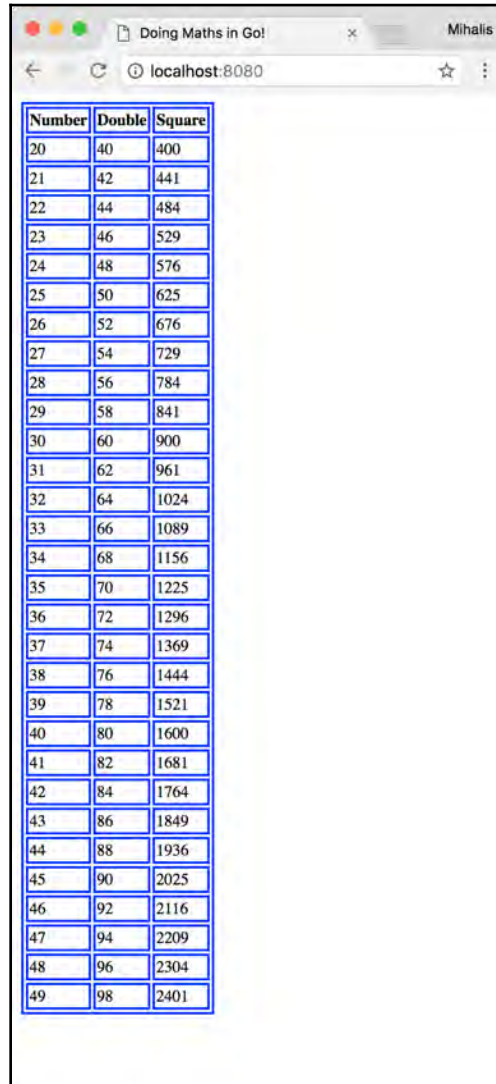
```
</body>
</html>
```

The last part of `html.gohtml` is mainly used for properly ending the generated HTML code according to the HTML standards. Before being able to compile and execute `htmlT.go`, you will need to download the package that will help the Go programming language to communicate with SQLite3. You can do this by executing the following command:

```
$ go get github.com/mattn/go-sqlite3
```

As you already know, you can find the source code of the downloaded package inside `~/go/src` and its compiled version inside `~/go/pkg/darwin_amd64` if you are on a macOS machine. Otherwise, check the contents of `~/go/pkg` to find out your own architecture. Note that the `~` character denotes the home directory of the current user.

Keep in mind that additional Go packages exist that can help you to communicate with an SQLite3 database. However, the one used here is the only one that currently supports the `database/sql` interface. Executing `htmlT.go` will produce the kind of output on a web browser that you can see in the following figure:



The screenshot shows a web browser window titled "Doing Maths in Go!" with the address bar displaying "localhost:8080". The browser shows a table with three columns: "Number", "Double", and "Square". The table contains 30 rows of data, starting from 20 and ending at 49. Each row is highlighted with a blue border.

Number	Double	Square
20	40	400
21	42	441
22	44	484
23	46	529
24	48	576
25	50	625
26	52	676
27	54	729
28	56	784
29	58	841
30	60	900
31	62	961
32	64	1024
33	66	1089
34	68	1156
35	70	1225
36	72	1296
37	74	1369
38	76	1444
39	78	1521
40	80	1600
41	82	1681
42	84	1764
43	86	1849
44	88	1936
45	90	2025
46	92	2116
47	94	2209
48	96	2304
49	98	2401

Figure 6.1: The output of the `htmlT.go` program

Moreover, `htmlT.go` will generate the following type of output in your UNIX shell, which is mainly debugging information:

```
$ go run htmlT.go htmlT.db html.gohtml
Emptying database table.
Populating htmlT.db
Host: localhost:8080 Path: /
Host: localhost:8080 Path: /favicon.ico
Host: localhost:8080 Path: /123
```

If you want to see the HTML output of the program from the UNIX shell, you can use the `wget (1)` utility as follows:

```
$ wget -qO- http://localhost:8080
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Doing Maths in Go!</title>
  <style>
    html {
      font-size: 14px;
    }
    table, th, td {
      border: 2px solid blue;
    }
  </style>
</head>
<body>
<table>
  <thead>
    <tr>
```

Both `text/template` and `html/template` are powerful packages that can save you a lot of time, so I suggest that you use them when they fit the requirements of your applications.

Additional resources

You will find the next resources very useful:

- Visit the documentation page of the `syscall` standard Go package at <https://golang.org/pkg/syscall/>. This is one of the biggest Go documentation pages that I have ever seen!

- Visit the documentation page of the `text/template` package, which can be found at <https://golang.org/pkg/text/template/>.
- Similarly, go to <https://golang.org/pkg/html/template/> for the documentation of the `html/template` package.
- You can find out more about the `go/token` package at <https://golang.org/pkg/go/token/>.
- You can find out more about the `go/parser` package at <https://golang.org/pkg/go/parser/>.
- You can find out more about the `go/scanner` package at <https://golang.org/pkg/go/scanner/>.
- You can find out more about the `go/ast` package at <https://golang.org/pkg/go/ast/>.
- Visit the home page of SQLite3 at <https://www.sqlite.org/>.
- Watch the "Writing Beautiful Packages in Go" video by Mat Ryer at <https://www.youtube.com/watch?v=cAWlv2SeQus>.
- If you want to know about **Plan 9**, look at <https://plan9.io/plan9/>.
- Take the time to look at the `find(1)` command-line tool by visiting its man page (`man 1 find`).

Exercises

- Seek out more information about the actual implementation of the `fmt.Printf()` function.
- Can you write a function that sorts three `int` values? Try to write two versions of the function: one with named returned values and another without named return values. Which one do you think is better?
- Can you modify the Go code of `htmlT.go` in order to use `text/template` instead of `html/template`?
- Can you modify the Go code of `htmlT.go` in order to use either <https://github.com/feyeleonor/gosqlite3> or the <https://github.com/phf/go-sqlite3> package for communicating with the SQLite3 database?
- Create your own Go module and develop three major versions of it.
- Write a Go program like `htmlT.go` that reads data from a MySQL database. Write down the code changes that you made.

Summary

This chapter presented three primary topics: Go functions, Go packages, and Go modules. The main advantage of Go modules is that they record the exact dependency requirements, which makes creating reproducible builds easy and straightforward.

This chapter also offered you ample advice about developing good Go packages. It subsequently talked about the `text/template` and `html/template` packages, which allow you to create plain text and HTML output based on templates, as well as the `go/token`, `go/parser`, and `go/scanner` packages. Lastly, it talked about the `syscall` standard Go package that offers advanced features.

The next chapter will discuss two important Go features: interfaces and reflection. Additionally, it will talk about object-oriented programming in Go, debugging, and Go type methods. All of these topics are pretty advanced and you might find them difficult at first. However, learning more about them will unquestionably make you a better Go programmer.

Finally, the next chapter includes a quick introduction to the `git` utility, which was used in this chapter to create Go modules.

7 Reflection and Interfaces for All Seasons

The previous chapter talked about developing packages, modules, and functions in Go, as well as working with text and HTML templates with the help of the `text/template` and `html/template` packages. The chapter also explained the use of the `syscall` package.

This chapter is going to teach you three very interesting, handy, and somewhat advanced Go concepts: **reflection**, **interfaces**, and **type** methods. Although Go interfaces are used all the time, reflection is not, mainly because it is not usually necessary for your programs to use it. Furthermore, you will learn about type assertions, the Delve debugger, and **object-oriented programming** in Go. Lastly, this chapter will present a short introduction to **git** and **GitHub**.

Therefore, in this chapter, you will learn about:

- Type methods
- Go interfaces
- Type assertions
- Developing and using your own interfaces
- GitHub and git
- An introduction to the Delve debugger
- Object-oriented programming in Go
- Reflection and the `reflect` standard Go package
- Reflection and the `reflectwalk` library

Type methods

A Go type method is a function with a special receiver argument. You declare methods as ordinary functions with an additional parameter that appears in front of the function name. This particular parameter connects the function to the type of that extra parameter. As a result, that parameter is called the **receiver of the method**.

The following Go code is the implementation of the `Close()` function as found in https://golang.org/src/os/file_plan9.go:

```
func (f *File) Close() error {
    if err := f.checkValid("close"); err != nil {
        return err
    }
    return f.file.close()
}
```

The `Close()` function is a type method because there is that `(f *File)` parameter in front of its name and after the `func` keyword. The `f` parameter is called the receiver of the method. In object-oriented programming terminology this process can be described as sending a message to an **object**. In Go, the receiver of a method is defined using a regular variable name without the need to use a dedicated keyword such as `this` or `self`.

Now let me offer you a complete example using the Go code of the `methods.go` file, which will be presented in four parts.

The first part of `methods.go` comes with the following Go code:

```
package main

import (
    "fmt"
)

type twoInts struct {
    X int64
    Y int64
}
```

In the preceding Go code, you can see the definition of a new structure with two fields named `twoInts`.

The second code segment of `methods.go` is next:

```
func regularFunction(a, b twoInts) twoInts {  
    temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}  
    return temp  
}
```

In this part, you define a new function named `regularFunction()` that accepts two parameters of type `twoInts` and returns just one `twoInts` value.

The third part of the program contains the next Go code:

```
func (a twoInts) method(b twoInts) twoInts {  
    temp := twoInts{X: a.X + b.X, Y: a.Y + b.Y}  
    return temp  
}
```

The `method()` function is equivalent to the `regularFunction()` function defined in the previous part of `methods.go`. However, the `method()` function is a type method and you are going to learn a different way of calling it in a moment.



The really interesting thing here is that the implementation of `method()` is exactly the same as the implementation of `regularFunction()`!

The last code segment of `methods.go` is the following:

```
func main() {  
    i := twoInts{X: 1, Y: 2}  
    j := twoInts{X: -5, Y: -2}  
    fmt.Println(regularFunction(i, j))  
    fmt.Println(i.method(j))  
}
```

As you can see, the way you call a type method (`i.method(j)`) is different from the way you call a conventional function (`regularFunction(i, j)`).

Executing `methods.go` will create the next output:

```
$ go run methods.go  
{-4 0}  
{-4 0}
```

Notice that type methods are also associated with interfaces, which will be the subject of the next section. As a result, you will see more type methods later.

Go interfaces

Strictly speaking, a Go interface type defines the behavior of other types by specifying a set of methods that need to be implemented. For a type to satisfy an interface, it needs to implement all the methods required by that interface, which are usually not too many.

Putting it simply, interfaces are **abstract types** that define a set of functions that need to be implemented so that a type can be considered an instance of the interface. When this happens, we say that the type satisfies this interface. So, an interface is two things: a set of methods and a type, and it is used to define the behavior of other types.

The biggest advantage you get from having and using an interface is that you can pass a variable of a type that implements that particular interface to any function that expects a parameter of that specific interface. Without that amazing capability, interfaces would have been only a formality without any practical or real benefit.



Please note that if you find yourself defining an interface and its implementation in the same Go package, you might need to rethink your approach. This is not because this is technically wrong, but because it looks logically wrong.

Two very common Go interfaces are `io.Reader` and `io.Writer` and they are used in file input and output operations. More specifically, `io.Reader` is used for reading from a file, whereas `io.Writer` is used for writing to a file of any type.

The definition of `io.Reader` as found in <https://golang.org/src/io/io.go> is the following:

```
type Reader interface {  
    Read(p []byte) (n int, err error)  
}
```

So, in order for a type to satisfy the `io.Reader` interface, you will need to implement the `Read()` method as described in the interface definition.

Similarly, the definition of `io.Writer` as found in <https://golang.org/src/io/io.go> is next:

```
type Writer interface {
```

```
    Write(p []byte) (n int, err error)
}
```

To satisfy the `io.Writer` interface, you will just need to implement a single method named `Write()`.

Each of the `io.Reader` and `io.Writer` interfaces requires the implementation of just one method. Yet both interfaces are very powerful – most likely their power comes from their simplicity. Generally speaking, most interfaces are fairly simple.

In the next subsections, you will learn how to define an interface on your own and how to use it in other Go packages. Notice that it is not necessary for an interface to be fancy or impressive as long as it does what you want it to do.



Putting it simply, interfaces should be utilized when there is a need for making sure that certain conditions will be met and certain behaviors will be anticipated from a Go element.

About type assertions

A type assertion is the `x.(T)` notation, where `x` is an interface type and `T` is a type. Additionally, the actual value stored in `x` is of type `T` and `T` must satisfy the interface type of `x`. The following paragraphs, as well as the code example, will help you to clarify this relatively eccentric definition of a type assertion.

Type assertions help you to do *two things*. The first thing is checking whether an interface value keeps a particular type. When used this way, a type assertion returns two values: the underlying value and a `bool` value. Although the underlying value is what you might want to use, the Boolean value tells you whether the type assertion was successful or not.

The second thing that type assertions help with is allowing you to use the concrete value stored in an interface or assign it to a new variable. This means that if there is an `int` variable in an interface, you can get that value using a type assertion.

However, if a type assertion is not successful and you do not handle that failure on your own, your program will panic. Look at the Go code of the `assertion.go` program, which will be presented in two parts. The first part contains the following Go code:

```
package main

import (
    "fmt"
```

```
)

func main() {
    var myInt interface{} = 123

    k, ok := myInt.(int)
    if ok {
        fmt.Println("Success:", k)
    }

    v, ok := myInt.(float64)
    if ok {
        fmt.Println(v)
    } else {
        fmt.Println("Failed without panicking!")
    }
}
```

First, you declare the `myInt` variable that has a dynamic type `int` and value `123`. Then, you use a type assertion twice to testing the interface of the `myInt` variable – once for `int` and once for `float64`.

As the `myInt` variable does not contain a `float64` value, the `myInt.(float64)` type assertion will fail unless handled properly. Fortunately, in this case, the correct use of the `ok` variable will save your program from panicking.

The second part comes with the next Go code:

```
i := myInt.(int)
fmt.Println("No checking:", i)

j := myInt.(bool)
fmt.Println(j)
}
```

There are two type assertions taking place here. The first type assertion is successful, so there will be no problem with that. But let me talk a little bit more about this particular type assertion. The type of variable `i` will be `int` and its value will be `123`, which is the value stored in `myInt`. So, as `int` satisfies the `myInt` interface, which in this case happens because the `myInt` interface requires no functions to be implemented, the value of `myInt.(int)` is an `int` value.

However, the second type assertion, which is `myInt.(bool)`, will trigger a panic because the underlying value of `myInt` is not `Boolean` (`bool`).

Therefore, executing `assertion.go` will generate the following output:

```
$ go run assertion.go
Success: 123
Failed without panicking!
No checking: 123
panic: interface conversion: interface {} is int, not bool
goroutine 1 [running]:
main.main()
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch07/assertion.go:25 +0x1c1
exit status 2
```

Go states pretty clearly the reason for panicking: `interface {} is int, not bool`.

Generally speaking, when using interfaces, expect to use type assertions as well. You will see more type assertions in the `useInterface.go` program coming up.

Writing your own interfaces

In this section, you will learn how to develop your own interfaces, which is a relatively easy process as long as you know what you want to develop.

The technique is going to be illustrated using the Go code of `myInterface.go`, which will be presented shortly. The interface that is going to be created will help you to work with geometric shapes of the plane.

The Go code of `myInterface.go` is next:

```
package myInterface

type Shape interface {
    Area() float64
    Perimeter() float64
}
```


The definition of the `shape` interface is truly straightforward as it requires that you implement just two functions, named `Area()` and `Perimeter()`, that both return a `float64` value. The first function will be used to calculate the area of a shape in the plane and the second one to calculate the perimeter of a shape in the plane.

After that, you will need to install the `myInterface.go` package and make it available to the current user. As you already know, the installation process involves the execution of the following UNIX commands:

```
$ mkdir ~/go/src/myInterface
$ cp myInterface.go ~/go/src/myInterface
$ go install myInterface
```

Using a Go interface

This subsection will teach you how to use the interface defined in `myInterface.go` in a Go program named `useInterface.go`, which will be presented in five parts.

The first part of `useInterface.go` comes with the next Go code:

```
package main

import (
    "fmt"
    "math"
    "myInterface"
)

type square struct {
    X float64
}

type circle struct {
    R float64
}
```

As the desired interface is defined in its own package, it should come as no surprise that you are importing the `myInterface` package.

The second code portion from `useInterface.go` contains the following code:

```
func (s square) Area() float64 {
    return s.X * s.X
}

func (s square) Perimeter() float64 {
    return 4 * s.X
}
```

In this part, you implement the shape interface for the `square` type.

The third part contains the next Go code:

```
func (s circle) Area() float64 {
    return s.R * s.R * math.Pi
}

func (s circle) Perimeter() float64 {
    return 2 * s.R * math.Pi
}
```

In this part, you implement the shape interface for the `circle` type.

The fourth part of `useInterface.go` comes with the following Go code:

```
func Calculate(x myInterface.Shape) {
    _, ok := x.(circle)
    if ok {
        fmt.Println("Is a circle!")
    }

    v, ok := x.(square)
    if ok {
        fmt.Println("Is a square:", v)
    }

    fmt.Println(x.Area())
    fmt.Println(x.Perimeter())
}
```

So, in the preceding code, you implement one function that requires a single shape parameter (`myInterface.Shape`). The magic here should be more obvious once you understand that it requires any shape parameter, which is any parameter whose type implements the shape interface.

The code at the beginning of the function shows how you can differentiate between the various data types that implement the desired interface. In the second block, you can see how you can find out the values stored in a `square` parameter – you can use this technique for any type that implements the `myInterface.Shape` interface.

The last code segment includes the next code:

```
func main() {
    x := square{X: 10}
    fmt.Println("Perimeter:", x.Perimeter())
    Calculate(x)
    y := circle{R: 5}
    Calculate(y)
}
```

In this part, you can see how you can use both `circle` and `square` variables as parameters to the `Calculate()` function you implemented earlier.

If you execute `useInterface.go`, you will get the next output:

```
$ go run useInterface.go
Perimeter: 40
Is a square: {10}
100
40
Is a circle!
78.53981633974483
31.41592653589793
```

Using switch with interface and data types

In this subsection, you will learn how to use the `switch` statement to differentiate between different data types using the Go code of `switch.go`, which will be presented in four parts. The Go code of `switch.go` is partially based on `useInterface.go`, but it will add another type named `rectangle` and will not need to implement the methods of any interface.

The first part of the program is next:

```
package main

import (
    "fmt"
)
```

As the code in `switch.go` will not work with the interface defined in `myInterface.go`, there is no need to import the `myInterface` package.

The second part is where you define the three new data types that will be used in the program:

```
type square struct {
    X float64
}

type circle struct {
    R float64
}

type rectangle struct {
    X float64
    Y float64
}
```

All three types are pretty simple.

The third code segment from `switch.go` comes with the following Go code:

```
func tellInterface(x interface{}) {
    switch v := x.(type) {
    case square:
        fmt.Println("This is a square!")
    case circle:
        fmt.Printf("%v is a circle!\n", v)
    case rectangle:
        fmt.Println("This is a rectangle!")
    default:
        fmt.Printf("Unknown type %T!\n", v)
    }
}
```

Here you can see the implementation of a function named `tellInterface()` with a single parameter named `x` and `type interface{}`.

This trick will help you to differentiate between the different data types of the `x` parameter. All the magic is performed with the use of the `x.(type)` statement that returns the type of the `x` element. The `%v` verb used in `fmt.Printf()` allows you to acquire the value of the type.

The last part of `switch.go` contains the implementation of the `main()` function:

```
func main() {
    x := circle{R: 10}
    tellInterface(x)
    y := rectangle{X: 4, Y: 1}
    tellInterface(y)
    z := square{X: 4}
    tellInterface(z)
    tellInterface(10)
}
```

Executing `switch.go` will generate the next kind of output:

```
$ go run switch.go
{10} is a circle!
This is a rectangle!
This is a square!
Unknown type int!
```

Reflection

Reflection is an advanced Go feature that allows you to dynamically learn the type of an arbitrary object, as well as information about its structure. Go offers the `reflect` package for working with reflection. What you should remember is that you will most likely not need to use reflection in every Go program. So, the first two questions are: why is reflection necessary and when should you use it?

Reflection is necessary for the implementation of packages such as `fmt`, `text/template`, and `html/template`. In the `fmt` package, reflection saves you from having to explicitly deal with every data type that exists. However, even if you had the patience to write code to work with every data type that you know of, you would still not be able to work with all possible types! In this case, reflection makes it possible for the methods of the `fmt` package to find the structure and to work with new types.

Therefore, you might need to use reflection when you want to be as generic as possible or when you want to make sure that you will be able to deal with data types that do not exist at the time of writing your code but might exist in the future. Additionally, reflection is handy when working with values of types that do not implement a common interface.



As you can see, reflection helps you to work with unknown types and unknown values of types. However, that flexibility comes at a cost.

The stars of the `reflect` package are two types named `reflect.Value` and `reflect.Type`. The former type is used for storing values of any type, whereas the latter type is used for representing Go types.

A simple reflection example

This section will present you with a relatively simple reflection example in order to help you to feel comfortable with this advanced Go feature.

The name of the Go source file is `reflection.go` and it will be presented to you in four parts. The purpose of `reflection.go` is to examine an "unknown" structure variable and find out more about it at runtime. In order to be more interesting, the program will define two new `struct` types. Based on these two types, it will also define two new variables; however, it will only examine one of them.

If the program has no command-line arguments, it will examine the first one, otherwise it will explore the second one – practically, this means that the program will not know in advance (at runtime) the kind of `struct` variable it will have to process.

The first part of `reflection.go` contains the next Go code:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type a struct {
    X int
    Y float64
    Z string
}
```

```
}

type b struct {
    F int
    G int
    H string
    I float64
}
```

In this part, you can see the definition of the `struct` data types that will be used in the program.

The second code segment from `reflection.go` is the following:

```
func main() {
    x := 100
    xRefl := reflect.ValueOf(&x).Elem()
    xType := xRefl.Type()
    fmt.Printf("The type of x is %s.\n", xType)
```

The preceding Go code presents a small and naive reflection example. First you declare a variable named `x` and then you call the `reflect.ValueOf(&x).Elem()` function. Then you call `xRefl.Type()` in order to get the type of the variable, which is stored in `xType`. These three lines of code illustrate how you can get the data type of a variable using reflection. However, if all you care about is the data type of a variable, you can just call `reflect.TypeOf(x)` instead.

The third code portion from `reflection.go` contains the next Go code:

```
A := a{100, 200.12, "Struct a"}
B := b{1, 2, "Struct b", -1.2}
var r reflect.Value

arguments := os.Args
if len(arguments) == 1 {
    r = reflect.ValueOf(&A).Elem()
} else {
    r = reflect.ValueOf(&B).Elem()
}
```

In this part, you declare two variables named `A` and `B`. The type of the `A` variable is `a` and the type of the `B` variable is `b`. The type of the `r` variable should be `reflect.Value` because this is what the `reflect.ValueOf()` function returns. The `Elem()` method returns the value contained in the reflection interface (`reflect.Value`).

The last part of `reflection.go` is next:

```
iType := r.Type()
fmt.Printf("i Type: %s\n", iType)
fmt.Printf("The %d fields of %s are:\n", r.NumField(), iType)

for i := 0; i < r.NumField(); i++ {
    fmt.Printf("Field name: %s ", iType.Field(i).Name)
    fmt.Printf("with type: %s ", r.Field(i).Type())
    fmt.Printf("and value %v\n", r.Field(i).Interface())
}
}
```

In this part of the program, you use the appropriate functions of the `reflect` package in order to obtain the desired information. The `NumField()` method returns the number of fields in a `reflect.Value` structure, whereas the `Field()` function returns the field of the structure that is specified by its parameter. The `Interface()` function returns the value of a field of the `reflect.Value` structure as an interface.

Executing `reflection.go` twice will generate the following output:

```
$ go run reflection.go 1
The type of x is int.
i Type: main.b
The 4 fields of main.b are:
Field name: F with type: int and value 1
Field name: G with type: int and value 2
Field name: H with type: string and value Struct b
Field name: I with type: float64 and value -1.2
$ go run reflection.go
The type of x is int.
i Type: main.a
The 3 fields of main.a are:
Field name: X with type: int and value 100
Field name: Y with type: float64 and value 200.12
Field name: Z with type: string and value Struct a
```

It is important to note that Go uses its internal representation to print the data types of variables `A` and `B`, which are `main.a` and `main.b`, respectively. However, this is not the case with variable `x`, which is an `int`.

A more advanced reflection example

In this section, we are going to see more advanced uses of reflection illustrated in relatively small code blocks using the Go code of `advRef1.go`.

The `advRef1.go` program will be presented in five parts – its first part is next:

```
package main

import (
    "fmt"
    "os"
    "reflect"
)

type t1 int
type t2 int
```

Note that although both `t1` and `t2` types are based on `int`, and therefore are essentially the same type as `int`, Go treats them as totally different types. Their internal representation after Go parses the code of the program will be `main.t1` and `main.t2`, respectively.

The second code portion from `advRef1.go` is the following:

```
type a struct {
    X    int
    Y    float64
    Text string
}

func (a1 a) compareStruct(a2 a) bool {
    r1 := reflect.ValueOf(&a1).Elem()
    r2 := reflect.ValueOf(&a2).Elem()

    for i := 0; i < r1.NumField(); i++ {
        if r1.Field(i).Interface() != r2.Field(i).Interface() {
            return false
        }
    }
    return true
}
```

In this code, you define a Go structure type named `a` and implement a Go function named `compareStruct()`. The purpose of this function is to find out whether two variables of the `a` type are exactly the same or not. As you can see, `compareStruct()` uses Go code from `reflection.go` to perform its task.

The third code segment of `advRef1.go` comes with the following Go code:

```
func printMethods(i interface{}) {
    r := reflect.ValueOf(i)
    t := r.Type()
```

```

    fmt.Printf("Type to examine: %s\n", t)

    for j := 0; j < r.NumMethod(); j++ {
        m := r.Method(j).Type()
        fmt.Println(t.Method(j).Name, "-->", m)
    }
}

```

The `printMethods()` function prints the methods of a variable. The variable type that will be used in `advRef1.go` to illustrate `printMethods()` will be `os.File`.

The fourth code segment from `advRef1.go` contains the following Go code:

```

func main() {
    x1 := t1(100)
    x2 := t2(100)
    fmt.Printf("The type of x1 is %s\n", reflect.TypeOf(x1))
    fmt.Printf("The type of x2 is %s\n", reflect.TypeOf(x2))

    var p struct{}
    r := reflect.New(reflect.ValueOf(&p).Type()).Elem()
    fmt.Printf("The type of r is %s\n", reflect.TypeOf(r))
}

```

The last code portion of `advRef1.go` is as follows:

```

a1 := a{1, 2.1, "A1"}
a2 := a{1, -2, "A2"}

if a1.compareStruct(a1) {
    fmt.Println("Equal!")
}

if !a1.compareStruct(a2) {
    fmt.Println("Not Equal!")
}

var f *os.File
printMethods(f)
}

```

As you will see later, the `a1.compareStruct(a1)` call returns `true` because we are comparing `a1` with itself, whereas the `a1.compareStruct(a2)` call will return `false` because the `a1` and `a2` variables have different values.

Executing `advRef1.go` will create the following output:

```

$ go run advRef1.go
The type of x1 is main.t1

```

```
The type of x2 is main.t2
The type of r is reflect.Value
Equal!
Not Equal!
Type to examine: *os.File
Chdir --> func() error
Chmod --> func(os.FileMode) error
Chown --> func(int, int) error
Close --> func() error
Fd --> func() uintptr
Name --> func() string
Read --> func([]uint8) (int, error)
ReadAt --> func([]uint8, int64) (int, error)
Readdir --> func(int) ([]os.FileInfo, error)
Readdirnames --> func(int) ([]string, error)
Seek --> func(int64, int) (int64, error)
Stat --> func() (os.FileInfo, error)
Sync --> func() error
Truncate --> func(int64) error
Write --> func([]uint8) (int, error)
WriteAt --> func([]uint8, int64) (int, error)
WriteString --> func(string) (int, error)
```

You can see that the type of the `r` variable, which is returned by `reflect.New()`, will be `reflect.Value`. Additionally, the output of the `printMethods()` method tells us that the `*os.File` type supports a plethora of methods, such as `Chdir()` and `Chmod()`.

The three disadvantages of reflection

Without a doubt, reflection is a powerful Go feature. However, as with all tools, reflection should be used rationally for three main reasons. The first reason is that extensive use of reflection will make your programs hard to read and maintain. A potential solution to this problem is good documentation, but developers are famous for not having the time to write the required documentation.

The second reason is that the Go code that uses reflection will make your programs slower. Generally speaking, Go code that is made to work with a particular data type will always be faster than Go code that uses reflection to dynamically work with any Go data type. Additionally, such dynamic code will make it difficult for tools to refactor or analyze your code.

The last reason is that reflection errors cannot be caught at build time and are reported at runtime as panics, which means that reflection errors can potentially crash your programs.

This can happen months or even years after the development of a Go program! One solution to this problem is extensive testing before a dangerous function call. However, this will add even more Go code to your programs, which will make them even slower.

The reflectwalk library

The `reflectwalk` library allows you to walk complex values in Go using reflection in a way that is similar to the way you walk a filesystem. The following example, which is going to walk a structure, is called `walkRef.go` and it is going to be presented in five parts.

The first part of `walkRef.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/mitchellh/reflectwalk"
    "reflect"
)

type Values struct {
    Extra map[string]string
}
```

As `reflectwalk` is not a standard Go package, you will need to call it using its full address.

The second part of `walkRef.go` is as follows:

```
type WalkMap struct {
    MapVal reflect.Value
    Keys   map[string]bool
    Values map[string]bool
}

func (t *WalkMap) Map(m reflect.Value) error {
    t.MapVal = m
    return nil
}
```

The `Map()` function is required by an interface that is defined in `reflectwalk` and used for searching maps.

The third part of `walkRef.go` is as follows:

```
func (t *WalkMap) MapElem(m, k, v reflect.Value) error {
    if t.Keys == nil {
        t.Keys = make(map[string]bool)
        t.Values = make(map[string]bool)
    }

    t.Keys[k.Interface().(string)] = true
    t.Values[v.Interface().(string)] = true
    return nil
}
```

The fourth part of `walkRef.go` is as follows:

```
func main() {
    w := new(WalkMap)

    type S struct {
        Map map[string]string
    }

    data := &S{
        Map: map[string]string{
            "V1": "v1v",
            "V2": "v2v",
            "V3": "v3v",
            "V4": "v4v",
        },
    }

    err := reflectwalk.Walk(data, w)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Here, you define a new variable named `data` that holds the map and you call `reflectwalk.Walk()` to learn more about it.

The last part of `walkRef.go` is as follows:

```
r := w.MapVal
fmt.Println("MapVal:", r)
rType := r.Type()
fmt.Printf("Type of r: %s\n", rType)

for _, key := range r.MapKeys() {
```

```
        fmt.Println("key:", key, "value:", r.MapIndex(key))
    }
}
```

The last part of `walkRef.go` shows how to use reflection to print the contents of the `MapVal` field of the `WalkMap` structure. The `MapKeys()` method returns a slice of `reflect.Value`s – each value holds a single map key. The `MapIndex()` method allows you to print the value of a key. The `MapKeys()` and `MapIndex()` methods only work with the `reflect.Map` type and allow you to iterate over a map – the order of the returned map elements will be random.

Before using the `reflectwalk` library for the first time, you will need to download it, which can be done as follows:

```
$ go get github.com/mitchellh/reflectwalk
```



If you decide to use Go modules, the process of downloading the `reflectwalk` library will be much simpler and automated.

Executing `walkRef.go` will generate the following output:

```
$ go run walkRef.go
MapVal: map[V1:v1v V2:v2v V3:v3v V4:v4v]
Type of r: map[string]string
key: V2 value: v2v
key: V3 value: v3v
key: V4 value: v4v
key: V1 value: v1v
```

Object-oriented programming in Go

You should know by now that Go does not use inheritance; instead, it supports composition. Go interfaces provide a kind of polymorphism. So, although Go is not an object-oriented programming language, it has some features that allow us to mimic object-oriented programming.



If you really want to develop applications using the object-oriented methodology, then choosing Go might not be your best option. As I am not really into **Java**, I would suggest looking at **C++** or **Python** instead.

First, let me explain to you the two techniques that will be used in the Go program of this section. The first technique uses methods in order to associate a function with a type, which means that in some ways, the function and the type construct an object. In the second technique, you embed a type into a new structure type in order to create a kind of hierarchy.

There is also a third technique where you use a Go interface to make two or more elements *objects of the same class*. This technique is not going to be illustrated in this section as it was shown earlier in this chapter.

The key point here is that a Go interface allows you to define a common behavior between different elements such that all these different elements share the characteristics of an object. This might permit you to say that these different elements are objects of the same class; however, objects and classes of an actual object-oriented programming language can do many more things.

The first two techniques will be illustrated in `ooo.go`, which will be presented in four parts. The first code segment from `ooo.go` contains the next Go code:

```
package main

import (
    "fmt"
)

type a struct {
    XX int
    YY int
}

type b struct {
    AA string
    XX int
}
```

The second part of the program is the following:

```
type c struct {
    A a
    B b
}
```

So, composition allows you to create a structure in your Go elements using multiple `struct` types. In this case, data type `C` groups an `a` variable and a `b` variable.

The third portion of `ooo.go` comes with the next Go code:

```
func (A a) A() {
    fmt.Println("Function A() for A")
}

func (B b) A() {
    fmt.Println("Function A() for B")
}
```

The two methods defined here can have the same name (`A()`) because they have different function headers – the first one works with `a` variables, whereas the second one works with `b` variables. This technique allows you to share the same function name between multiple types.

The last part of `ooo.go` is next:

```
func main() {
    var i c
    i.A.A()
    i.B.A()
}
```

All the Go code in `ooo.go` is pretty simplistic compared to the code of an object-oriented programming language that would implement abstract classes and inheritance. However, it is more than adequate for generating types and elements with a structure in them, as well as for having different data types with the same method names.

Executing `ooo.go` will generate the following output:

```
$ go run ooo.go
Function A() for A
Function A() for B
```

Nevertheless, as the following code illustrates, composition is not inheritance and the first type knows nothing about the changes made to the `shared()` function by the second type:

```
package main

import (
    "fmt"
)

type first struct{}
```



```
func (a first) F() {
    a.shared()
}

func (a first) shared() {
    fmt.Println("This is shared() from first!")
}

type second struct {
    first
}

func (a second) shared() {
    fmt.Println("This is shared() from second!")
}

func main() {
    first{}.F()
    second{}.shared()
    i := second{}
    j := i.first
    j.F()
}
```

Please note that the `second` type embeds the `first` type and that the two types share a function named `shared()`.

Saving the former Go code as `goCoIn.go` and executing it will generate the following output:

```
$ go run goCoIn.go
This is shared() from first!
This is shared() from second!
This is shared() from first!
```

Although the calls to `first{}.F()` and `second{}.shared()` generate the expected results, the call to `j.F()` still calls `first.shared()` instead of the `second.shared()` function despite the fact that the `second` type changes the implementation of the `shared()` function. This is called **method overriding** in object-oriented terminology.

Note that the `j.F()` call can be written as `(i.first).F()` or as `(second{}.first).F()` without the need to define too many variables. Breaking it into three lines of code makes it a little easier to understand.

An introduction to git and GitHub

GitHub is a website and service for storing and building software. You can work on GitHub using its graphical user interface or using command-line utilities. On the other hand, `git` (1) is a command-line utility that can do many things, including working with GitHub repositories.



An alternative to GitHub is **GitLab**. Most, if not all, of the presented `git` (1) commands and options will work for communicating with GitLab without any modifications.

This section will offer you a quick introduction to `git` (1) and its most common and frequently used commands.

Using git

Note that `git` (1) has a huge number of commands and options that you do not need to use on a daily basis. In this subsection, I am going to present you with the most useful and popular `git` (1) commands in my experience and based on the way I work.

Please note that in order to get an existing GitHub repository onto your local computer, you will need to use the `git clone` command followed by the URL of the repository:

```
$ git clone git@github.com:mactsouk/go-kafka.git
Cloning into 'go-kafka'...
remote: Enumerating objects: 13, done.
remote: Counting objects: 100% (13/13), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 13 (delta 4), reused 10 (delta 4), pack-reused 0
Receiving objects: 100% (13/13), done.
Resolving deltas: 100% (4/4), done.
```

The git status command

The `git status` command shows the status of the working tree. If everything is in sync, `git status` will return an output similar to the following:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
nothing to commit, working tree clean
```

If there are changes, the output of `git status` will look similar to the following:

```
On branch master
Your branch is up to date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)
    modified:   main.go
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    newFile.go
no changes added to commit (use "git add" and/or "git commit -a")
```

The git pull command

The `git pull` command is used to get updates from the remote repository. This is especially useful when multiple people are working on the same repository or if you are working from multiple machines.

The git commit command

The `git commit` command is for recording changes to the repository. After a `git commit` command, you will most likely need to issue a `git push` command to send the changes to the remote repository. A very common way to execute the `git commit` command is the following:

```
$ git commit -a -m "Commit message"
```

The `-m` option specifies that the message that will go with the commit, whereas the `-a` option tells `git commit` to automatically include all modified files. Please note that this will exclude new files that need to be added first using `git add`.

The git push command

For local changes to be transferred to the GitHub repository, you will need to execute the `git push` command. The output of the `git push` command is similar to the following:

```
$ touch a_file.go
$ git add a_file.go
$ git commit -a -m "Adding a new file"
[master 782c4da] Adding a new file
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 ch07/a_file.go
```

```
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 98f8a77..782c4da master -> master
```

Working with branches

A branch offers a way to manage your workflow and separate changes from the main branch. Each repository has a default branch, which is usually called the `master` branch, and potentially multiple other branches.

You can create a new branch named `new_branch` on your local machine and go to it as follows:

```
$ git checkout -b new_branch
Switched to a new branch 'new_branch'
```

If you want to connect that branch with GitHub, you should execute the following command:

```
$ git push --set-upstream origin new_branch
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'new_branch' on GitHub by visiting:
remote:
https://github.com/PacktPublishing/Mastering-Go-Second-Edition/pull/new/new_branch
remote:
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 * [new branch]      new_branch -> new_branch
Branch 'new_branch' set up to track remote branch 'new_branch' from
'origin'.
```

If you want to change your current branch and go back to the `master` branch, you can execute the following command:

```
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
```

If you want to delete a local branch, `new_branch` in this case, you can execute the `git branch -D` command:

```
$ git --no-pager branch -a
* master
  new_branch
remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/new_branch
$ git branch -D new_branch
Deleted branch new_branch (was 98f8a77).
$ git --no-pager branch -a
* master
  remotes/origin/HEAD -> origin/master
remotes/origin/master
remotes/origin/new_branch
```

Working with files

When you add or remove one or more files from a repository, `git` (1) should know about it from you. You can delete a file named `a_file.go` as follows:

```
$ rm a_file.go
$ git rm a_file.go
rm 'ch07/a_file.go'
```

Executing `git status` at this point will generate the following kind of output:

```
$ git status
On branch master
Your branch is up to date with 'origin/master'.
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)
    deleted:    a_file.go
```

For changes to take effect you will need to `git commit` first and `git push` afterwards:

```
$ git commit -a -m "Deleting a_file.go"
[master 1b06700] Deleting a_file.go
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 ch07/a_file.go
$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 296 bytes | 296.00 KiB/s, done.
```

```
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 782c4da..1b06700 master -> master
```

The .gitignore file

The `.gitignore` file is used to list files or directories that you want to ignore when committing to GitHub. The contents of a sample `.gitignore` file might look as follows:

```
$ cat .gitignore
public/
.DS_Store
*.swp
```

Please note that after being created for the first time, `.gitignore` should be added to the current branch using `git add`.

Using git diff

The `git diff` command shows differences between commits and a working repository or branch, and so on.

The following command will show the changes between your files and the files that are on GitHub (before the last `git push`). These will be the changes that will be added to the version that is on GitHub after a `git push` command:

```
$ git diff
diff --git a/content/blog/Stats.md b/content/blog/Stats.md
index 0f36b60..af64ec3 100644
--- a/content/blog/Stats.md
+++ b/content/blog/Stats.md
@@ -16,6 +16,8 @@ title: Statistical analysis of random numbers
## Developing a Kafka producer in Go
+Please note that the format of the first record that is written to Kafka
+specifies the format of the subsequent records
### Viewing the data in Lenses
```

Working with tags

A **tag** is a way of identifying specific release versions of your code. You can think of a tag as a branch that never changes.

You can create a new **lightweight tag** as follows:

```
$ git tag c7.0
```

You can find information about a specific tag as follows:

```
$ git --no-pager show v1.0.0
commit f415872e62bd71a004b680d50fa089c139359533 (tag: v1.0.0)
Author: Mihalis Tsoukalos <mihalistsoukalos@gmail.com>
Date: Sat Mar 2 20:33:58 2019 +0200
    Initial version 1.0.0
diff --git a/go.mod b/go.mod
new file mode 100644
index 0000000..c4928c5
--- /dev/null
+++ b/go.mod
@@ -0,0 +1,3 @@
+module github.com/mactsouk/myModule
+
+go 1.12
diff --git a/myModule.go b/myModule.go
index e69de29..fa6b0fe 100644
--- a/myModule.go
+++ b/myModule.go
@@ -0,0 +1,9 @@
+package myModule
+
+import (
+    "fmt"
+)
+
+func Version() {
+    fmt.Println("Version 1.0.0")
+}
```

You can list all available tags using the `git tag` command:

```
$ git --no-pager tag
c7.0
```

You can push a tag to GitHub as follows:

```
$ git push origin c7.0
Total 0 (delta 0), reused 0 (delta 0)
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
 * [new tag] c7.0 -> c7.0
```

You can delete an existing tag from localhost as follows:

```
$ git tag -d c7.0
Deleted tag 'c7.0' (was 1b06700)
```

You can delete an existing tag from remote, which is the GitHub server, as follows:

```
$ git push origin :refs/tags/c7.0
To github.com:PacktPublishing/Mastering-Go-Second-Edition.git
- [deleted]          c7.0
```

The git cherry-pick command

The `git cherry-pick` command is an advanced command that should be used with care as it applies the changes introduced by some existing commits to the current branch. The following command will apply commit `4226f2c4` to the current branch:

```
$ git cherry-pick 4226f2c4
```

The following command will apply all commits from `4226f2c4` to `0d820a87` to the current branch without including the `4226f2c4` commit:

```
$ git cherry-pick 4226f2c4..0d820a87
```

The following command will apply all commits from `4226f2c4` to `0d820a87` to the current branch, including the `4226f2c4` commit:

```
$ git cherry-pick 4226f2c4^..0d820a87
```



Although the presented list of `git` (1) commands and options is far from complete, they will allow you to work with `git` (1) and GitHub, and will come in handy when creating Go modules.

Debugging with Delve

Delve is a text-based **debugger** for Go programs written in Go. On macOS Mojave, you can download Delve as follows:

```
$ go get -u github.com/go-delve/delve/cmd/dlv
$ ls -l ~/go/bin/dlv
-rwxr-xr-x  1 mtsouk  staff    16M Mar  7 09:04 /Users/mtsouk/go/bin/dlv
```


As Delve depends on a lot of Go modules and packages, the installation process is going to take a while. The Delve binary is installed on `~/go/bin`. Executing `dlv version` will reveal information about its version:

```
$ ~/go/bin/dlv version
Delve Debugger
Version: 1.2.0
Build: $Id: 068e2451004e95d0b042e5257e34f0f08ce01466 $
```

Note that Delve also works on Linux and Microsoft Windows machines. Also note that Delve is an external program, which means that you do not need to include any packages in your Go programs for Delve to work.

As this section is a quick introduction to Delve, the following subsection will present a small example to help you to get started with the Delve debugger. The presented information shows the general ideas behind Delve and almost every other debugger.

A debugging example

If `~/go/bin` is in your `PATH` environment variable, then you can call Delve from everywhere as `dlv`. Otherwise, you will need to provide its full path. I am going to use the full path in this subsection.

The first Delve command that you should know is `debug`. This command will tell Delve to compile the `main` package in the current working directory and begin to debug it. If there is no `main` package in the current working directory, you will get the following error message:

```
$ ~/go/bin/dlv debug
go: cannot find main module; see 'go help modules'
exit status 1
```

So, let us go to the `./ch07/debug` directory and debug a real program that is stored in `main.go`. The Go code of `main.go` is as follows:

```
package main

import (
    "fmt"
    "os"
)

func function(i int) int {
    return i * i
}
```

```
}

func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need at least one argument.")
        return
    }

    i := 5
    fmt.Println("Debugging with Delve")
    fmt.Println(i)
    fmt.Println(function(i))

    for arg, _ := range os.Args[1:] {
        fmt.Println(arg)
    }
}
```

In order to pass some command-line arguments to the program, you should execute Delve as follows:

```
$ ~/go/bin/dlv debug -- arg1 arg2 arg3
```

We are going to execute Delve as follows:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
```

The prompt of Delve is `(dlv)`, which is where you give your Delve commands. If you just press `c` (for `continue`) at this point, your Go program will be executed as if you were issuing the `go run` command from your shell:

```
(dlv) c
Debugging with Delve
5
25
0
1
2
Process 57252 has exited with status 0
(dlv)
```

If you press `c` or type `continue` again, you will get the following:

```
(dlv) c
Process 57252 has exited with status 0
```

This happens because your program has ended and cannot continue at this point – you will have to restart it in order to be able to debug it. You can restart a program by typing `r` or `restart`, which is equal to executing Delve from the UNIX shell:

```
(dlv) r
Process restarted with PID 57257
(dlv)
```

At this point, we will have to try something different. What we are going to do is set two breakpoints, one for the `main()` function and another one for `function()`:

```
$ ~/go/bin/dlv debug -- 1 2 3
Type 'help' for list of commands.
(dlv)
(dlv) break main.main
Breakpoint 1 set at 0x10b501b for main.main() ./main.go:12
(dlv) break function
Breakpoint 2 set at 0x10b4fe0 for main.function() ./main.go:8
(dlv)
```

If you press `continue` at this point, the debugger will stop the program at either `main()` or `function()`, whatever comes first. As this is an executable program, the `main()` function will be first:

```
(dlv) c
> main.main() ./main.go:12 (hits goroutine(1):1 total:1) (PC: 0x10b501b)
  7:
  8:     func function(i int) int {
  9:         return i * i
 10:     }
 11:
=> 12:     func main() {
 13:         if len(os.Args) == 1 {
 14:             fmt.Println("Need at least one argument.")
 15:             return
 16:         }
 17:
(dlv)
```

The `=>` arrow shows the line of the source code that the break happened in. Typing `next` will take us to the next Go statement – we can type `next` as many times as we want until we reach the end of the program:

```
(dlv) next
> main.main() ./main.go:13 (PC: 0x10b5032)
  8:     func function(i int) int {
  9:         return i * i
```

```

10:    }
11:
12:    func main() {
=> 13:        if len(os.Args) == 1 {
14:            fmt.Println("Need at least one argument.")
15:            return
16:        }
17:
18:        i := 5
(dlv) next
> main.main() ./main.go:18 (PC: 0x10b50d0)
13:        if len(os.Args) == 1 {
14:            fmt.Println("Need at least one argument.")
15:            return
16:        }
17:
=> 18:        i := 5
19:        fmt.Println("Debugging with Delve")
20:        fmt.Println(i)
21:        fmt.Println(function(i))
22:
23:        for arg, _ := range os.Args[1:] {
(dlv) next
> main.main() ./main.go:19 (PC: 0x10b50db)
14:            fmt.Println("Need at least one argument.")
15:            return
16:        }
17:
18:        i := 5
=> 19:        fmt.Println("Debugging with Delve")
20:        fmt.Println(i)
21:        fmt.Println(function(i))
22:
23:        for arg, _ := range os.Args[1:] {
24:            fmt.Println(arg)
(dlv) print i
5

```

The last command (`print i`) prints the value of variable `i`. Typing `continue` will take us to the next break point, if there is one, or to the end of the program:

```

(dlv) c
Debugging with Delve
5
> main.function() ./main.go:8 (hits goroutine(1):1 total:1) (PC: 0x10b4fe0)
3:    import (
4:        "fmt"
5:        "os"

```

```
6:    )
7:
=> 8:    func function(i int) int {
9:        return i * i
10:    }
11:
12:    func main() {
13:        if len(os.Args) == 1 {
```

In this case, the next break point is the `function()` function, as defined earlier.

Please note that for debugging Go tests, you should use the `dlv test` command and Delve will take care of the rest.

Additional resources

You will find the next resources very handy:

- Visit the documentation page of the `reflect` Go standard package, which can be found at <https://golang.org/pkg/reflect/>. This package has many more capabilities than the ones presented in this chapter.
- GitHub: <https://github.com/>.
- GitLab: <https://gitlab.com/>.
- You can find out more about Delve at <https://github.com/go-delve/delve>.
- You can find out more about the `reflectwalk` library by Mitchell Hashimoto at <https://github.com/mitchellh/reflectwalk>. Studying its code will help you to learn more about reflection.

Exercises

- Write your own interface and use it in another Go program. Then state why your interface is useful.
- Write an interface for calculating the volume of shapes with three dimensions, such as cubes and spheres.
- Write an interface for calculating the length of line segments and the distance between two points in the plane.
- Explore reflection using your own example.
- How does reflection work on Go maps?

- If you are good at mathematics, try to write an interface that implements the four basic mathematical operations for both real numbers and complex numbers. Do not use the `complex64` and `complex128` standard Go types – define your own structure for supporting complex numbers.

Summary

In this chapter, you learned about debugging, `git (1)`, GitHub, and interfaces, which are like contracts, and also about type methods, type assertion, and reflection in Go. Although reflection is a very powerful Go feature, it might slow down your Go programs because it adds a layer of complexity at runtime. Furthermore, your Go programs could crash if you use reflection carelessly.

You additionally learned about creating Go code that follows the principles of object-oriented programming. If you are going to remember just one thing from this chapter, it should be that Go is not an object-oriented programming language, but it can mimic some of the functionality offered by object-programming languages, such as Java and C++. This means that if you plan to develop software using the object-oriented paradigm all of the time, it would be best to choose a programming language other than Go. Nevertheless, object-oriented programming is not a panacea, and you might create a better, cleaner, and more robust design if you choose a programming language such as Go!

Although there may have been more theory in this chapter than you expected, the next chapter will reward your patience; it will address systems programming in Go. File I/O, working with UNIX system files, handling UNIX signals, and supporting UNIX pipes will be discussed.

The next chapter will also talk about using the `flag` and `viper` packages to support multiple command-line arguments and options in your command-line tools, as well as the `cobra` package, UNIX file permissions, and some advanced uses of the functionality offered by the `syscall` standard Go package. If you are really into systems programming with Go, my book on the subject elaborates on this (*Go Systems Programming* (Packt Publishing, 2017)).

8 Telling a UNIX System What to Do

In the previous chapter, we talked about two advanced, but somewhat theoretical, Go topics: interfaces and reflection. The Go code that you will find in this chapter is anything but theoretical!

The subject of this chapter is **systems programming** because, after all, Go is a mature systems programming language that was born out of frustration. Its spiritual fathers were unsatisfied with the programming language choices they had for creating systems software, so they decided to create a new programming language.



This chapter contains some interesting and somewhat advanced topics that are not included in *Go Systems Programming* (Packt Publishing, 2017).

This chapter focuses on the following topics:

- UNIX processes
- The `flag` package
- The `viper` package
- The `cobra` package
- The use of the `io.Reader` and `io.Writer` interfaces
- Handling UNIX **signals** in Go with the help of the `os/signal` package
- Supporting UNIX **pipes** in your UNIX system utilities
- Creating Go clients for Docker
- Reading text files
- Reading CSV files
- Writing to files

- The `bytes` package
- Advanced uses of the `syscall` package
- UNIX file permissions

About UNIX processes

Strictly speaking, a **process** is an execution environment that contains instructions, user data and system data parts, and other types of resources that are obtained during runtime. On the other hand, a **program** is a binary file that contains instructions and data that are used for initializing the instruction and user data parts of a process. Each running UNIX process is uniquely identified by an unsigned integer, which is called the **process ID** of the process.

There are three categories of processes: **user processes**, **daemon processes**, and **kernel processes**. User processes run in user space and usually have no special access rights. Daemon processes are programs that can be found in the user space and run in the background without the need for a terminal. Kernel processes are executed in kernel space only and can fully access all kernel data structures.



The C way of creating new processes involves calling the `fork()` system call. The return value of `fork()` allows the programmer to differentiate between the parent and the child process. In contrast, Go does not support a similar functionality but offers goroutines.

The flag package

Flags are specially-formatted strings that are passed into a program to control its behavior. Dealing with flags on your own might become very difficult if you want to support multiple flags. Thus, if you are developing UNIX system command-line utilities, you will find the `flag` package very interesting and useful.

The `flag` package makes no assumptions about the order of command-line arguments and options, and it prints helpful messages in case there was an error in the way the command-line utility was executed.



The biggest advantage of the `flag` package is that it is part of the standard Go library, which means that it has been extensively tested and debugged.

I will present two Go programs that use the `flag` package: a simple one and a more advanced one. The first one, named `simpleFlag.go`, will be offered in four parts. The `simpleFlag.go` program will recognize two command-line options: the first one will be a Boolean option and the second one will require an integer value.

The first part of `simpleFlag.go` contains the following Go code:

```
package main

import (
    "flag"
    "fmt"
)
```

The second code portion from `simpleFlag.go` is as follows:

```
func main() {
    minusK := flag.Bool("k", true, "k flag")
    minusO := flag.Int("O", 1, "O")
    flag.Parse()
}
```

The `flag.Bool("k", true, "k flag")` statement defines a Boolean command-line option named `k` with the default value of `true`. The last parameter of the statement is the usage string that will be displayed with the usage information of the program. Similarly, the `flag.Int()` function adds support for an integer command-line option.



You always need to call `flag.Parse()` after defining the command-line options that you want to support.

The third part of the `simpleFlag.go` program contains the following Go code:

```
valueK := *minusK
valueO := *minusO
valueO++
```

In the preceding Go code, you can see how you can obtain the values of your options. The good thing here is that the `flag` package automatically converts the input associated with the `flag.Int()` flag to an integer value. This means that you do not have to do that on your own. Additionally, the `flag` package makes sure that it was given an acceptable integer value.

The remaining Go code from `simpleFlag.go` follows:

```
fmt.Println("-k:", valueK)
```

```
    fmt.Println("-O:", valueO)
}
```

After getting the values of the desired parameters, you are now ready to use them.

Interacting with `simpleFlag.go` will create the following type of output:

```
$ go run simpleFlag.go -O 100
-k: true
-O: 101
$ go run simpleFlag.go -O=100
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k false
-k: true
-O: 101
$ go run simpleFlag.go -O=100 -k=false
-k: false
-O: 101
```

If there is an error in the way `simpleFlag.go` was executed, you will get the following type of error message from the `flag` package:

```
$ go run simpleFlag.go -O=notAnInteger
invalid value "notAnInteger" for flag -O: parse error
Usage of /var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-
build593534621/b001/exe/simpleFlag:
  -O int
      O (default 1)
  -k    flag (default true)
exit status 2
```

Notice the convenient usage message that is automatically printed when there is an error in the command-line options given to your program.

Now it is time to present a more realistic and advanced program that uses the `flag` package. Its name is `funWithFlag.go`, and it will be presented in five parts. The `funWithFlag.go` utility will recognize various kinds of options, including one that accepts multiple values that are separated by commas. Additionally, it will illustrate how you can access the command-line arguments that are located at the end of the executable and do not belong to any option.

The `flag.Var()` function used in `funWithFlag.go` creates a flag of any type that satisfies the `flag.Value` interface, which is defined as follows:

```
type Value interface {
    String() string
    Set(string) error
}
```

The first part of `funWithFlag.go` contains the following Go code:

```
package main

import (
    "flag"
    "fmt"
    "strings"
)

type NamesFlag struct {
    Names []string
}
```

The `NamesFlag` structure will be used in a short while for the `flag.Value` interface.

The second part of `funWithFlag.go` is as follows:

```
func (s *NamesFlag) GetNames() []string {
    return s.Names
}

func (s *NamesFlag) String() string {
    return fmt.Sprintf("%v", s.Names)
}
```

The third code portion from `funWithFlag.go` contains the following code:

```
func (s *NamesFlag) Set(v string) error {
    if len(s.Names) > 0 {
        return fmt.Errorf("Cannot use names flag more than once!")
    }

    names := strings.Split(v, ",")
    for _, item := range names {
        s.Names = append(s.Names, item)
    }
    return nil
}
```

First, the `Set()` method makes sure that the related command-line option is not already set. After that, it gets the input and separates its arguments using the `strings.Split()` function. Then, the arguments are saved in the `Names` field of the `NamesFlag` structure.

The fourth code segment from `funWithFlag.go` is shown in the following Go code:

```
func main() {
    var manyNames NamesFlag
    minusK := flag.Int("k", 0, "An int")
    minusO := flag.String("o", "Mihalis", "The name")
    flag.Var(&manyNames, "names", "Comma-separated list")

    flag.Parse()
    fmt.Println("-k:", *minusK)
    fmt.Println("-o:", *minusO)
```

The last part of the `funWithFlag.go` utility follows:

```
    for i, item := range manyNames.GetNames() {
        fmt.Println(i, item)
    }

    fmt.Println("Remaining command line arguments:")
    for index, val := range flag.Args() {
        fmt.Println(index, ":", val)
    }
}
```

The `flag.Args()` slice keeps the command-line arguments that are left, while the `manyNames` variable holds the values from the `flag.Var()` command-line option.

Executing `funWithFlag.go` will create the following type of output:

```
$ go run funWithFlag.go -names=Mihalis,Jim,Athina 1 two Three
-k: 0
-o: Mihalis
0 Mihalis
1 Jim
2 Athina
Remaining command line arguments:
0 : 1
1 : two
2 : Three
$ go run funWithFlag.go -Invalid=Marietta 1 two Three
flag provided but not defined: -Invalid
Usage of funWithFlag:
-k int
```

```
    An int
    -names value
        Comma-separated list
    -o string
        The name (default "Mihalis")
exit status 2
$ go run funWithFlag.go -names=Marietta -names=Mihalis
invalid value "Mihalis" for flag -names: Cannot use names flag more than
once!
Usage of funWithFlag:
    -k int
        An int
    -names value
        Comma-separated list
    -o string
        The name (default "Mihalis")
exit status 2
```



Unless you are developing a trivial command-line utility that requires no command-line options, you will most likely need to use a Go package to process the command-line arguments of your program.

The viper package

`viper` is a powerful Go package that supports a plethora of options. All `viper` projects follow a pattern. First, you initialize `viper` and then you define the elements that interest you. After that, you get these elements and read their values in order to use them. Notice that the `viper` package can entirely replace the `flag` package.

The desired values can be either taken directly, as happens when you are using the `flag` package of the standard Go library, or indirectly using configuration files. When using formatted configuration files in the JSON, YAML, TOML, HCL, or Java properties format, `viper` does all the parsing for you, which saves you from having to write and debug lots of Go code. `viper` also allows you to extract and save values in Go structures. However, this also requires the fields of the Go structure to match the keys of the configuration file.

The home page of `viper` is on GitHub (<https://github.com/spf13/viper>). Please note that you are not obliged to use every capability of `viper` in your tools – just the features that you want. The general rule is to use the features of `viper` that simplify your code. Put simply, if your command-line utility requires too many command-line parameters and flags, then it would be better to use a configuration file instead.

A simple viper example

Before going into more advanced examples, I will present some sample Go code that uses `viper`. The name of the program is `usingViper.go` and it will be presented in three parts.

The first part of `useViper.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

The second part of `useViper.go` is as follows:

```
func main() {
    viper.BindEnv("GOMAXPROCS")
    val := viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
    viper.Set("GOMAXPROCS", 10)
    val = viper.Get("GOMAXPROCS")
    fmt.Println("GOMAXPROCS:", val)
}
```

The last part of `useViper.go` contains the following Go code:

```
viper.BindEnv("NEW_VARIABLE")
val = viper.Get("NEW_VARIABLE")
if val == nil {
    fmt.Println("NEW_VARIABLE not defined.")
    return
}
fmt.Println(val)
}
```

The purpose of this program is to illustrate how you can read and modify environment variables using `viper`. The `flag` package does not offer such functionality but the `os` standard Go package does, although not as easily as the `viper` package.

The first time you are going to use `viper`, you will need to download it. If you are not using Go modules, this can be done as follows:

```
$ go get -u github.com/spf13/viper
```

If you are using Go modules, Go will automatically download `viper` the first time you try to execute a Go program that uses it.

Executing `useViper.go` will generate the following kind of output:

```
$ go run useViper.go
GOMAXPROCS: <nil>
GOMAXPROCS: 10
NEW_VARIABLE not defined.
```

From flag to viper

There is a chance that you have a Go program that already uses the `flag` package and that you want to convert to using the `viper` package. Let us say that we have the following Go code that uses the `flag` package:

```
package main

import (
    "flag"
    "fmt"
)

func main() {
    minusI := flag.Int("i", 100, "i parameter")
    flag.Parse()
    i := *minusI
    fmt.Println(i)
}
```

The new version of that program, which is going to be saved in `flagToViper.go`, is going to use the `viper` package and will be presented in three parts. The first part of `flagToViper.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "github.com/spf13/pflag"
    "github.com/spf13/viper"
)
```

You will need to import the `pflag` package in order to work with command-line arguments in `viper`.

The second part of `flagToViper.go` contains the following Go code:

```
func main() {
    flag.Int("i", 100, "i parameter")
    pflag.CommandLine.AddGoFlagSet(flag.CommandLine)
    pflag.Parse()
```

So, you still use `flag.Int()` in order to change as little code as possible, but for the parsing, you call `pflag.Parse()`. However, all the magic happens with the call to `pflag.CommandLine.AddGoFlagSet(flag.CommandLine)` because this call imports the data from the `flag` package to the `pflag` package.

The last part of `flagToViper.go` is the following:

```
    viper.BindPFlags(pflag.CommandLine)
    i := viper.GetInt("i")
    fmt.Println(i)
}
```

The last function call that you have to make is `viper.BindPFlags()`. After that, you can get the value of an integer command-line parameter using `viper.GetInt()`. For other kinds of data types, you will have to call different `viper` functions.

At this point, you might need to download the `pflag` Go package for `flagToViper.go` to work, which can be done as follows:

```
$ go get -u github.com/spf13/pflag
```

Executing `flagToViper.go` will generate the following kind of output:

```
$ go run flagToViper.go
100
$ go build flagToViper.go
$ ./flagToViper -i 0
0
$ ./flagToViper -i abcd
invalid argument "abcd" for "-i, --i" flag: parse error
Usage of ./flagToViper:
  -i, --i int    i parameter
invalid argument "abcd" for "-i, --i" flag: parse error
```

If, for some reason, you give an unknown command-line parameter to `flagToViper.go`, `viper` will complain about it:


```
$ ./flagToViper -j 200
unknown shorthand flag: 'j' in -j
Usage of ./flagToViper:
  -i, --i int    i parameter (default 100)
unknown shorthand flag: 'j' in -j
exit status 2
```

Reading JSON configuration files

In this subsection, you will learn how to read JSON configuration files with the `viper` package. The name of the utility is `readJSON.go` and it is going to be presented in three parts. The first part of `readJSON.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/spf13/viper"
)
```

The second part of `readJSON.go` contains the following code:

```
func main() {
    viper.SetConfigType("json")
    viper.SetConfigFile("./myJSONConfig.json")
    fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
    viper.ReadInConfig()
```

This is where the parsing of the configuration file takes place. Please note that the filename of the JSON configuration file is hardcoded inside `readJSON.go` using the `viper.SetConfigFile("./myJSONConfig.json")` function call.

The last part of `readJSON.go` is as follows:

```
if viper.IsSet("item1.key1") {
    fmt.Println("item1.key1:", viper.Get("item1.key1"))
} else {
    fmt.Println("item1.key1 not set!")
}

if viper.IsSet("item2.key3") {
    fmt.Println("item2.key3:", viper.Get("item2.key3"))
} else {
    fmt.Println("item2.key3 is not set!")
}
```

```
    if !viper.IsSet("item3.key1") {  
        fmt.Println("item3.key1 is not set!")  
    }  
}
```

This is where the values of the JSON configuration file are examined by the program in order to find out whether the desired keys exist or not.

The contents of the `myJSONConfig.json` file are as follows:

```
{  
  "item1": {  
    "key1": "val1",  
    "key2": false,  
    "key3": "val3"  
  },  
  "item2": {  
    "key1": "val1",  
    "key2": true,  
    "key3": "val3"  
  }  
}
```

Executing `readJSON.go` will generate the following kind of output:

```
$ go run readJSON.go  
Using config: ./myJSONConfig.json  
item1.key1: val1  
item2.key3: val3  
item3.key1 is not set!
```

If `myJSONConfig.json` cannot be located, the program will not complain and will act as if it has read an empty JSON configuration file:

```
$ mv myJSONConfig.json ..  
$ go run readJSON.go  
Using config: ./myJSONConfig.json  
item1.key1 not set!  
item2.key3 is not set!  
item3.key1 is not set!
```

Reading YAML configuration files

YAML is another popular text-based format that is used for configuration files. In this subsection, you will learn how to read YAML configuration files with the `viper` package.

However, this time, the filename of the YAML configuration file will be given as a command-line argument to the utility. Additionally, the utility will use the `viper.AddConfigPath()` function to add three search paths, which are places where `viper` will automatically look for configuration files. The name of the utility, which will be presented in four parts, is `readYAML.go`.

The first part of `readYAML.go` is as follows:

```
package main

import (
    "fmt"
    flag "github.com/spf13/pflag"
    "github.com/spf13/viper"
    "os"
)
```

In the preamble of the program, we define an alias (`flag`) for the `pflag` Go package.

The second part of `readYAML.go` contains the following Go code:

```
func main() {
    var configFile *string = flag.String("c", "myConfig", "Setting the
configuration file")
    flag.Parse()

    _, err := os.Stat(*configFile)

    if err == nil {
        fmt.Println("Using User Specified Configuration file!")
        viper.SetConfigFile(*configFile)
    } else {
        viper.SetConfigName(*configFile)
        viper.AddConfigPath("/tmp")
        viper.AddConfigPath("$HOME")
        viper.AddConfigPath(".")
    }
}
```

The code checks whether the value of the config flag (`--c`) exists using a call to `os.Stat()`. If it exists, the provided file will be used; otherwise, the default config filename (`myConfig`) will be used. Notice that we are not explicitly specifying that we want to use a YAML configuration file - the program will look for all supported file formats, provided that the filename without the file extension is `myConfig`, because this is the way `viper` works. Three paths will be searched for configuration files: `/tmp`, the home directory of the current user, and the current working directory, in that order.

Using the `/tmp` directory to keep your configuration files is not recommended, mainly because the contents of `/tmp` are automatically deleted after each system reboot - it is only used here for reasons of simplicity.

The use of `viper.ConfigFileUsed()` makes perfect sense because there is no hardcoded configuration file, which means that we will have to define it on our own.

The third part of `readYAML.go` is as follows:

```
err = viper.ReadInConfig()
if err != nil {
    fmt.Printf("%v\n", err)
    return
}
fmt.Printf("Using config: %s\n", viper.ConfigFileUsed())
```

The YAML file is read and parsed using a call to `viper.ReadInConfig()`.

The last part of `readYAML.go` is as follows:

```
if viper.IsSet("item1.k1") {
    fmt.Println("item1.val1:", viper.Get("item1.k1"))
} else {
    fmt.Println("item1.k1 not set!")
}

if viper.IsSet("item1.k2") {
    fmt.Println("item1.val2:", viper.Get("item1.k2"))
} else {
    fmt.Println("item1.k2 not set!")
}

if !viper.IsSet("item3.k1") {
    fmt.Println("item3.k1 is not set!")
}
}
```

This is where the contents of the parsed configuration file are examined by the program in order to find out whether the desired keys exist or not.

The contents of `myConfig.yaml` are as follows:

```
item1:
  k1:
    - true
  k2:
    - myValue
```

Executing `readYAML.go` will generate the following kind of output:

```
$ go build readYAML.go
$ ./readYAML
Using config: /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch08/viper/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
$ mv myConfig.yaml /tmp
$ ./readYAML
Using config: /tmp/myConfig.yaml
item1.val1: [true]
item1.val2: [myValue]
item3.k1 is not set!
```

The cobra package

`cobra` is a very handy and popular Go package that allows you to develop command-line utilities with commands, subcommands, and aliases. If you have ever used `hugo`, `docker`, or `kubectl` you will understand immediately what `Cobra` does, as all these tools were developed using `cobra`.

As you will see in this section, commands in `cobra` can have one or more aliases, which is very handy when you want to please both amateur and experienced users. `cobra` also supports Persistent Flags and Local Flags, which are flags that are available to all commands and flags that are available to a given command only, respectively. Also, by default, `cobra` uses `viper` for parsing its command-line arguments.

All `cobra` projects follow the same development pattern. You use the `cobra` tool, then you create commands, and then you make the desired changes to the generated Go source code files in order to implement the desired functionality. Depending on the complexity of your utility, you might need to make lots of changes to the created files. Although `cobra` saves you lots of time, you will still have to write the code that implements the desired functionality.

If you want to install your utility in binary version, you can always execute `go install` from anywhere in the `cobra` project directory. Unless otherwise specified, the binary executable will be placed in `~/go/bin`.

The home page of `cobra` is on GitHub (<https://github.com/spf13/cobra>).

A simple cobra example

In this section, we are going to implement a simple command-line utility using cobra and the `~/go/bin/cobra` tool that comes with the package. If you execute `~/go/bin/cobra` without any command-line arguments, you will get the following output:

```
$ ~/go/bin/cobra
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.
Usage:
  cobra [command]
Available Commands:
  add      Add a command to a Cobra Application
  help     Help about any command
  init     Initialize a Cobra Application
Flags:
  -a, --author string    author name for copyright attribution (default
"YOUR NAME")
  --config string        config file (default is $HOME/.cobra.yaml)
  -h, --help             help for cobra
  -l, --license string   name of license for the project
  --viper                use Viper for configuration (default true)
Use "cobra [command] --help" for more information about a command.
```

With that information in mind, we are going to create a new cobra project, as follows:

```
$ ~/go/bin/cobra init cli
Your Cobra application is ready at
/Users/mtsouk/go/src/cli
Give it a try by going there and running `go run main.go`.
Add commands to it by running `cobra add [cmdname]`.
$ cd ~/go/src/cli
$ ls -l
total 32
-rw-r--r--  1 mtsouk  staff  11358 Mar 13 09:51 LICENSE
drwxr-xr-x  3 mtsouk  staff    96 Mar 13 09:51 cmd
-rw-r--r--  1 mtsouk  staff    669 Mar 13 09:51 main.go
$ ~/go/bin/cobra add cmdOne
cmdOne created at /Users/mtsouk/go/src/cli/cmd/cmdOne.go
$ ~/go/bin/cobra add cmdTwo
cmdTwo created at /Users/mtsouk/go/src/cli/cmd/cmdTwo.go
```

The `cobra init` command creates a new cobra project inside `~/go/src`, named after its last parameter (`cli`). The `cobra add` command adds a new command to the command-line tool and creates all necessary files and Go functions to support that command.

Therefore, after each execution of the `cobra add` command, Cobra does most of the dirty work for us. However, you will still need to implement the functionality of the commands you just added - in this case, the commands are called `cmdOne` and `cmdTwo`. The `cmdOne` command will accept a local command-line flag named `number` - you will also need to write some extra code for this feature to work.

At this point, if you execute `go run main.go`, you will get the following output:

```
A longer description that spans multiple lines and likely contains
examples and usage of using your application. For example:
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.
Usage:
  cli [command]
Available Commands:
  cmdOne      A brief description of your command
  cmdTwo      A brief description of your command
  help        Help about any command
Flags:
  --config string  config file (default is $HOME/.cli.yaml)
  -h, --help       help for cli
  -t, --toggle     Help message for toggle
Use "cli [command] --help" for more information about a command.
```

The Go code for the `cmdOne` command can be found in `./cmd/cmdOne.go` and for the `cmdTwo` command in `./cmd/cmdTwo.go`.

The final version of `./cmd/cmdOne.go` is as follows:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

// cmdOneCmd represents the cmdOne command
var cmdOneCmd = &cobra.Command{
    Use:     "cmdOne",
    Short:   "A brief description of your command",
    Long:    `A longer description that spans multiple lines and likely
contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
```

```

to quickly create a Cobra application.`
Run: func(cmd *cobra.Command, args []string) {
    fmt.Println("cmdOne called!")
    number, _ := cmd.Flags().GetInt("number")
    fmt.Println("Going to use number", number)
    fmt.Printf("Square: %d\n", number*number)
},
}

func init() {
    rootCmd.AddCommand(cmdOneCmd)
    cmdOneCmd.Flags().Int("number", 0, "A help for number")
}

```

The previous Go code comes without the comments that are automatically generated by cobra. The following line of code in the `init()` function is what defines the new local command-line flag:

```
cmdOneCmd.Flags().Int("number", 0, "A help for number")
```

That flag, which is called `number`, is used in the `cobra.Command` block as follows:

```
number, _ := cmd.Flags().GetInt("number")
```

After that, you can use the `number` variable any way you want.

The Go code of the final version of `./cmd/cmdTwo.go`, which includes comments and license information, is as follows:

```

// Copyright © 2019 NAME HERE <EMAIL ADDRESS>
//
// Licensed under the Apache License, Version 2.0 (the "License");
// you may not use this file except in compliance with the License.
// You may obtain a copy of the License at
//
//      http://www.apache.org/licenses/LICENSE-2.0
//
// Unless required by applicable law or agreed to in writing, software
// distributed under the License is distributed on an "AS IS" BASIS,
// WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
// See the License for the specific language governing permissions and
// limitations under the License.

package cmd

import (
    "fmt"
    "github.com/spf13/cobra"

```



```

)

// cmdTwoCmd represents the cmdTwo command
var cmdTwoCmd = &cobra.Command{
    Use:     "cmdTwo",
    Short:   "A brief description of your command",
    Long:    `A longer description that spans multiple lines and likely
contains examples
and usage of using your command. For example:

Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.` ,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("cmdTwo called!")
    },
}

func init() {
    rootCmd.AddCommand(cmdTwoCmd)

    // Here you will define your flags and configuration settings.

    // Cobra supports Persistent Flags which will work for this command
    // and all subcommands, e.g.:
    // cmdTwoCmd.PersistentFlags().String("foo", "", "A help for foo")

    // Cobra supports local flags which will only run when this command
    // is called directly, e.g.:
    // cmdTwoCmd.Flags().BoolP("toggle", "t", false, "Help message for
toggle")
}

```

This is the default implementation of the `cmdTwo` command generated by `cobra`.

Executing the `cli` tool will generate the following kind of output:

```

$ go run main.go cmdOne
cmdOne called!
Going to use number 0
Square: 0
$ go run main.go cmdOne --number -20
cmdOne called!
Going to use number -20
Square: 400
$ go run main.go cmdTwo
cmdTwo called!

```

If you give the `cli` the tool wrong input, it will generate the following kind of error messages:

```
$ go run main.go cmdThree
Error: unknown command "cmdThree" for "cli"
Run 'cli --help' for usage.
unknown command "cmdThree" for "cli"
exit status 1
$ go run main.go cmdOne --n -20
Error: unknown flag: --n
Usage:
  cli cmdOne [flags]
Flags:
  -h, --help           help for cmdOne
  --number int         A help for number
Global Flags:
  --config string      config file (default is $HOME/.cli.yaml)
unknown flag: --n
exit status 1
```

The help screen for the `cmdOne` command, which is automatically generated, is as follows:

```
$ go run main.go cmdOne --help
A longer description that spans multiple lines and likely contains examples
and usage of using your command. For example:
Cobra is a CLI library for Go that empowers applications.
This application is a tool to generate the needed files
to quickly create a Cobra application.
Usage:
  cli cmdOne [flags]
Flags:
  -h, --help           help for cmdOne
  --number int         A help for number
Global Flags:
  --config string      config file (default is $HOME/.cli.yaml)
```

The directory structure and the files of the final version of the utility can be displayed with the help of the `tree(1)` command:

```
$ tree
.
├── LICENSE
├── cmd
│   ├── cmdOne.go
│   ├── cmdTwo.go
│   └── root.go
└── main.go
1 directory, 5 files
```

Creating command aliases

In this section, you will learn how to create aliases for existing commands with `cobra`.

As before, you will first need to create a new `cobra` project, which in this case is going to be called `aliases`, along with the desired commands. This can be done as follows:

```
$ ~/go/bin/cobra init aliases
$ cd ~/go/src/aliases
$ ~/go/bin/cobra add initialization
initialization created at
/Users/mtsouk/go/src/aliases/cmd/initialization.go
$ ~/go/bin/cobra add preferences
preferences created at /Users/mtsouk/go/src/aliases/cmd/preferences.go
```

So far, you have a command-line utility that supports two commands, named `initialization` and `preferences`.

Each alias of an existing `cobra` command needs to be explicitly specified in the Go code. For the `initialization` command, you will need the following line of code in the appropriate place in `./cmd/initialization.go`:

```
Aliases: []string{"initialize", "init"},
```

The previous statement creates two aliases for the `initialization` command, named `initialize` and `init`. The contents of the final version of `./cmd/initialization.go`, without any comments, will be as follows:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var initializationCmd = &cobra.Command{
    Use:     "initialization",
    Aliases: []string{"initialize", "init"},
    Short:   "A brief description of your command",
    Long:    `A longer description of your command`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("initialization called")
    },
}

func init() {
```

```
    rootCmd.AddCommand(initializationCmd)
}
```

Similarly, for the preferences command, you will need to include the following line of Go code in the appropriate place in `./cmd/preferences.go`:

```
Aliases: []string{"prefer", "pref", "prf"},
```

The aforementioned statement creates three aliases for the preferences command, named `prefer`, `pref`, and `prf`.

The contents of the final version of `./cmd/preferences.go`, without any comments, are as follows:

```
package cmd

import (
    "fmt"
    "github.com/spf13/cobra"
)

var preferencesCmd = &cobra.Command{
    Use:     "preferences",
    Aliases: []string{"prefer", "pref", "prf"},
    Short:   "A brief description of your command",
    Long:    `A longer description of your command`,
    Run: func(cmd *cobra.Command, args []string) {
        fmt.Println("preferences called")
    },
}

func init() {
    rootCmd.AddCommand(preferencesCmd)
}
```

Executing the `aliases` command-line utility will generate the following kind of output:

```
$ go run main.go prefer
preferences called
$ go run main.go prf
preferences called
$ go run main.go init
initialization called
```

If you give erroneous commands to `aliases`, then it will generate the following kind of error messages:

```
$ go run main.go inits
Error: unknown command "inits" for "aliases"
Run 'aliases --help' for usage.
unknown command "inits" for "aliases"
exit status 1
$ go run main.go preferR
Error: unknown command "preferR" for "aliases"
Did you mean this?
    preferences
Run 'aliases --help' for usage.
unknown command "preferR" for "aliases"
Did you mean this?
    preferences
exit status 1
```

The `tree(1)` command, which does not come with most UNIX systems and must be installed separately, can help us to get an idea of the directory structure and the files of the generated `cobra` utility:

```
$ tree
.
├── LICENSE
├── cmd
│   ├── initialization.go
│   ├── preferences.go
│   └── root.go
└── main.go
1 directory, 5 files
```



Both `viper` and `cobra` have more features and capabilities than the ones presented here.

The `io.Reader` and `io.Writer` Interfaces

As stated in the previous chapter, compliance with the `io.Reader` interface requires the implementation of the `Read()` method, whereas if you want to satisfy the `io.Writer` interface guidelines, you will need to implement the `Write()` method. Both these interfaces are very popular in Go and we will put them to use in a little while.

Buffered and unbuffered file input and output

Buffered file input and output happens when there is a buffer for temporarily storing data before reading data or writing data. Thus, instead of reading a file byte by byte, you read many bytes at once. You put the data in a buffer and wait for someone to read it in the desired way. Unbuffered file input and output happens when there is no buffer to temporarily store data before actually reading or writing it.

The next question that you might ask is how to decide when to use buffered and when to use unbuffered file input and output. When dealing with critical data, unbuffered file input and output is generally a better choice because buffered reads might result in out-of-date data and buffered writes might result in data loss when the power of your computer is interrupted. However, most of the time, there is no definitive answer to that question. This means that you can use whatever makes your tasks easier to implement.

The bufio package

As the name suggests, the `bufio` package is about buffered input and output. However, the `bufio` package still uses (internally) the `io.Reader` and `io.Writer` objects, which it wraps in order to create the `bufio.Reader` and `bufio.Writer` objects, respectively. As you will see in the forthcoming sections, the `bufio` package is very popular for reading text files.

Reading text files

A text file is the most common kind of file that you can find on a UNIX system. In this section, you will learn how to read text files in three ways: line by line, word by word, and character by character. As you will see, reading a text file line by line is the easiest method to access a text file, while reading a text file word by word is the most difficult method of all.

If you look closely at the `byLine.go`, `byWord.go`, and `byCharacter.go` programs, you will see many similarities in their Go code. Firstly, all three utilities read the input file line by line. Secondly, all three utilities have the same `main()` function, with the exception of the function that is called in the `for` loop of the `main()` function. Lastly, the three functions that process the input text files are almost identical, except for the part that implements the actual functionality of the function.

Reading a text file line by line

Going line by line is the most common method of reading a text file. This is the main reason that it is being shown first. The Go code of `byLine.go`, which will be presented in three parts, will help you to understand the technique.

The first code segment from `byLine.go` is shown in the following Go code:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

As you can see by the presence of the `bufio` package, we will use buffered input.

The second part of `byLine.go` contains the following Go code:

```
func lineByLine(file string) error {
    var err error

    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            break
        }
        fmt.Print(line)
    }
    return nil
}
```

All the magic happens in the `lineByLine()` function. After making sure that you can open the given filename for reading, you create a new reader using `bufio.NewReader()`. Then, you use that reader with `bufio.ReadString()` in order to read the input file line by line. The trick is done by the parameter of `bufio.ReadString()`, which is a character that tells `bufio.ReadString()` to keep reading until that character is found. Constantly calling `bufio.ReadString()` when that parameter is the newline character results in reading the input file line by line. The use of `fmt.Print()` instead of `fmt.Println()` for printing the input line shows that the newline character is included in each input line.

The third part of `byLine.go` follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byLine <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := lineByLine(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Executing `byLine.go` and processing its output with `wc (1)` will generate the following type of output:

```
$ go run byLine.go /tmp/swtag.log /tmp/adobegc.log | wc
4761    88521  568402
```

The following command will verify the accuracy of the preceding output:

```
$ wc /tmp/swtag.log /tmp/adobegc.log
  131      693    8440 /tmp/swtag.log
4630   87828 559962 /tmp/adobegc.log
4761   88521 568402 total
```

Reading a text file word by word

The technique presented in this subsection will be demonstrated by the `byWord.go` file and shown in four parts. As you will see in the Go code, separating the words of a line can be tricky.

The first part of this utility is as follows:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
    "regexp"
)
```

The second code portion of `byWord.go` is shown in the following Go code:

```
func wordByWord(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
    }
}
```

This part of the `wordByWord()` function is the same as the `lineByLine()` function of the `byLine.go` utility.

The third part of `byWord.go` is as follows:

```
    r := regexp.MustCompile("^[^\s]+")
    words := r.FindAllString(line, -1)
    for i := 0; i < len(words); i++ {
        fmt.Println(words[i])
    }
}
return nil
}
```

The remaining code of the `wordByWord()` function is totally new and it uses regular expressions to separate the words found in each line of the input file. The regular expression defined in the `regexp.MustCompile("[^\\s]+")` statement states that empty characters will separate one word from another.

The last code segment of `byWord.go` is as follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byWord <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := wordByWord(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

Executing `byWord.go` will produce the following type of output:

```
$ go run byWord.go /tmp/adobegc.log
01/08/18
20:25:09:669
|
[INFO]
```

You can verify the validity of `byWord.go` with the help of the `wc(1)` utility:

```
$ go run byWord.go /tmp/adobegc.log | wc
91591  91591  559005
$ wc /tmp/adobegc.log
4831   91591  583454 /tmp/adobegc.log
```

As you can see, the number of words calculated by `wc(1)` is the same as the number of lines and words that you took from the execution of `byWord.go`.

Reading a text file character by character

In this section, you will learn how to read a text file character by character, which is a pretty rare requirement unless you want to develop a text editor. The relevant Go code will be saved as `byCharacter.go`, which will be presented in four parts.

The first part of `byCharacter.go` is shown in the following Go code:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "os"
)
```

As you can see, you will not need to use regular expressions for this task.

The second code segment from `byCharacter.go` is as follows:

```
func charByChar(file string) error {
    var err error
    f, err := os.Open(file)
    if err != nil {
        return err
    }
    defer f.Close()

    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            return err
        }
    }
}
```

The third part of `byCharacter.go` is where the logic of the program is found:

```
    for _, x := range line {
        fmt.Println(string(x))
    }
    return nil
}
```

Here, you take each line that you read and split it using `range`, which returns two values. You discard the first, which is the location of the current character in the `line` variable, and you use the second. However, that value is not a character - that is the reason you have to convert it into a character using the `string()` function.

Note that, due to the `fmt.Println(string(x))` statement, each character is printed in a distinct line, which means that the output of the program will be large. If you want a more compressed output, you should use the `fmt.Print()` function instead.

The last part of `byCharacter.go` contains the following Go code:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: byChar <file1> [<file2> ...]\n")
        return
    }

    for _, file := range flag.Args() {
        err := charByChar(file)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

The execution of `byCharacter.go` will generate the following type of output:

```
$ go run byCharacter.go /tmp/adobegc.log
0
1
/
0
8
/
1
8
```

Note that the Go code presented here can be used for counting the number of characters found in the input file, which can help you to implement a Go version of the handy `wc(1)` command-line utility.

Reading from `/dev/random`

In this section, you will learn how to read from the `/dev/random` system device. The purpose of the `/dev/random` system device is to generate random data, which you might use for testing your programs or, in this case, as the seed for a random number generator. Getting data from `/dev/random` can be a little bit tricky, and this is the main reason for specifically discussing it here.

On a macOS Mojave machine, the `/dev/random` file has the following permissions:

```
$ ls -l /dev/random
crw-rw-rw-  1 root  wheel   14,   0 Mar  12 20:24 /dev/random
```

Similarly, on a Debian Linux machine, the `/dev/random` system device has the following UNIX file permissions:

```
$ ls -l /dev/random
crw-rw-rw-  1 root  root   1,  8 Oct 13 12:19 /dev/random
```

This means that the `/dev/random` file has analogous file permissions on both UNIX variants. The only difference between these two UNIX variants is the UNIX group that owns the file, which is `wheel` on macOS and `root` on Debian Linux.

The name of the program for this topic is `devRandom.go`, and it will be presented in three parts. The first part of the program is as follows:

```
package main

import (
    "encoding/binary"
    "fmt"
    "os"
)
```

In order to read from `/dev/random`, you will need to import the `encoding/binary` standard Go package, because `/dev/random` returns binary data that needs to be decoded. The second code portion of `devRandom.go` follows:

```
func main() {
    f, err := os.Open("/dev/random")
    defer f.Close()

    if err != nil {
        fmt.Println(err)
        return
    }
}
```

You open `/dev/random` as usual because everything in UNIX is a file.

The last code segment of `devRandom.go` is shown in the following Go code:

```
var seed int64
binary.Read(f, binary.LittleEndian, &seed)
fmt.Println("Seed:", seed)
}
```

You need the `binary.Read()` function, which requires three parameters, in order to read from the `/dev/random` system device. The value of the second parameter (`binary.LittleEndian`) specifies that you want to use the **little endian** byte order. The other option is `binary.BigEndian`, which is used when your computer is using the **big endian** byte order.

Executing `devRandom.go` will generate the following type of output:

```
$ go run devRandom.go
Seed: -2044736418491485077
$ go run devRandom.go
Seed: -5174854372517490328
$ go run devRandom.go
Seed: 7702177874251412774
```

Reading a specific amount of data

In this section, you will learn how to read exactly the amount of data you want. This technique is particularly useful when reading binary files, where you have to decode the data you read in a particular way. Nevertheless, this technique still works with text files.

The logic behind this technique is as follows: you create a **byte slice** with the size you need and use that byte slice for reading. To make this more interesting, this functionality is going to be implemented as a function with two parameters. One parameter will be used to specify the amount of data that you want to read, and the other parameter, which will have the `*os.File` type, will be used to access the desired file. The return value of that function will be the data you have read.

The name of the Go program for this topic will be `readSize.go` and it will be presented in four parts. The utility will accept a single parameter, which will be the size of the byte slice.



This particular program, when used with the presented technique, can help you to copy any file using the buffer size you want.

The first part of `readSize.go` has the expected preamble:

```
package main

import (
    "fmt"
    "io"
```

```
    "os"  
    "strconv"  
)
```

The second part of `readSize.go` contains the following Go code:

```
func readSize(f *os.File, size int) []byte {  
    buffer := make([]byte, size)  
  
    n, err := f.Read(buffer)  
    if err == io.EOF {  
        return nil  
    }  
  
    if err != nil {  
        fmt.Println(err)  
        return nil  
    }  
  
    return buffer[0:n]  
}
```

This is the function discussed earlier. Although the code is straightforward, there is one point that needs an explanation. The `io.Reader.Read()` method returns two parameters: the number of bytes read and an error variable. The `readSize()` function uses the former return value of `io.Read()` in order to return a byte slice of that size. Although this is a tiny detail, and it is only significant when you reach the end of the file, it ensures that the output of the utility will be same as the input and that it will not contain any extra characters.

Finally, there is code that checks for `io.EOF`, which is an error that signifies that you have reached the end of a file. When that kind of error occurs, the function returns. Dave Cheney, a project member for Go and open-source contributor, calls these errors "sentinel errors" because they signify that an error did not occur.

The third code portion of this utility is as follows:

```
func main() {  
    arguments := os.Args  
    if len(arguments) != 3 {  
        fmt.Println("<buffer size> <filename>")  
        return  
    }  
  
    bufferSize, err := strconv.Atoi(os.Args[1])  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
}
```

```
    }

    file := os.Args[2]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

The last code segment of `readSize.go` is as follows:

```
    for {
        readData := readSize(f, bufferSize)
        if readData != nil {
            fmt.Print(string(readData))
        } else {
            break
        }
    }
}
```

Here, you keep reading your input file until `readSize()` returns an error or `nil`.

Executing `readSize.go` by telling it to process a binary file and handling its output with `wc(1)` will validate the correctness of the program:

```
$ go run readSize.go 1000 /bin/ls | wc
   80   1032   38688
$ wc /bin/ls
   80   1032   38688 /bin/ls
```

The advantages of binary formats

In the previous section, the `readSize.go` utility illustrated how you can read a file byte by byte, which is a technique that best applies to binary files. So, you might ask, why read data in binary format when text formats are so much easier to understand? The main reason is space reduction. Imagine that you want to store the number 20 as a string to a file. It is easy to understand that you will need two bytes to store 20 using ASCII characters: one for storing 2 and another for storing 0.

Storing 20 in binary format requires just one byte, since 20 can be represented as 00010100 in binary or as 0x14 in hexadecimal.

This difference might look insignificant when you are dealing with small amounts of data, but it could be pretty substantial when dealing with data found in applications such as database servers.

Reading CSV files

CSV files are plain text files with a format. In this section, you will learn how to read a text file that contains points of a plane, which means that each line will contain a pair of coordinates. Additionally, you are also going to use an external Go library named **Glott**, which will help you to create a plot of the points that you read from the CSV file. Note that Glott uses **Gnuplot**, which means that you will need to install Gnuplot on your UNIX machine in order to use Glott.

The name of the source file for this topic is `CSVplot.go`, and it is going to be presented in five parts. The first code segment is as follows:

```
package main

import (
    "encoding/csv"
    "fmt"
    "github.com/Arafatk/glott"
    "os"
    "strconv"
)
```

The second part of `CSVplot.go` is shown in the following Go code:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a data file!")
        return
    }

    file := os.Args[1]
    _, err := os.Stat(file)
    if err != nil {
        fmt.Println("Cannot stat", file)
        return
    }
}
```

In this part, you can see a technique for checking whether a file already exists or not using the powerful `os.Stat()` function.

The third part of `CSVplot.go` is as follows:

```
f, err := os.Open(file)
if err != nil {
    fmt.Println("Cannot open", file)
    fmt.Println(err)
    return
}
defer f.Close()

reader := csv.NewReader(f)
reader.FieldsPerRecord = -1
allRecords, err := reader.ReadAll()
if err != nil {
    fmt.Println(err)
    return
}
```

The fourth code segment of `CSVplot.go` is shown in the following Go code:

```
xP := []float64{}
yP := []float64{}
for _, rec := range allRecords {
    x, _ := strconv.ParseFloat(rec[0], 64)
    y, _ := strconv.ParseFloat(rec[1], 64)
    xP = append(xP, x)
    yP = append(yP, y)
}

points := [][]float64{}
points = append(points, xP)
points = append(points, yP)
fmt.Println(points)
```

Here, you convert the string values you read into numbers and put them into a slice with two dimensions named `points`.

The last part of `CSVplot.go` contains the following Go code:

```
dimensions := 2
persist := true
debug := false
plot, _ := glot.NewPlot(dimensions, persist, debug)

plot.SetTitle("Using Glot with CSV data")
plot.SetXLabel("X-Axis")
plot.SetYLabel("Y-Axis")
style := "circle"
plot.AddPointGroup("Circle:", style, points)
plot.SavePlot("output.png")
}
```

In the preceding Go code, you saw how you can create a PNG file with the help of the Glot library and its `glot.SavePlot()` function.

As you might guess, you will need to download the Glot library before being able to compile and execute the `CSVplot.go` source code, which requires the execution of the following command from your favorite UNIX shell:

```
$ go get github.com/Arafatk/glot
```

The CSV data file containing the points that will be plotted has the following format:

```
$ cat /tmp/dataFile
1,2
2,3
3,3
4,4
5,8
6,5
-1,12
-2,10
-3,10
-4,10
```

Executing `CSVplot.go` will generate the following kind of output:

```
$ go run CSVplot.go /tmp/doesNotExist
Cannot stat /tmp/doesNotExist
$ go run CSVplot.go /tmp/dataFile
[[1 2 3 4 5 6 -1 -2 -3 -4] [2 3 3 4 8 5 12 10 10 10]]
```

You can see the results of `CSVplot.go` in a much better format in the following figure:

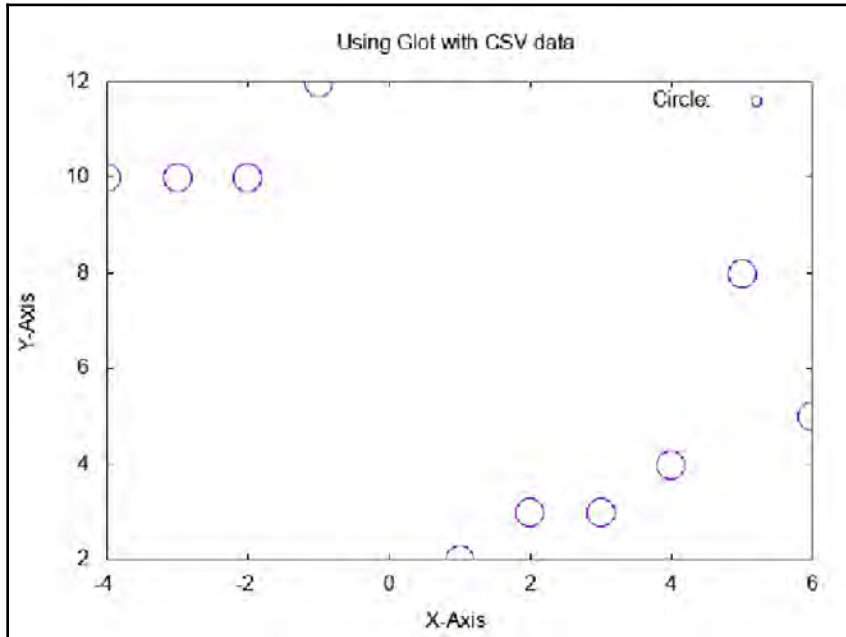


Figure 8.1: The type of graphical output you can get from Glot

Writing to a file

Generally speaking, you can use the functionality of the `io.Writer` interface for writing data to files on a disk. Nevertheless, the Go code of `save.go` will show you five ways to write data to a file. The `save.go` program will be presented in six parts.

The first part of `save.go` is as follows:

```
package main

import (
```

```
    "bufio"  
    "fmt"  
    "io"  
    "io/ioutil"  
    "os"  
)
```

The second code portion of `save.go` is shown in the following Go code:

```
func main() {  
    s := []byte("Data to write\n")  
  
    f1, err := os.Create("f1.txt")  
    if err != nil {  
        fmt.Println("Cannot create file", err)  
        return  
    }  
    defer f1.Close()  
    fmt.Fprintf(f1, string(s))  
}
```

Notice that the `s` byte slice will be used in every line that involves writing presented in this Go program. Additionally, the `fmt.Fprintf()` function used here can help you to write data to your own log files using the format you want. In this case, `fmt.Fprintf()` writes your data to the file identified by `f1`.

The third part of `save.go` contains the following Go code:

```
f2, err := os.Create("f2.txt")  
if err != nil {  
    fmt.Println("Cannot create file", err)  
    return  
}  
defer f2.Close()  
n, err := f2.WriteString(string(s))  
fmt.Printf("wrote %d bytes\n", n)
```

In this case, `f2.WriteString()` is used for writing your data to a file.

The fourth code segment of `save.go` is next:

```
f3, err := os.Create("f3.txt")  
if err != nil {  
    fmt.Println(err)  
    return  
}  
w := bufio.NewWriter(f3)  
n, err = w.WriteString(string(s))
```

```
fmt.Printf("wrote %d bytes\n", n)
w.Flush()
```

In this case, `bufio.NewWriter()` opens a file for writing and `bufio.WriteString()` writes the data.

The fifth part of `save.go` will teach you another method for writing to a file:

```
f4 := "f4.txt"
err = ioutil.WriteFile(f4, s, 0644)
if err != nil {
    fmt.Println(err)
    return
}
```

This method needs just a single function call named `ioutil.WriteFile()` for writing your data, and it does not require the use of `os.Create()`.

The last code segment of `save.go` is as follows:

```
f5, err := os.Create("f5.txt")
if err != nil {
    fmt.Println(err)
    return
}
n, err = io.WriteString(f5, string(s))
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("wrote %d bytes\n", n)
}
```

The last technique uses `io.WriteString()` to write the desired data to a file.

Executing `save.go` will create the following type of output:

```
$ go run save.go
wrote 14 bytes
wrote 14 bytes
wrote 14 bytes
$ ls -l f?.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f1.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f2.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f3.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f4.txt
-rw-r--r-- 1 mtsouk staff 14 Jan 23 20:30 f5.txt
$ cat f?.txt
```

```
Data to write  
Data to write  
Data to write  
Data to write  
Data to write
```

The next section will show you how to save data to a file with the help of a specialized function of a package that is in the standard Go library.

Loading and saving data on disk

Do you remember the `keyValue.go` application from Chapter 4, *The Uses of Composite Types*? Well, it was far from perfect, so in this section you will learn how to save the data of a **key-value store** on disk and how to load it back into memory when you next start your application.

We are going to create two new functions, one named `save()` for saving data to disk and another named `load()` for retrieving the data from disk. Thus, we will only present the code differences between `keyValue.go` and `kvSaveLoad.go` using the `diff(1)` UNIX command-line utility.



The `diff(1)` UNIX command-line utility can be very convenient when you want to spot the differences between two text files. You can learn more about it by executing `man 1 diff` at the command line of your favorite UNIX shell.

If you think about the task to be implemented here, you will recognize that what you really need is an easy way to save the contents of a Go map to disk, as well as a way to load the data from a file and put it into a Go map.

The process of converting your data into a stream of bytes is called **serialization**. The process of reading a data file and converting it into an object is called **deserialization**. The `encoding/gob` standard Go package will be used for the `kvSaveLoad.go` program. It will help the serialization and deserialization process. The `encoding/gob` package uses the **gob format** to store its data. The official name for such formats is **stream encoding**. The good thing about the gob format is that Go does all of the dirty work, so you do not have to worry about the encoding and decoding stages.

Other Go packages that can help you to serialize and deserialize data are `encoding/xml`, which uses the **XML format**, and `encoding/json`, which uses the **JSON format**.

The following output reveals the code changes between `kvSaveLoad.go` and `keyValue.go` without including the implementations of the `save()` and `load()` functions, which will be fully presented shortly:

```
$ diff keyValue.go kvSaveLoad.go
4a5
>     "encoding/gob"
16a18,55
> var DATAFILE = "/tmp/dataFile.gob"
> func save() error {
>
>     return nil
> }
>
> func load() error {
>
> }
59a99,104
>
>     err := load()
>     if err != nil {
>         fmt.Println(err)
>     }
>
88a134,137
>         err = save()
>         if err != nil {
>             fmt.Println(err)
>         }
```

An important part of the `diff(1)` output is the definition of the `DATAFILE` global variable, which holds the path to the file that is used by the key-value store. Apart from this, you can see where the `load()` function is called, as well as the point where the `save()` function is called. The `load()` function is used first in the `main()` function, while the `save()` function is executed when the user issues the `STOP` command.

Each time you execute `kvSaveLoad.go`, the program checks whether there is data to be read by trying to read the default data file. If there is no data file to read, you will start with an empty key-value store. When the program is about to terminate, it writes all of its data on the disk using the `save()` function.

The `save()` function has the following implementation:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
```



```
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}
```

Note that the first thing the `save()` function does is to delete the existing data file using the `os.Remove()` function in order to create it later on.

One of the most critical tasks the `save()` function does is to make sure that you can actually create and write to the desired file. Although there are many ways to do this, the `save()` function uses the simplest way, which is checking the error value returned by the `os.Create()` function. If that value is not `nil`, then there is a problem and the `save()` function finishes without saving any data.

The `load()` function is implemented as follows:

```
func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}
```

One of the tasks of the `load()` function is to make sure that the file that you are trying to read is actually there and that you can read it without any problems.

Once again, the `load()` function uses the simplest approach, which is to look at the return value of the `os.Open()` function. If the `error` value returned is equal to `nil`, then everything is fine.

It is also important to close the file after reading the data from it, as it will be overwritten by the `save()` function later on. The release of the file is accomplished by the `deferloadFrom.Close()` statement.

Executing `kvSaveLoad.go` will generate the following type of output:

```
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
ADD 1 2 3
ADD 4 5 6
STOP
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 1 value: {2 3 }
key: 4 value: {5 6 }
DELETE 1
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
rMacBook:code mtsouk$ go run kvSaveLoad.go
Loading /tmp/dataFile.gob
PRINT
key: 4 value: {5 6 }
STOP
Saving /tmp/dataFile.gob
$ ls -l /tmp/dataFile.gob
-rw-r--r--  1 mtsouk  wheel   80 Jan 22 11:22 /tmp/dataFile.gob
$ file /tmp/dataFile.gob
/tmp/dataFile.gob: data
```

In Chapter 13, *Network Programming – Building Your Own Servers and Clients*, you are going to see the final version of the key-value store, which will be able to operate over a TCP/IP connection and will serve multiple network clients using **goroutines**.

The strings package revisited

We first talked about the handy `strings` package back in Chapter 4, *The Uses of Composite Types*. This section will address the functions of the `strings` package that are related to file input and output.

The first part of `str.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "io"
    "os"
    "strings"
)
```

The second code segment of `str.go` is as follows:

```
func main() {

    r := strings.NewReader("test")
    fmt.Println("r length:", r.Len())
```

The `strings.NewReader()` function creates a read-only `Reader` from a string. The `strings.Reader` object implements the `io.Reader`, `io.ReaderAt`, `io.Seeker`, `io.WriterTo`, `io.ByteScanner`, and `io.RuneScanner` interfaces.

The third part of `str.go` follows:

```
    b := make([]byte, 1)
    for {
        n, err := r.Read(b)
        if err == io.EOF {
            break
        }

        if err != nil {
            fmt.Println(err)
            continue
        }
        fmt.Printf("Read %s Bytes: %d\n", b, n)
    }
```

Here, you can see how to use `strings.Reader` as an `io.Reader` in order to read a string byte by byte using the `Read()` function.

The last code portion of `str.go` contains the following Go code:

```
s := strings.NewReader("This is an error!\n")
fmt.Println("r length:", s.Len())
n, err := s.WriteTo(os.Stderr)

if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Wrote %d bytes to os.Stderr\n", n)
}
```

In this code segment, you can see how you can write to standard error with the help of the `strings` package.

Executing `str.go` will generate the following output:

```
$ go run str.go
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
This is an error!
Wrote 18 bytes to os.Stderr
$ go run str.go 2>/dev/null
r length: 4
Read t Bytes: 1
Read e Bytes: 1
Read s Bytes: 1
Read t Bytes: 1
r length: 18
Wrote 18 bytes to os.Stderr
```

About the bytes package

The `bytes` standard Go package contains functions for working with **byte slices** in the same way that the `strings` standard Go package helps you to work with **strings**. The name of the Go source code file is `bytes.go`, and it will be presented in three code portions.

The first part of `bytes.go` follows:

```
package main

import (
    "bytes"
    "fmt"
    "io"
    "os"
)
```

The second code portion of `bytes.go` contains the following Go code:

```
func main() {
    var buffer bytes.Buffer
    buffer.Write([]byte("This is"))
    fmt.Fprintf(&buffer, " a string!\n")
    buffer.WriteTo(os.Stdout)
    buffer.WriteTo(os.Stdout)
}
```

First, you create a new `bytes.Buffer` variable and you put data into it using `buffer.Write()` and `fmt.Fprintf()`. Then, you call `buffer.WriteTo()` twice.

The first `buffer.WriteTo()` call will print the contents of the `buffer` variable. However, the second call to `buffer.WriteTo()` has nothing to print because the `buffer` variable will be empty after the first `buffer.WriteTo()` call.

The last part of `bytes.go` is as follows:

```
    buffer.Reset()
    buffer.Write([]byte("Mastering Go!"))
    r := bytes.NewReader([]byte(buffer.String()))
    fmt.Println(buffer.String())
    for {
        b := make([]byte, 3)
        n, err := r.Read(b)
        if err == io.EOF {
            break
        }

        if err != nil {
            fmt.Println(err)
            continue
        }
        fmt.Printf("Read %s Bytes: %d\n", b, n)
    }
}
```

The `Reset()` method resets the `buffer` variable and the `Write()` method puts some data into it again. Then, you create a new reader using `bytes.NewReader()`, and after that you use the `Read()` method of the `io.Reader` interface to read the data found in the `buffer` variable.

Executing `bytes.go` will create the following type of output:

```
$ go run bytes.go
This is a string!
Mastering Go!
Read Mas Bytes: 3
Read ter Bytes: 3
Read ing Bytes: 3
Read  Go Bytes: 3
Read ! Bytes: 1
```

File permissions

A popular topic in UNIX systems programming is UNIX file permissions. In this section, you will learn how to print the permissions of any file, provided that you have the required UNIX permission to do so. The name of the program is `permissions.go`, and it will be presented in three parts.

The first part of `permissions.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
)
```

The second code segment of `permissions.go` is shown in the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("usage: permissions filename\n")
        return
    }
}
```

The last part of this utility follows:

```
filename := arguments[1]
info, _ := os.Stat(filename)
```

```
mode := info.Mode()
fmt.Println(filename, "mode is", mode.String()[1:10])
}
```

The call to `os.Stat(filename)` returns a big structure with lots of data. As we are only interested in the permissions of the file, we call the `Mode()` method and print its output. Actually, we are printing a part of the output denoted by `mode.String()[1:10]` because this is where the data that interests us is found.

Executing `permissions.go` will create the following type of output:

```
$ go run permissions.go /tmp/adobegc.log
/tmp/adobegc.log mode is rw-rw-rw-
$ go run permissions.go /dev/random
/dev/random mode is crw-rw-rw-
```

The output of the `ls(1)` utility verifies the correctness of `permissions.go`:

```
$ ls -l /dev/random /tmp/adobegc.log
crw-rw-rw- 1 root wheel 14, 0 Jan 8 20:24 /dev/random
-rw-rw-rw- 1 root wheel 583454 Jan 16 19:12 /tmp/adobegc.log
```

Handling UNIX signals

Go provides the `os/signal` package to help developers to work with signals. This section will show you how to use it for UNIX signal handling.

First, let me present some useful information about UNIX signals. Have you ever pressed Ctrl+C in order to stop a running program? If your answer is "yes," then you are already familiar with signals because Ctrl+C sends the **SIGINT signal** to a program. Strictly speaking, UNIX **signals** are software interrupts that can be accessed either by name or by number, and they offer a way to handle asynchronous events on a UNIX system. Generally speaking, it is safer to send a signal by name because you are less likely to send the wrong signal accidentally.

A program cannot handle all of the available signals. Some signals cannot be caught, but nor can they be ignored. The `SIGKILL` and `SIGSTOP` signals cannot be caught, blocked, or ignored. The reason for this is that they provide the kernel and the root user with a way of stopping any process that they desire. The `SIGKILL` signal, which is also known by the number 9, is usually called in extreme conditions where you need to act fast. Thus, it is the only signal that is usually called by number, simply because it is quicker to do so.



`signal.SIGINFO` is not available on Linux machines, which means that if you find it in a Go program that you want to run on a Linux machine, you need to replace it with another signal, or your Go program will not be able to compile and execute.

The most common way to send a signal to a process is by using the `kill(1)` utility. By default, `kill(1)` sends the `SIGTERM` signal. If you want to find all of the supported signals on your UNIX machine, you should execute the `kill -l` command.

If you try to send a signal to a process without having the required permissions, `kill(1)` will not do the job and you will get an error message similar to the following:

```
$ kill 1210
-bash: kill: (1210) - Operation not permitted
```

Handling two signals

In this subsection, you will learn how to handle two signals in a Go program using the code found in `handleTwo.go`, which will be presented in four parts. The signals that will be handled by `handleTwo.go` are `SIGINFO` and `SIGINT`, which in Go are named `syscall.SIGINFO` and `os.Interrupt`, respectively.



If you look at the documentation of the `os` package, you will find that the only two signals that are guaranteed to be present on all systems are `syscall.SIGKILL` and `syscall.SIGINT`, which in Go are also defined as `os.Kill` and `os.Interrupt`, respectively.

The first part of `handleTwo.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)
```

The second part of the `handleTwo.go` program follows:

```
func handleSignal(signal os.Signal) {
    fmt.Println("handleSignal() Caught:", signal)
}
```


The `handleSignal()` function will be used for handling the `syscall.SIGINFO` signal, while the `os.Interrupt` signal will be handled inline.

The third code segment of `handleTwo.go` is shown in the following Go code:

```
func main() {
    sigs := make(chan os.Signal, 1)
    signal.Notify(sigs, os.Interrupt, syscall.SIGINFO)
    go func() {
        for {
            sig := <-sigs
            switch sig {
            case os.Interrupt:
                fmt.Println("Caught:", sig)
            case syscall.SIGINFO:
                handleSignal(sig)
                return
            }
        }
    }()
}
```

This technique works as follows: first, you define a channel named `sigs` that helps you to pass data around. Then, you call `signal.Notify()` in order to state the signals that interest you. Next, you implement an **anonymous function** that runs as a goroutine in order to act when you receive any one of the signals you care about. You will have to wait for Chapter 9, *Concurrency in Go – Goroutines, Channels, and Pipelines*, to learn more about goroutines and channels.

The last portion of the `handleTwo.go` program is as follows:

```
for {
    fmt.Printf(".")
    time.Sleep(20 * time.Second)
}
}
```

The `time.Sleep()` call is used to prohibit the program from terminating, as it has no real work to do. In an actual application, there would be no need to use similar code.

As we need the process ID of a program in order to send signals to it using the `kill(1)` utility, we will first compile `handleTwo.go` and run the executable file instead of using `go run handleTwo.go`. Working with `handleTwo` will generate the following type of output:

```
$ go build handleTwo.go
$ ls -l handleTwo
-rwxr-xr-x 1 mtsouk staff 2005200 Jan 18 07:49 handleTwo
```

```
$ ./handleTwo
.^CCaught: interrupt
.Caught: interrupt
handleSignal() Caught: information request
.Killed: 9
```

Note that you will need an additional terminal in order to interact with `handleTwo.go` and obtain the preceding output. You will execute the following commands in that terminal:

```
$ ps ax | grep ./handleTwo | grep -v grep
47988 s003  S+      0:00.00 ./handleTwo
$ kill -s INT 47988
$ kill -s INFO 47988
$ kill -s USR1 47988
$ kill -9 47988
```

The first command is used for finding the process ID of the `handleTwo` executable, while the remaining commands are used for sending the desired signals to that process. The `SIGUSR1` signal is ignored and it does not appear in the output.

The problem with `handleTwo.go` is that if it gets a signal that it is not programmed to handle, it will ignore it. Thus, in the next section, you will see a technique that uses a relatively different approach in order to handle signals in a more efficient way.

Handling all signals

In this subsection, you will learn how to handle all signals but respond only to the ones that really interest you. This is a much better and safer technique than the one presented in the previous subsection. The technique will be illustrated using the Go code of `handleAll.go`, which will be presented in four parts.

The first part of `handleAll.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
    "os/signal"
    "syscall"
    "time"
)

func handle(signal os.Signal) {
    fmt.Println("Received:", signal)
```

```
}
```

The second code segment from `handleAll.go` is shown in the following Go code:

```
func main() {  
    sigs := make(chan os.Signal, 1)  
    signal.Notify(sigs)
```

So, all the magic happens due to the `signal.Notify(sigs)` statement. As no signals are specified, all incoming signals will be handled.



You are allowed to call `signal.Notify()` multiple times in the same program using different channels and the same signals. In that case, each relevant channel will receive a copy of the signals that it was programmed to handle.

The third code portion of the `handleAll.go` utility follows:

```
go func() {  
    for {  
        sig := <-sigs  
        switch sig {  
        case os.Interrupt:  
            handle(sig)  
        case syscall.SIGTERM:  
            handle(sig)  
            os.Exit(0)  
        case syscall.SIGUSR2:  
            fmt.Println("Handling syscall.SIGUSR2!")  
        default:  
            fmt.Println("Ignoring:", sig)  
        }  
    }  
}()
```

It is very convenient to use one of the signals in order to exit your program. This gives you the opportunity to do some housekeeping in your program when needed. In this case, the `syscall.SIGTERM` signal is used for that purpose. This does not prevent you from using `SIGKILL` to kill a program, though.

The remaining Go code for `handleAll.go` follows:

```
    for {  
        fmt.Printf(".")  
        time.Sleep(30 * time.Second)  
    }  
}
```

You still need to call `time.Sleep()` to prevent your program from terminating immediately.

Again, it would be better to build an executable file for `handleAll.go` using the `go build` tool. Executing `handleAll` and interacting with it from another terminal will generate the following type of output:

```
$ go build handleAll.go
$ ls -l handleAll
-rwxr-xr-x 1 mtsouk staff 2005216 Jan 18 08:25 handleAll
$ ./handleAll
.Ignoring: hangup
Handling syscall.SIGUSR2!
Ignoring: user defined signal 1
Received: interrupt
^CReceived: interrupt
Received: terminated
```

The commands issued from the second terminal are as follows:

```
$ ps ax | grep ./handleAll | grep -v grep
49902 s003 S+      0:00.00 ./handleAll
$ kill -s HUP 49902
$ kill -s USR2 49902
$ kill -s USR1 49902
$ kill -s INT 49902
$ kill -s TERM 49902
```

Programming UNIX pipes in Go

According to the UNIX philosophy, UNIX command-line utilities should do one job and perform that job well. In practice, this means that instead of developing huge utilities that do lots of jobs, you should develop multiple smaller programs, which, when combined, should perform the desired job. The most common way for two or more UNIX command-line utilities to communicate is by using pipes. In a **UNIX pipe**, the output of a command-line utility becomes the input of another command-line utility. This process may involve more than two programs. The symbol that is used for UNIX pipes is the `|` character.

Pipes have two serious limitations: firstly, they usually communicate in one direction, and secondly, they can only be used between processes that have a common ancestor. The general idea behind the implementation of UNIX pipes is that if you do not have a file to process, you should wait to get your input from standard input.

Similarly, if you are not told to save your output to a file, you should write your output to standard output, either for the user to see it or for another program to process it.

In Chapter 1, *Go and the Operating System*, you learned how to read from standard input and how to write to standard output. If you have doubts about these two operations, it would be a good time to review the Go code of `stdOUT.go` and `stdIN.go`.

Implementing the `cat(1)` utility in Go

In this section, you will see a Go version of the `cat(1)` utility. You will most likely be surprised by the length of the program. The source code of `cat.go` will be presented in three parts. The first part of `cat.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "io"
    "os"
)
```

The second code segment of `cat.go` contains the following Go code:

```
func printFile(filename string) error {
    f, err := os.Open(filename)
    if err != nil {
        return err
    }
    defer f.Close()
    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        io.WriteString(os.Stdout, scanner.Text())
        io.WriteString(os.Stdout, "\n")
    }
    return nil
}
```

In this part, you can see the implementation of a function whose purpose it is to print the contents of a file in the standard output.

The last part of `cat.go` is as follows:

```
func main() {
    filename := ""
    arguments := os.Args
    if len(arguments) == 1 {
        io.Copy(os.Stdout, os.Stdin)
        return
    }

    for i := 1; i < len(arguments); i++ {
        filename = arguments[i]
        err := printFile(filename)
        if err != nil {
            fmt.Println(err)
        }
    }
}
```

The preceding code contains all of the magic of `cat.go`, because this is where you define how the program will behave. First of all, if you execute `cat.go` without any command-line arguments, then the program will just copy standard input to standard output as defined by the `io.Copy(os.Stdout, os.Stdin)` statement. However, if there are command-line arguments, then the program will process them all in the same order that they were given.

Executing `cat.go` will create the following type of output:

```
$ go run cat.go
Mastering Go!
Mastering Go!
1 2 3 4
1 2 3 4
```

However, things get really interesting if you execute `cat.go` using UNIX pipes:

```
$ go run cat.go /tmp/1.log /tmp/2.log | wc
2367  44464 279292
$ go run cat.go /tmp/1.log /tmp/2.log | go run cat.go | wc
2367  44464 279292
```

`cat.go` is also able to print multiple files on your screen:

```
$ go run cat.go 1.txt 1 1.txt
2367  44464 279292
2367  44464 279292
open 1: no such file or directory
```

```
2367    44464    279292
2367    44464    279292
```

Please note that if you try to execute `cat.go` as `go run cat.go cat.go` and expect that you will get the contents of `cat.go` on your screen, the process will fail and you will get the following error message instead:

```
package main: case-insensitive file name collision: "cat.go" and "cat.go"
```

The reason for this is that Go does not understand that the second `cat.go` should be used as a command-line argument to the `go run cat.go` command. Instead, `go run` tries to compile `cat.go` twice, which causes the error message. The solution to this problem is to execute `go build cat.go` first, and then use `cat.go` or any other Go source file as the argument to the generated binary executable file.

About syscall.PtraceRegs

You might have assumed that you are done dealing with the `syscall` standard Go package, but you are mistaken! In this section, we will work with `syscall.PtraceRegs`, which is a structure that holds information about the state of the registers.

You will now learn how to print the values of all of the following registers on your screen using the Go code of `ptraceRegs.go`, which will be presented in four parts. The star of the `ptraceRegs.go` utility is the `syscall.PtraceGetRegs()` function - there are also the `syscall.PtraceSetRegs()`, `syscall.PtraceAttach()`, `syscall.PtracePeekData()`, and `syscall.PtracePokeData()` functions that can help you to work with registers, but these functions will not be used in `ptraceRegs.go`.

The first part of the `ptraceRegs.go` utility follows:

```
package main

import (
    "fmt"
    "os"
    "os/exec"
    "syscall"
    "time"
)
```

The second code portion of `ptraceRegs.go` is shown in the following Go code:

```
func main() {
```

```
var r syscall.PtraceRegs
cmd := exec.Command(os.Args[1], os.Args[2:]...)

cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr
```

The last two statements redirect the standard output and standard error from the executed command to the UNIX standard output and standard error, respectively.

The third part of `ptraceRegs.go` contains the following Go code:

```
cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}
err := cmd.Start()
if err != nil {
    fmt.Println("Start:", err)
    return
}

err = cmd.Wait()
fmt.Printf("State: %v\n", err)
wpid := cmd.Process.Pid
```

In the preceding Go code, you call an external command, which is specified in the command-line arguments of the program, and you find its process ID, which will be used in the `syscall.PtraceGetRegs()` call. The `&syscall.SysProcAttr{Ptrace: true}` statement specifies that you want to use `ptrace` on the child process.

The last code segment of `ptraceRegs.go` follows:

```
err = syscall.PtraceGetRegs(wpid, &r)
if err != nil {
    fmt.Println("PtraceGetRegs:", err)
    return
}
fmt.Printf("Registers: %#v\n", r)
fmt.Printf("R15=%d, Gs=%d\n", r.R15, r.Gs)

time.Sleep(2 * time.Second)
}
```

Here, you call `syscall.PtraceGetRegs()` and you print the results that are stored in the `r` variable, which should be passed as a pointer.

Executing `ptraceRegs.go` on a macOS Mojave machine will generate the following output:

```
$ go run ptraceRegs.go
# command-line-arguments
./ptraceRegs.go:11:8: undefined: syscall.PtraceRegs
./ptraceRegs.go:14:9: undefined: syscall.PtraceGetRegs
```

This means that this program will not work on machines running macOS and Mac OS X.

Executing `ptraceRegs.go` on a Debian Linux machine will create the following output:

```
$ go version
go version go1.7.4 linux/amd64
$ go run ptraceRegs.go echo "Mastering Go!"
State: stop signal: trace/breakpoint trap
Registers: syscall.PtraceRegs{R15:0x0, R14:0x0, R13:0x0, R12:0x0, Rbp:0x0,
Rbx:0x0, R11:0x0, R10:0x0, R9:0x0, R8:0x0, Rax:0x0, Rcx:0x0, Rdx:0x0,
Rsi:0x0, Rdi:0x0, Orig_rax:0x3b, Rip:0x7f4045f81c20, Cs:0x33, Eflags:0x200,
Rsp:0x7ffe1905b070, Ss:0x2b, Fs_base:0x0, Gs_base:0x0, Ds:0x0, Es:0x0,
Fs:0x0, Gs:0x0}
R15=0, Gs=0
Mastering Go!
```

You can also find the list of registers on the documentation page of the `syscall` package.

Tracing system calls

This section will present a pretty advanced technique that uses the `syscall` package and allows you to monitor the system calls executed in a Go program.

The name of the Go utility is `traceSyscall.go`, and it is going to be presented in five code segments. The first part of `traceSyscall.go` follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
    "os/exec"
    "strings"
    "syscall"
)
```

```
var maxSyscalls = 0

const SYSCALLFILE = "SYSCALLS"
```

You will learn more about the purpose of the SYSCALLFILE variable in a short while.

The second code segment from `traceSyscall.go` is the following:

```
func main() {
    var SYSTEMCALLS []string
    f, err := os.Open(SYSCALLFILE)
    defer f.Close()
    if err != nil {
        fmt.Println(err)
        return
    }

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        line := scanner.Text()
        line = strings.Replace(line, " ", "", -1)
        line = strings.Replace(line, "SYS_", "", -1)
        temp := strings.ToLower(strings.Split(line, "=")[0])
        SYSTEMCALLS = append(SYSTEMCALLS, temp)
        maxSyscalls++
    }
}
```

Note that the information of the SYSCALLS text file is taken from the documentation of the `syscall` package, and it associates each system call with a number, which is the internal Go representation of the system call. This file is mainly used for printing the names of the system calls used by the program that is being traced.

The format of the SYSCALLS text file is as follows:

```
SYS_READ = 0
SYS_WRITE = 1
SYS_OPEN = 2
SYS_CLOSE = 3
SYS_STAT = 4
```

After reading the text file, the program creates a slice named `SYSTEMCALLS` for storing that information.

The third part of `traceSyscall.go` is as follows:

```
COUNTER := make([]int, maxSyscalls)
var regs syscall.PtraceRegs
cmd := exec.Command(os.Args[1], os.Args[2:]...)

cmd.Stdin = os.Stdin
cmd.Stdout = os.Stdout
cmd.Stderr = os.Stderr
cmd.SysProcAttr = &syscall.SysProcAttr{Ptrace: true}

err = cmd.Start()
err = cmd.Wait()
if err != nil {
    fmt.Println("Wait:", err)
}

pid := cmd.Process.Pid
fmt.Println("Process ID:", pid)
```

The `COUNTER` slice stores the number of times each system call is found in the program that is being traced.

The fourth code segment of `traceSyscall.go` contains the following Go code:

```
before := true
forCount := 0
for {
    if before {
        err := syscall.PtraceGetRegs(pid, &regs)
        if err != nil {
            break
        }
        if regs.Orig_rax > uint64(maxSyscalls) {
            fmt.Println("Unknown:", regs.Orig_rax)
            return
        }

        COUNTER[regs.Orig_rax]++
        forCount++
    }

    err = syscall.PtraceSyscall(pid, 0)
    if err != nil {
        fmt.Println("PtraceSyscall:", err)
        return
    }
}
```

```

_, err = syscall.Wait4(pid, nil, 0, nil)
if err != nil {
    fmt.Println("Wait4:", err)
    return
}
before = !before
}

```

The `syscall.PtraceSyscall()` function tells Go to continue the execution of the program that is being traced, but to stop when that program enters or exits a system call, which is exactly what we want! As each system call is traced before being called, right after it has finished its job, we use the `before` variable in order to count each system call only once.

The last part of `traceSyscall.go` follows:

```

for i, x := range COUNTER {
    if x != 0 {
        fmt.Println(SYSTEMCALLS[i], "->", x)
    }
}
fmt.Println("Total System Calls:", forCount)
}

```

In this part, we print the contents of the `COUNTER` slice. The `SYSTEMCALLS` slice is used here for finding out the name of a system call when we know its numerical Go representation.

Executing `traceSyscall.go` on a macOS Mojave machine will create the following output:

```

$ go run traceSyscall.go
# command-line-arguments
./traceSyscall.go:36:11: undefined: syscall.PtraceRegs
./traceSyscall.go:57:11: undefined: syscall.PtraceGetRegs
./traceSyscall.go:70:9: undefined: syscall.PtraceSyscall

```

Once again, the `traceSyscall.go` utility will not run on macOS and Mac OS X.

Executing the same program on a Debian Linux machine will create the following output:

```

$ go run traceSyscall.go ls /tmp/
Wait: stop signal: trace/breakpoint trap
Process ID: 5657
go-build084836422 test.go upload_progress_cache
read -> 11
write -> 1
open -> 37

```

```

close -> 27
stat -> 1
fstat -> 25
mmap -> 39
mprotect -> 16
munmap -> 4
brk -> 3
rt_sigaction -> 2
rt_sigprocmask -> 1
ioctl -> 2
access -> 9
execve -> 1
getdents -> 2
getrlimit -> 1
statfs -> 2
arch_prctl -> 1
futex -> 1
set_tid_address -> 1
openat -> 1
set_robust_list -> 1
Total System Calls: 189

```

At the end of the program, `traceSyscall.go` prints the number of times each system call was called in the program. The correctness of `traceSyscall.go` is verified by the output of the `strace -c` utility:

```

$ strace -c ls /tmp
test.go upload_progress_cache
% time   seconds  usecs/call   calls   errors syscall
-----
 0.00    0.000000      0         11         read
 0.00    0.000000      0          1         write
 0.00    0.000000      0         37        13 open
 0.00    0.000000      0         27         close
 0.00    0.000000      0          1         stat
 0.00    0.000000      0         25         fstat
 0.00    0.000000      0         39         mmap
 0.00    0.000000      0         16         mprotect
 0.00    0.000000      0          4         munmap
 0.00    0.000000      0          3         brk
 0.00    0.000000      0          2         rt_sigaction
 0.00    0.000000      0          1         rt_sigprocmask
 0.00    0.000000      0          2         ioctl
 0.00    0.000000      0          9         access
 0.00    0.000000      0          1         execve
 0.00    0.000000      0          2         getdents
 0.00    0.000000      0          1         getrlimit
 0.00    0.000000      0          2         statfs

```

0.00	0.000000	0	1	arch_prctl
0.00	0.000000	0	1	futex
0.00	0.000000	0	1	set_tid_address
0.00	0.000000	0	1	openat
0.00	0.000000	0	1	set_robust_list

100.00	0.000000		189	24 total

User ID and group ID

In this section, you will learn how to find the user ID of the current user, as well as the group IDs to which the current user belongs. Both the user ID and group IDs are positive integers kept in UNIX system files.

The name of the utility is `ids.go`, and it will be presented in two parts. The first part of the utility follows:

```
package main

import (
    "fmt"
    "os"
    "os/user"
)

func main() {
    fmt.Println("User id:", os.Getuid())
}
```

Finding the user ID of the current user is as simple as calling the `os.Getuid()` function.

The second part of `ids.go` is as follows:

```
var u *user.User
u, _ = user.Current()
fmt.Print("Group ids: ")
groupIDs, _ := u.GroupIds()
for _, i := range groupIDs {
    fmt.Print(i, " ")
}
fmt.Println()
}
```

On the other hand, finding the group IDs to which a user belongs is a much trickier task.

Executing `ids.go` will generate the following type of output:

```
$ go run ids.go
User id: 501
Group ids: 20 701 12 61 79 80 81 98 33 100 204 250 395 398 399
```

The Docker API and Go

If you work with Docker, you will find this section particularly handy as it will teach you how to communicate with Docker using Go and the **Docker API**.

The `dockerAPI.go` utility, which will be presented in four parts, implements the `docker ps` and the `docker image ls` commands. The first command lists all running containers, whereas the second command lists all available images on the local machine.

The first part of `dockerAPI.go` is as follows:

```
package main

import (
    "fmt"
    "github.com/docker/docker/api/types"
    "github.com/docker/docker/client"
    "golang.org/x/net/context"
)
```

As `dockerAPI.go` requires lots of external packages, it would be a good idea to execute it using Go modules; therefore, execute `export GO111MODULE=on` before running `dockerAPI.go` for the first time. This will also save you from having to manually download all of the required packages.

The second part of `dockerAPI.go` contains the following Go code:

```
func listContainers() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    containers, err := cli.ContainerList(context.Background(),
types.ContainerListOptions{})
    if err != nil {
        return (err)
    }
}
```

```

    for _, container := range containers {
        fmt.Println("Images:", container.Image, "with ID:", container.ID)
    }
    return nil
}

```

The definition of the `types.Container`

(<https://godoc.org/github.com/docker/docker/api/types#Container>) structure, which is returned by `ContainerList()`, is as follows:

```

type Container struct {
    ID          string `json:"Id"`
    Names       []string
    Image       string
    ImageID     string
    Command     string
    Created     int64
    Ports       []Port
    SizeRw      int64 `json:",omitempty"`
    SizeRootFs  int64 `json:",omitempty"`
    Labels      map[string]string
    State       string
    Status      string
    HostConfig  struct {
        NetworkMode string `json:",omitempty"`
    }
    NetworkSettings *SummaryNetworkSettings
    Mounts          []MountPoint
}

```

Should you wish to find any other information about the list of running Docker images (containers), you should make use of the other fields of the `types.Container` structure.

The third part of `dockerAPI.go` is as follows:

```

func listImages() error {
    cli, err := client.NewEnvClient()
    if err != nil {
        return (err)
    }

    images, err := cli.ImageList(context.Background(),
        types.ImageListOptions{})
    if err != nil {
        return (err)
    }

    for _, image := range images {

```



```

        fmt.Printf("Images %s with size %d\n", image.RepoTags,
image.Size)
    }
    return nil
}

```

The definition of the `types.ImageSummary`

(<https://godoc.org/github.com/docker/docker/api/types#ImageSummary>) structure, which is the data type of the slice that is returned by `ImageList()`, is as follows:

```

type ImageSummary struct {
    Containers int64 `json:"Containers"`
    Created int64 `json:"Created"`
    ID string `json:"Id"`
    Labels map[string]string `json:"Labels"`
    ParentID string `json:"ParentId"`
    RepoDigests []string `json:"RepoDigests"`
    RepoTags []string `json:"RepoTags"`
    SharedSize int64 `json:"SharedSize"`
    Size int64 `json:"Size"`
    VirtualSize int64 `json:"VirtualSize"`
}

```

The last part of `dockerAPI.go` is the following:

```

func main() {
    fmt.Println("The available images are:")
    err := listImages()
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("The running Containers are:")
    err = listContainers()
    if err != nil {
        fmt.Println(err)
    }
}

```

Executing `dockerAPI.go` on my macOS Mojave machine will generate the following kind of output:

```

$ go run dockerAPI.go
The available images are:
Images [golang:1.12] with size 772436547
Images [landoop/kafka-lenses-dev:latest] with size 1379088343
Images [confluentinc/cp-kafka:latest] with size 568503378
Images [landoop/fast-data-dev:latest] with size 1052076831

```

```
The running Containers are:  
Images: landoop/kafka-lenses-dev with ID:  
90e1caaab43297810341290137186425878ef5891c787f6707c03be617862db5
```

If Docker is not available or if it is not running, you will get the following error message:

```
$ go run dockerAPI.go  
The available images are:  
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the  
docker daemon running?  
The running Containers are:  
Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the  
docker daemon running?
```

Additional resources

You will find the following web links very useful:

- Read the documentation page of the `io` package, which can be found at <https://golang.org/pkg/io/>.
- Hear from Dave Cheney on error handling: <https://dave.cheney.net/2016/04/27/dont-just-check-errors-handle-them-gracefully>.
- You can learn more about the **Glott** plotting library by visiting its official web page at <https://github.com/Arafatk/glott>.
- You can find more examples of the use of the Docker API in Go, Python, and HTTP at <https://docs.docker.com/develop/sdk/examples/>.
- You can learn more about the encoding/binary standard package by visiting <https://golang.org/pkg/encoding/binary/>.
- Check out the documentation page of the encoding/gob package, which can be found at <https://golang.org/pkg/encoding/gob/>.
- You can also watch <https://www.youtube.com/watch?v=JRFNIKUROPE> and <https://www.youtube.com/watch?v=w8nFRoFJ6EQ>.
- You can learn about **Endianness** in many places, including <https://en.wikipedia.org/wiki/Endianness>.
- Visit the documentation page of the `flag` package, which can be found at <https://golang.org/pkg/flag/>.

Exercises

- Write a Go program that takes three arguments: the name of a text file, and two strings. This utility should then replace every occurrence of the first string in the file with the second string. For reasons of security, the final output will be printed on the screen, which means that the original text file will remain intact.
- Use the `encoding/gob` package to serialize and deserialize a Go map, as well as a slice of structures.
- Create a Go program that handles any three signals you choose.
- Create a utility in Go that replaces all tab characters found in a text file with a given number of spaces, specified as a command-line parameter to the program. Once again, the output will be printed on the screen.
- Develop a utility that reads a text file line by line and removes the space characters from each line using the `strings.TrimSpace()` function.
- Modify `kvSaveLoad.go` in order to support a single command-line argument, which is the filename that you will use both to load and save your data.
- Can you create a Go version of the `wc(1)` utility? Look at the manual page of `wc(1)` to find out about the command-line options that it supports.
- Can you write a program that uses **Glot** to plot a function?
- Modify `traceSyscall.go` in order to display each system call at the time it is being traced.
- Modify `cat.go` just to do `io.Copy(os.Stdout, f)` in order to copy the contents of a file straight out, instead of scanning it all.
- Use the Docker API to write a utility that terminates all containers that begin with a given string.
- The `cobra` package also supports subcommands, which are commands associated with specific commands like `go run main.go command list`. Try to implement a utility with subcommands.
- You can also use `bufio.NewScanner()` and `bufio.ScanWords` to read a line word by word. Find out how and create a new version of the `byWord.go` utility.

Summary

This invaluable chapter talked about many interesting topics, including reading files, writing to files, using the Docker API, and using the `flag`, `cobra`, and `viper` packages. Nevertheless, there are many more topics related to systems programming not mentioned in this chapter, such as working with directories; copying, deleting, and renaming files; dealing with UNIX users, groups, and UNIX processes; changing UNIX file permissions; generating **sparse files**; file locking and creating; and using and rotating your own log files, as well as the information found in the structure returned by the `os.Stat()` call.

At the end of this chapter, I presented two advanced utilities written in Go. The first one allowed you to inspect the state of the registers, while the second one showed you a technique that allows you to trace the system calls of any program.

The next chapter will talk about goroutines, channels, and pipelines, which are unique and powerful Go features.

9

Concurrency in Go – Goroutines, Channels, and Pipelines

The previous chapter discussed systems programming in Go, including the Go functions and techniques that allow you to communicate with your operating system. Two of the areas of systems programming that were not covered in the previous chapter are concurrent programming and how to create and manage multiple threads. Both of these topics will be addressed in this chapter and the next one.

Go offers its own unique and innovative way of achieving concurrency, which comes in the form of **goroutines** and **channels**. Goroutines are the smallest Go entities that can be executed on their own in a Go program. Channels can get data from goroutines in a concurrent and efficient way. This allows goroutines to have a point of reference and they can communicate with each other. Everything in Go is executed using goroutines, which makes perfect sense since Go is a concurrent programming language by design. Therefore, when a Go program starts its execution, its single goroutine calls the `main()` function, which starts the actual program execution.

The contents and the code of this chapter will be pretty simple, and you should have no problem following and understanding them. I left the more advanced parts of goroutines and channels for [Chapter 10, Concurrency in Go – Advanced Topics](#).

In this chapter, you will learn about the following topics:

- The differences between processes, threads, and goroutines
- The Go scheduler
- Concurrency versus parallelism
- The concurrency models of Erlang and Rust
- Creating goroutines

- Creating channels
- Reading or receiving data from a channel
- Writing or sending data to a channel
- Creating pipelines
- Waiting for your goroutines to finish

About processes, threads, and goroutines

A **process** is an execution environment that contains instructions, user data, and system data parts, as well as other types of resources that are obtained during runtime, whereas a **program** is a file that contains instructions and data that are used for initializing the instruction and user-data parts of a process.

A **thread** is a smaller and lighter entity than a process or a program. Threads are created by processes and have their own flow of control and stack. A quick and simplistic way to differentiate a thread from a process is to consider a process as the running binary file and a thread as a subset of a process.

A **goroutine** is the minimum Go entity that can be executed concurrently. The use of the word "minimum" is very important here, as goroutines are not autonomous entities like UNIX processes – goroutines live in UNIX threads that live in UNIX processes. The main advantage of goroutines is that they are extremely lightweight and running thousands or hundreds of thousands of them on a single machine is not a problem.

The good thing is that goroutines are lighter than threads, which, in turn, are lighter than processes. In practice, this means that a process can have multiple threads as well as lots of goroutines, whereas a goroutine needs the environment of a process in order to exist. So, in order to create a goroutine, you will need to have a process with at least one thread – UNIX takes care of the process and thread management, while Go and the developer need to take care of the goroutines.

Now that you know the basics of processes, programs, threads, and goroutines, let us talk a little bit about the **Go scheduler**.

The Go scheduler

The UNIX kernel scheduler is responsible for the execution of the threads of a program. On the other hand, the Go runtime has its own scheduler, which is responsible for the execution of the goroutines using a technique known as **m:n scheduling**, where m goroutines are executed using n operating system threads using multiplexing. The Go scheduler is the Go component responsible for the way and the order in which the goroutines of a Go program get executed. This makes the Go scheduler a really important part of the Go programming language, as everything in a Go program is executed as a goroutine.

Be aware that as the Go scheduler only deals with the goroutines of a single program, its operation is much simpler, cheaper, and faster than the operation of the kernel scheduler.

Chapter 10, *Concurrency in Go – Advanced Topics*, will talk about the way the Go scheduler operates in much more detail.

Concurrency and parallelism

It is a very common misconception that **concurrency** is the same thing as **parallelism** – this is just not true! Parallelism is the simultaneous execution of multiple entities of some kind, whereas concurrency is a way of structuring your components so that they can be executed independently when possible.

It is only when you build software components concurrently that you can safely execute them in parallel, when and if your operating system and your hardware permit it. The Erlang programming language did this a long time ago – long before CPUs had multiple cores and computers had lots of RAM.

In a valid concurrent design, adding concurrent entities makes the whole system run faster because more things can be executed in parallel. So, the desired parallelism comes from a better concurrent expression and implementation of the problem. The developer is responsible for taking concurrency into account during the design phase of a system and will benefit from a potential parallel execution of the components of the system. So, the developer should not think about parallelism but about breaking things into independent components that solve the initial problem when combined.

Even if you cannot run your functions in parallel on a UNIX machine, a valid concurrent design will still improve the design and the maintainability of your programs. In other words, concurrency is better than parallelism!

Goroutines

You can define, create, and execute a new goroutine using the `go` keyword followed by a function name or the full definition of an **anonymous function**. The `go` keyword makes the function call return immediately, while the function starts running in the background as a goroutine and the rest of the program continues its execution.

However, as you will see in a moment, you cannot control or make any assumptions about the order in which your goroutines are going to be executed because this depends on the scheduler of the operating system, the Go scheduler, and the load of the operating system.

Creating a goroutine

In this subsection, you will learn two ways of creating goroutines. The first one is by using regular functions, while the second method is by using anonymous functions – these two ways are equivalent.

The name of the program covered in this section is `simple.go`, and it is presented in three parts.

The first part of `simple.go` is the following Go code:

```
package main

import (
    "fmt"
    "time"
)

func function() {
    for i := 0; i < 10; i++ {
        fmt.Print(i)
    }
}
```

Apart from the `import` block, the preceding code defines a function named `function()` that will be used in a short while.

The function name `function()` is nothing special – you can give it any valid function name you want.

The second part of `simple.go` is as follows:

```
func main() {  
    go function()  
}
```

The preceding code starts by executing `function()` as a goroutine. After that, the program continues its execution, while `function()` begins to run in the background.

The last code portion of `simple.go` is shown in the following Go code:

```
go func() {  
    for i := 10; i < 20; i++ {  
        fmt.Print(i, " ")  
    }  
}()  
  
time.Sleep(1 * time.Second)  
fmt.Println()  
}
```

With this code, you create a goroutine using an anonymous function. This method works best for relatively small functions. However, if you have lots of code, it is considered a better practice to create a regular function and execute it using the `go` keyword.

As you will see in the next section, you can create multiple goroutines any way you desire, including using a `for` loop.

Executing `simple.go` three times will generate the following type of output:

```
$ go run simple.go  
10 11 12 13 14 15 16 17 18 19 0123456789  
$ go run simple.go  
10 11 12 13 14 15 16 0117 2345678918 19  
$ go run simple.go  
10 11 12 012345678913 14 15 16 17 18 19
```

Although what you really want from your programs is to generate the same output for the same input, the output you get from `simple.go` is not always the same. The preceding output supports the fact that you cannot control the order in which your goroutines will be executed without taking extra care. This means writing extra code specifically for this to occur. In Chapter 10, *Concurrency in Go – Advanced Topics*, you will learn how to control the order in which your goroutines are executed, as well as how to print the results of one goroutine before printing the results of the following one.

Creating multiple goroutines

In this subsection, you will learn how to create a variable number of goroutines. The program reviewed in this section is called `create.go`. It is going to be presented in four parts, and it will allow you to create a dynamic number of goroutines. The number of goroutines will be given as a command-line argument to the program, which uses the `flag` package to process its command-line argument.

The first code part of `create.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "time"
)
```

The second code segment from `create.go` contains the following Go code:

```
func main() {
    n := flag.Int("n", 10, "Number of goroutines")
    flag.Parse()

    count := *n
    fmt.Printf("Going to create %d goroutines.\n", count)
```

The preceding code reads the value of the `n` command-line option, which determines the number of goroutines that are going to be created. If there is no `n` command-line option, the value of the `n` variable will be 10.

The third code portion of `create.go` is as follows:

```
    for i := 0; i < count; i++ {
        go func(x int) {
            fmt.Printf("%d ", x)
        }(i)
    }
```

A `for` loop is used to create the desired number of goroutines. Once again, you should remember that you cannot make any assumptions about the order in which they are going to be created and executed.

The last part of the Go code from `create.go` is the following:

```
    time.Sleep(time.Second)
```

```
    fmt.Println("\nExiting...")
}
```

The purpose of the `time.Sleep()` statement is to give the goroutines enough time to finish their jobs so that their output can be seen on the screen. In a real program, you will not need a `time.Sleep()` statement, as you will want to finish as soon as possible and, moreover, you will learn a better technique of making your program wait for the various goroutines to finish before the `main()` function returns.

Executing `create.go` multiple times will generate the following type of output:

```
$ go run create.go -n 100
Going to create 100 goroutines.
5 3 2 4 19 9 0 1 7 11 10 12 13 14 15 31 16 20 17 22 8 18 28 29 21 52 30 45
25 24 49 38 41 46 6 56 57 54 23 26 53 27 59 47 69 66 51 44 71 48 74 33 35
73 39 37 58 40 50 78 85 86 90 67 72 91 32 64 65 95 75 97 99 93 36 60 34 77
94 61 88 89 83 84 43 80 82 87 81 68 92 62 55 98 96 63 76 79 42 70
Exiting...
$ go run create.go -n 100
Going to create 100 goroutines.
2 5 3 16 6 7 8 9 1 22 10 12 13 17 11 18 15 14 19 20 31 23 26 21 29 24 30 25
37 32 36 38 35 33 45 41 43 42 40 39 34 44 48 46 47 56 53 50 0 49 55 59 58
28 54 27 60 4 57 51 52 64 61 65 72 62 63 67 69 66 74 73 71 75 89 70 76 84
85 68 79 80 93 97 83 82 99 78 88 91 92 77 81 95 94 98 87 90 96 86
Exiting...
```

Once again, you can see that the output is non-deterministic and messy in the sense that you will have to search the output to find what you are looking for. Additionally, if you do not use a suitable delay in the `time.Sleep()` call, you will not be able to see the output of the goroutines. `time.Second` might be fine for now, but this kind of code can cause nasty and unpredictable bugs further down the road.

In the next section, you will learn how to give your goroutines enough time to finish what they are doing before your program ends, without the need to call `time.Sleep()`.

Waiting for your goroutines to finish

This section will present a way to prevent the `main()` function from ending while it is waiting for its goroutines to finish using the `sync` package. The logic of the `syncGo.go` program will be based on `create.go`, which was presented in the previous section.

The first part of `syncGo.go` is as follows:

```
package main
```

```
import (  
    "flag"  
    "fmt"  
    "sync"  
)
```

As you can see, there is no need to import and use the `time` package, as we will use the functionality of the `sync` package and wait for as long as necessary for all the goroutines to end.



In Chapter 10, *Concurrency in Go – Advanced Topics*, you will study two techniques for timing out goroutines when they are taking longer than desired.

The second code segment of `syncGo.go` is shown in the following Go code:

```
func main() {  
    n := flag.Int("n", 20, "Number of goroutines")  
    flag.Parse()  
    count := *n  
    fmt.Printf("Going to create %d goroutines.\n", count)  
  
    var waitGroup sync.WaitGroup
```

In the preceding Go code, you define a `sync.WaitGroup` variable. If you look at the source code of the `sync` Go package, and more specifically at the `waitgroup.go` file that is located inside the `sync` directory, you will see that the `sync.WaitGroup` type is nothing more than a structure with three fields:

```
type WaitGroup struct {  
    noCopy  
    state1 [12]byte  
    sema   uint32  
}
```

The output of `syncGo.go` will reveal more information about the way `sync.WaitGroup` variables work. The number of goroutines that belong to a `sync.WaitGroup` group is defined by one or multiple calls to the `sync.Add()` function.

The third part of `syncGo.go` contains the following Go code:

```
fmt.Printf("%#v\n", waitGroup)  
for i := 0; i < count; i++ {  
    waitGroup.Add(1)  
    go func(x int) {
```

```

        defer waitGroup.Done()
        fmt.Printf("%d ", x)
    } (i)
}

```

Here, you create the desired number of goroutines using a `for` loop. (You could use multiple sequential Go statements instead.)

Each call to `sync.Add()` increases a counter in a `sync.WaitGroup` variable. Notice that it is really important to call `sync.Add(1)` before the `go` statement in order to prevent any **race conditions**. When each goroutine finishes its job, the `sync.Done()` function will be executed, which will decrease the same counter.

The last code portion of `syncGo.go` is as follows:

```

    fmt.Printf("%#v\n", waitGroup)
    waitGroup.Wait()
    fmt.Println("\nExiting...")
}

```

The `sync.Wait()` call blocks until the counter in the relevant `sync.WaitGroup` variable is zero, giving your goroutines time to finish.

Executing `syncGo.go` will create the following type of output:

```

$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x14, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 7 8 9 10 11 12 13 14 15 16 17 0 1 2 5 18 4 6 3
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
1 0 4 5 17 7 8 9 10 11 12 13 2 sync.WaitGroup{noCopy:sync.noCopy{},
state1:[12]uint8{0x0, 0x0, 0x0, 0x0, 0x17, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0},
sema:0x0}
29 15 6 27 24 25 16 22 14 23 18 26 3 19 20 28 21
Exiting...
$ go run syncGo.go -n 30
Going to create 30 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,

```

```

0x1e, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
29 1 7 8 2 9 10 11 12 4 13 15 0 6 5 22 25 23 16 28 26 20 19 24 21 14 3 17
18 27
Exiting...

```

The output of `syncGo.go` still varies from execution to execution, especially if you are dealing with a large number of goroutines. Most of the time, this is acceptable; however, there are times when this is not the desired behavior. Additionally, when the number of goroutines is 30, some of the goroutines have finished their job before the second `fmt.Printf("%#v\n", waitGroup)` statement. Finally, notice that one of the elements of the `state1` field in `sync.WaitGroup` is the one that holds the counter, which increases and decreases according to the `sync.Add()` and `sync.Done()` calls.

What if the number of `Add()` and `Done()` calls do not agree?

When the number of `sync.Add()` calls and `sync.Done()` calls are equal, everything will be fine in your programs. However, this section will tell you what will happen when these two numbers do not agree with each other.

If you have executed more `sync.Add()` calls than `sync.Done()` calls, in this case by adding a `waitGroup.Add(1)` statement before the first `fmt.Printf("%#v\n", waitGroup)` statement of the `syncGo.go` program, then the output of the `go run` command will be similar to the following:

```

$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x1, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x15, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 10 11 12 13 17 18 8 5 4 6 14 1 0 7 3 2 15 9 16 fatal error: all
goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc4200120bc)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:28 +0x2d7
exit status 2

```

The error message is pretty clear: `fatal error: all goroutines are asleep - deadlock!` This happened because you told your program to wait for `n+1` goroutines by calling the `sync.Add(1)` function `n+1` times while only `n` `sync.Done()` statements were executed by your `n` goroutines. As a result, the `sync.Wait()` call will wait indefinitely for one or more calls to `sync.Done()` without any luck, which is obviously a deadlock situation.

If you have made fewer `sync.Add()` calls than `sync.Done()` calls, which can be emulated by adding a `waitGroup.Done()` statement after the `for` loop of the `syncGo.go` program, then the `go run` output will be similar to the following:

```
$ go run syncGo.go
Going to create 20 goroutines.
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
sync.WaitGroup{noCopy:sync.noCopy{}, state1:[12]uint8{0x0, 0x0, 0x0, 0x0,
0x12, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0, 0x0}, sema:0x0}
19 6 1 2 9 7 8 15 13 0 14 16 17 3 11 4 5 12 18 10 panic: sync: negative
WaitGroup counter
goroutine 22 [running]:
sync.(*WaitGroup).Add(0xc4200120b0, 0xffffffffffffffff)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:75 +0x134
sync.(*WaitGroup).Done(0xc4200120b0)
    /usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:100 +0x34
main.main.func1(0xc4200120b0, 0x11)
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:25 +0xd8
created by main.main
    /Users/mtsouk/Desktop/masterGo/ch/ch9/code/syncGo.go:21 +0x206
exit status 2
```

Once again, the root of the problem is stated pretty clearly: `panic: sync: negative WaitGroup counter`.

Although the error messages are very descriptive in both cases and will help you to solve the real problem, you should be very careful with the number of `sync.Add()` and `sync.Done()` calls that you put into your programs. Additionally, notice that in the second error case (`panic: sync: negative WaitGroup counter`), the problem might not appear all of the time.

Channels

A channel is a communication mechanism that allows goroutines to exchange data, among other things.

However, there are some specific rules. Firstly, each channel allows the exchange of a particular data type, which is also called the **element type** of the channel, and secondly, for a channel to operate properly, you will need someone to receive what is sent via the channel. You should declare a new channel using the `chan` keyword, and you can close a channel using the `close()` function.

Finally, a very important detail: when you are using a channel as a function parameter, you can specify its direction; that is, whether it is going to be used for sending or receiving. In my opinion, if you know the purpose of a channel in advance, you should use this capability because it will make your programs more robust, as well as safer. You will not be able to send data accidentally to a channel from which you should only receive data, or receive data from a channel to which you should only be sending data. As a result, if you declare that a channel function parameter will be used for reading only and you try to write to it, you will get an error message that will most likely save you from nasty bugs. We will talk about this later on in this chapter.



Although you will learn many things about channels in this chapter, you will have to wait for [Chapter 10, Concurrency in Go – Advanced Topics](#), to fully understand the power and flexibility that channels offer to the Go developer.

Writing to a channel

The code in this subsection will teach you how to write to a channel. Writing the value `x` to channel `c` is as easy as writing `c <- x`. The arrow shows the direction of the value, and you will have no problem with this statement as long as both `x` and `c` have the same type. The example code in this section is saved in `writeCh.go`, and it will be presented in three parts.

The first code segment from `writeCh.go` is as follows:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println(x)
    c <- x
    close(c)
```



```
    fmt.Println(x)
}
```

The `chan` keyword is used for declaring that the `c` function parameter will be a channel, and it should be followed by the type of the channel (`int`). The `c <- x` statement allows you to write the value `x` to channel `c`, and the `close()` function closes the channel; that is, it makes writing to it impossible.

The second part of `writeCh.go` contains the following Go code:

```
func main() {
    c := make(chan int)
```

In the preceding code, you can find the definition of a channel variable, which is named `c`, and for the first time in this chapter you are using the `make()` function as well as the `chan` keyword. All channels have a type associated with them, which in this case is `int`.

The remaining code from `writeCh.go` is as follows:

```
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
}
```

Here, you execute the `writeToChannel()` function as a goroutine and call `time.Sleep()` in order to give enough time to the `writeToChannel()` function to execute.

Executing `writeCh.go` will create the following output:

```
$ go run writeCh.go
10
```

The strange thing here is that the `writeToChannel()` function prints the given value only once. The cause of this unexpected output is that the second `fmt.Println(x)` statement is never executed. The reason for this is pretty simple once you understand how channels work: the `c <- x` statement is blocking the execution of the rest of the `writeToChannel()` function because nobody is reading what was written to the `c` channel. Therefore, when the `time.Sleep(1 * time.Second)` statement finishes, the program terminates without waiting for `writeToChannel()`.

The next section will illustrate how to read data from a channel.

Reading from a channel

In this subsection, you will learn how to read from a channel. You can read a single value from a channel named `c` by executing `<-c`. In this case, the direction is from the channel to the outer world.

The name of the program that will be used to help you understand how to read from a channel is `readCh.go`, and it will be presented in three parts.

The first code segment from `readCh.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "time"
)

func writeToChannel(c chan int, x int) {
    fmt.Println("1", x)
    c <- x
    close(c)
    fmt.Println("2", x)
}
```

The implementation of the `writeToChannel()` function is the same as before.

The second part of `readCh.go` follows:

```
func main() {
    c := make(chan int)
    go writeToChannel(c, 10)
    time.Sleep(1 * time.Second)
    fmt.Println("Read:", <-c)
    time.Sleep(1 * time.Second)
}
```

In the preceding code, you read from channel `c` using the `<-c` notation. If you want to store that value to a variable named `k` instead of just printing it, you can use a `k := <-c` statement. The second `time.Sleep(1 * time.Second)` statement gives you the time to read from the channel.

The last code portion of `readCh.go` contains the following Go code:

```
_, ok := <-c
if ok {
    fmt.Println("Channel is open!")
} else {
```

```
        fmt.Println("Channel is closed!")
    }
}
```

In the preceding code, you can see a technique for determining whether a given channel is open or not. The current Go code works fine when the channel is closed; however, if the channel was open, the Go code presented here would have discarded the read value of the channel because of the use of the `_` character in the `_, ok := <-c` statement. Use a proper variable name instead of `_` if you also want to store the value found in the channel in case it is open.

Executing `readCh.go` will generate the following output:

```
$ go run readCh.go
1 10
Read: 10
2 10
Channel is closed!
$ go run readCh.go
1 10
2 10
Read: 10
Channel is closed!
```

Although the output is still not deterministic, both the `fmt.Println(x)` statements of the `writeToChannel()` function are executed because the channel is unblocked when you read from it.

Receiving from a closed channel

In this subsection, you will learn what happens when you try to read from a closed channel using the Go code found in `readClose.go`, which is going to be presented in two parts.

The first part of `readClose.go` is as follows:

```
package main

import (
    "fmt"
)

func main() {
    willClose := make(chan int, 10)

    willClose <- -1
```

```
willClose <- 0
willClose <- 2

<-willClose
<-willClose
<-willClose
```

In this part of the program, we create a new `int` channel named `willClose`, we write data to it, and we read all that data without doing anything with it.

The second part of `readClose.go` contains the following code:

```
close(willClose)
read := <-willClose
fmt.Println(read)
}
```

In this part, we close the `willClose` channel and we try to read from the `willClose` channel, which we emptied in the previous part.

Executing `readClose.go` will generate the following output:

```
$ go run readClose.go
0
```

This means that reading from a closed channel returns the zero value of its data type, which in this case is 0.

Channels as function parameters

Although neither `readCh.go` nor `writeCh.go` used this feature, Go allows you to specify the direction of a channel when used as a function parameter; that is, whether it will be used for reading or writing. These two types of channels are called unidirectional channels, whereas, by default, channels are bidirectional.

Examine the Go code of the following two functions:

```
func f1(c chan int, x int) {
    fmt.Println(x)
    c <- x
}

func f2(c chan<- int, x int) {
    fmt.Println(x)
    c <- x
}
```

Although both functions implement the same functionality, their definitions are slightly different. The difference is created by the `<-` symbol found on the right of the `chan` keyword in the definition of the `f2()` function. This denotes that the `c` channel can be used for writing only. If the code of a Go function attempts to read from a write-only channel (**send-only channel**) parameter, the Go compiler will generate the following kind of error message:

```
# command-line-arguments
a.go:19:11: invalid operation: range in (receive from send-only type chan<-
int)
```

Similarly, you can have the following function definitions:

```
func f1(out chan<- int64, in <-chan int64) {
    fmt.Println(x)
    c <- x
}

func f2(out chan int64, in chan int64) {
    fmt.Println(x)
    c <- x
}
```

The definition of `f2()` combines a read-only channel named `in` with a write-only channel named `out`. If you accidentally try to write and close a read-only channel (**receive-only channel**) parameter of a function, you will get the following kind of error message:

```
# command-line-arguments
a.go:13:7: invalid operation: out <- i (send to receive-only type <-chan
int)
a.go:15:7: invalid operation: close(out) (cannot close receive-only
channel)
```

Pipelines

A **pipeline** is a virtual method for connecting goroutines and channels so that the output of one goroutine becomes the input of another goroutine using channels to transfer your data.

One of the benefits that you get from using pipelines is that there is a constant data flow in your program, as no goroutine and channel have to wait for everything to be completed in order to start their execution. Additionally, you use fewer variables and therefore less memory space because you do not have to save everything as a variable. Finally, the use of pipelines simplifies the design of the program and improves its maintainability.

Pipelines are going to be illustrated using the code of `pipeline.go`. This program will be presented in six parts. The task performed by the `pipeline.go` program is to generate random numbers in a given range and stop when any number in the random sequence appears a second time. However, before terminating, the program will print the sum of all random numbers that appeared up to the point where the first random number appeared a second time. You will need three functions to connect the channels of the program. The logic of the program is found in these three functions, but the data flows in the channels of the pipeline.

This program will have two channels. The first channel (channel A) will be used to get the random numbers from the first function and send them to the second function. The second channel (channel B) will be used by the second function to send the acceptable random numbers to the third function. The third function will be responsible for getting the data from channel B, calculating it, and presenting the results.

The first code segment of `pipeline.go` contains the following Go code:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)

var CLOSEA = false

var DATA = make(map[int]bool)
```

As the `second()` function will need a way to tell the `first()` function to close the first channel, I will use a global variable named `CLOSEA` for that. The `CLOSEA` variable is only checked by the `first()` function, and it can only be altered by the `second()` function.

The second part of `pipeline.go` is shown in the following Go code:

```
func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func first(min, max int, out chan<- int) {
    for {
        if CLOSEA {
            close(out)
            return
        }
        out <- random(min, max)
    }
}
```

```
    }
    out <- random(min, max)
  }
}
```

The preceding code presents the implementation of two functions named `random()` and `first()`. You are already familiar with the `random()` function that generates random numbers in a given range. However, the `first()` function is really interesting as it keeps running using a `for` loop until a Boolean variable (`CLOSEA`) becomes `true`. In that case, it will close its `out` channel.

The third code segment of `pipeline.go` is as follows:

```
func second(out chan<- int, in <-chan int) {
    for x := range in {
        fmt.Print(x, " ")
        _, ok := DATA[x]
        if ok {
            CLOSEA = true
        } else {
            DATA[x] = true
            out <- x
        }
    }
    fmt.Println()
    close(out)
}
```

The `second()` function receives data from the `in` channel and keeps sending it to the `out` channel. However, as soon as the `second()` function finds a random number that already exists in the `DATA` map, it sets the `CLOSEA` global variable to `true` and stops sending any more numbers to the `out` channel. After that, it closes the `out` channel.



range loops over channels and will automatically exit when the channel is closed.

The fourth code portion of `pipeline.go` is shown in the following Go code:

```
func third(in <-chan int) {
    var sum int
    sum = 0
    for x2 := range in {
        sum = sum + x2
    }
}
```

```
    fmt.Printf("The sum of the random numbers is %d\n", sum)
}
```

The `third()` function keeps reading from the `in` function parameter channel. When that channel is closed by the `second()` function, the `for` loop will stop getting any more data and the function will display its output. At this point, it should become clear that the `second()` function controls many things.

The fifth code segment of `pipeline.go` is as follows:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    n1, _ := strconv.Atoi(os.Args[1])
    n2, _ := strconv.Atoi(os.Args[2])

    if n1 > n2 {
        fmt.Printf("%d should be smaller than %d\n", n1, n2)
        return
    }
}
```

The previous code is used for working with the command-line arguments of the program.

The last part of the `pipeline.go` program is as follows:

```
    rand.Seed(time.Now().UnixNano())
    A := make(chan int)
    B := make(chan int)

    go first(n1, n2, A)
    go second(B, A)
    third(B)
}
```

Here you define the required channels, and you execute two goroutines and one function. The `third()` function is what prevents `main()` from returning immediately, because it is not executed as a goroutine.

Executing `pipeline.go` will produce the following type of output:

```
$ go run pipeline.go 1 10
2 2
The sum of the random numbers is 2
$ go run pipeline.go 1 10
```



```

9 7 8 4 3 3
The sum of the random numbers is 31
$ go run pipeline.go 1 10
1 6 9 7 1
The sum of the random numbers is 23
$ go run pipeline.go 10 20
16 19 16
The sum of the random numbers is 35
$ go run pipeline.go 10 20
10 16 17 11 15 10
The sum of the random numbers is 69
$ go run pipeline.go 10 20
12 11 14 15 10 15
The sum of the random numbers is 62

```

The important point here is that although the `first()` function keeps generating random numbers at its own pace and the `second()` function will print all of them on your screen, the unwanted random numbers, which are the random numbers that have already appeared, will not be sent to the `third()` function and therefore will not be included in the final sum.

Race conditions

The code of `pipeline.go` is not perfect and contains a logical error, which in concurrent terminology is called a race condition. This can be revealed by executing the following command:

```

$ go run -race pipeline.go 1 10
2 2 =====
WARNING: DATA RACE
Write at 0x00000122bae8 by goroutine 7:
    main.second()
        /Users/mtsouk/ch09/pipeline.go:34 +0x15c
Previous read at 0x00000122bae8 by goroutine 6:
    main.first()
        /Users/mtsouk/ch09/pipeline.go:21 +0xa3
Goroutine 7 (running) created at:
    main.main()
        /Users/mtsouk/ch09/pipeline.go:72 +0x2a1
Goroutine 6 (running) created at:
    main.main()
        /Users/mtsouk/ch09/pipeline.go:71 +0x275
=====
2
The sum of the random numbers is 2.

```

```
Found 1 data race(s)
exit status 66
```

The problem here is that the goroutine that executes the `second()` function might change the value of the `CLOSEA` variable while the `first()` function reads the `CLOSEA` variable. As what will happen first and what will happen second is not deterministic, it is considered a race condition. In order to correct this race condition, we will need to use a signal channel and the `select` keyword.



You will learn more about race conditions, signal channels, and the `select` keyword in Chapter 10, *Concurrency in Go – Advanced Topics*.

The output of the `diff(1)` command will reveal the changes made to `pipeline.go` – the new version is called `plNoRace.go`:

```
$ diff pipeline.go plNoRace.go
14a15,16
> var signal chan struct{}
>
21c23,24
<         if CLOSEA {
---
>         select {
>         case <-signal:
23a27
>         case out <- random(min, max) :
25d28
<         out <- random(min, max)
31d33
<         fmt.Print(x, " ")
34c36
<         CLOSEA = true
---
>         signal <- struct{}{}
35a38
>         fmt.Print(x, " ")
61d63
<
66a69,70
>         signal = make(chan struct{})
>
```

The logical correctness of `plNoRace.go` can be verified by the output of the next command:

```
$ go run -race plNoRace.go 1 10
```

```
8 1 4 9 3
```

```
The sum of the random numbers is 25.
```

Comparing Go and Rust concurrency models

Rust is a very popular systems programming language that also supports concurrent programming. Briefly speaking, some characteristics of Rust and the Rust concurrency model are as follows:

- Rust threads are UNIX threads, which means that they are heavy but can do many things.
- Rust supports both **message-passing** and **shared-state** concurrency like Go does with channels, mutexes, and shared variables.
- Based on its strict type and ownership system, Rust provides a safe thread mutable state. The rules are enforced by the Rust compiler.
- There are Rust structures that allow you to share state.
- If a thread starts misbehaving, the system will not crash. This situation can be handled and controlled.
- The Rust programming language is under constant development, which might discourage some people from using it as they might need to make changes to their existing code all the time.

So, Rust has a flexible concurrency model that is even more flexible than the concurrency model of Go. However, the price you will have to pay for this flexibility is having to live with Rust and its idiosyncrasies.

Comparing Go and Erlang concurrency models

Erlang is a very popular concurrent functional programming language that was designed with high availability in mind. Briefly speaking, the main characteristics of Erlang and the Erlang concurrency model are as follows:

- Erlang is a mature and tested programming language – this also applies to its concurrency model.

- If you do not like the way Erlang code works, you can always try **Elixir**, which is based on Erlang and uses the Erlang VM, but its code is more pleasant.
- Erlang uses asynchronous communication and nothing else.
- Erlang uses error handling for developing robust concurrent systems.
- Erlang processes can crash but if that crashing is handled properly, the system can continue working without problems.
- Just like goroutines, Erlang processes are isolated and there is no shared state between them. The one and only way for Erlang processes to communicate with each other is through message passing.
- Erlang threads are lightweight, just like Go goroutines. This means that you will be able to create as many processes as you want.

In summary, both Erlang and Elixir are established choices for reliable and highly available systems, as long as you are willing to work with the Erlang concurrency approach.

Additional resources

Visit the following useful resources:

- Visit the documentation page of the `sync` package, which can be found at <https://golang.org/pkg/sync/>.
- Visit the Rust web site at <https://www.rust-lang.org/>.
- Visit the Erlang web site at <https://www.erlang.org/>.
- Visit the documentation page of the `sync` package once more. Pay close attention to the `sync.Mutex` and `sync.RWMutex` types that will appear in the next chapter.

Exercises

- Create a pipeline that reads text files, finds the number of occurrences of a given phrase in each text file, and calculate the total number of occurrences of the phrase in all files.
- Create a pipeline for calculating the sum of the squares of all of the natural numbers in a given range.
- Remove the `time.Sleep(1 * time.Second)` statement from the `simple.go` program and see what happens. Why is that?

- Modify the Go code of `pipeline.go` in order to create a pipeline with five functions and the appropriate number of channels.
- Modify the Go code of `pipeline.go` in order to find out what will happen when you forget to close the `out` channel of the `first()` function.

Summary

In this chapter, you studied many unique Go features including goroutines, channels, and pipelines. Additionally, you found out how to give your goroutines enough time to finish their jobs using the functionality offered by the `sync` package. Finally, you learned that channels can be used as parameters to Go functions. This allows developers to create pipelines where data can flow.

The next chapter will continue talking about Go concurrency by introducing the formidable `select` keyword. This keyword helps Go channels to perform many interesting jobs, and I think that you will be truly amazed by its power.

After that, you will see two techniques that allow you to time out one or more goroutines that are stalled for some reason. Afterward, you will learn about `nil` channels, signal channels, channel of channels, and buffered channels, as well as the `context` package.

You will also learn about **shared memory**, which is the traditional way of sharing information among the threads of the same UNIX process that also applies to goroutines. Nevertheless, shared memory is not that popular among Go programmers because Go offers better, safer, and faster ways for goroutines to exchange data.

10

Concurrency in Go – Advanced Topics

The previous chapter introduced goroutines, which are the most important feature of Go, channels, and pipelines. This chapter will continue from the point where the previous one left off in order to help you to learn more about goroutines, channels, and the `select` keyword, before discussing shared variables, as well as the `sync.Mutex` and `sync.RWMutex` types.

This chapter also includes code examples that demonstrate the use of **signal channels**, **buffered channels**, **nil channels**, and **channels of channels**. Additionally, early on in this chapter, you will learn two techniques for timing out a goroutine after a given amount of time, because nobody can guarantee that all goroutines will finish before a desired time.

The chapter will end by examining the `atomic` package, race conditions, the `context` standard Go package, and worker pools.

In this chapter, you will learn about the following topics:

- The `select` keyword
- How the Go scheduler works
- Two techniques that allow you to time out a goroutine that takes longer than expected to finish
- Signal channels
- Buffered channels
- Nil channels
- Monitor goroutines
- Shared memory and mutexes
- The `sync.Mutex` and `sync.RWMutex` types
- The `context` package and its advanced functionality

- The `atomic` package
- Worker pools
- Detecting race conditions

The Go scheduler revisited

A **scheduler** is responsible for distributing the amount of work that needs to be done over the available resources in an efficient way. In this section, we will examine the way that the Go scheduler operates in much greater depth than in the previous chapter. As you already know, Go works using the **m:n scheduler** (or **M:N scheduler**). It schedules goroutines, which are lighter than OS threads, using OS threads. First, though, let us review the necessary theory and define some useful terms.

Go uses the **fork-join concurrency** model. The fork part of the model states that a child branch can be created at any point of a program. Analogously, the join part of the Go concurrency model is where the child branch ends and joins with its parent. Among other things, both `sync.Wait()` statements and channels that collect the results of goroutines are join points, whereas each new goroutine creates a child branch.



The fork phase of the fork-join model and the `fork(2)` C system call are two totally different things.

The **fair scheduling strategy**, which is pretty straightforward and has a simple implementation, shares all load evenly among the available processors. At first, this might look like the perfect strategy because it does not have to take many things into consideration while keeping all processors equally occupied. However, it turns out that this is not exactly the case because most distributed tasks usually depend on other tasks. Therefore, some processors are underutilized, or equivalently, some processors are utilized more than others.

A goroutine in Go is a **task**, whereas everything after the calling statement of a goroutine is a **continuation**. In the **work-stealing strategy** used by the Go scheduler, a (logical) processor that is underutilized looks for additional work from other processors. When it finds such jobs, it steals them from the other processor or processors, hence the name. Additionally, the work-stealing algorithm of Go queues and steals continuations. A **stalling join**, as is suggested by its name, is a point where a thread of execution stalls at a join and starts looking for other work to do.

Although both task stealing and continuation stealing have stalling joins, continuations happen more often than tasks; therefore, the Go algorithm works with continuations rather than tasks.

The main disadvantage of continuation stealing is that it requires extra work from the compiler of the programming language. Fortunately, Go provides that extra help and therefore uses **continuation stealing** in its work-stealing algorithm.

One of the benefits of continuation stealing is that you get the same results when using just functions instead of goroutines or a single thread with multiple goroutines. This makes perfect sense, as only one thing is executed at any given time in both cases.

Now, let us return to the $m:n$ scheduling algorithm used in Go. Strictly speaking, at any time, you have m goroutines that are executed, and therefore scheduled to run, on n OS threads using, at most, `GOMAXPROCS` number of logical processors. You will learn about `GOMAXPROCS` shortly.

The Go scheduler works using three main kinds of entities: OS threads (**M**), which are related to the operating system in use, goroutines (**G**), and **logical processors (P)**. The number of processors that can be used by a Go program is specified by the value of the **GOMAXPROCS environment variable** – at any given time, there are at most `GOMAXPROCS` processors.

The following figure illustrates this point:

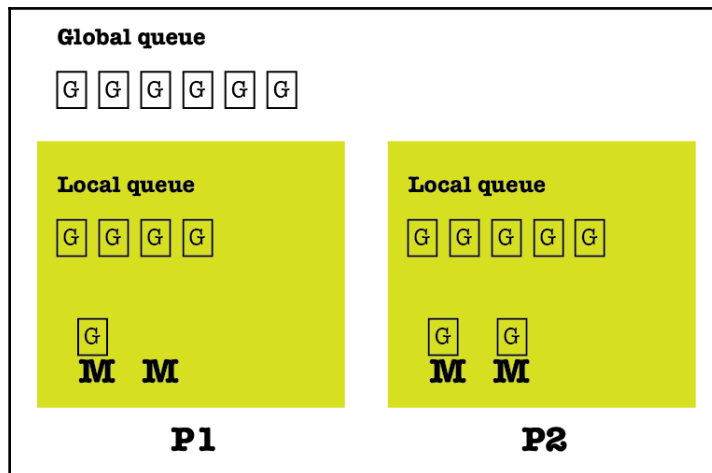


Figure 10.1: How the Go scheduler works

What the figure tells us is that there are two different kinds of queues: a global queue and a local queue attached to each logical processor. Goroutines from the global queue are assigned to the queue of a logical processor in order to be executed. As a result, the Go scheduler needs to check the global queue in order to avoid executing goroutines that are only located at the local queue of each logical processor. However, the global queue is not checked all of the time, which means that it does not have an advantage over the local queue.

Additionally, each logical processor can have multiple threads, and the stealing occurs between the local queues of the available logical processors. Finally, keep in mind that the Go scheduler is allowed to create more OS threads when needed. OS threads are pretty expensive, however, which means that dealing too much with OS threads might slow down your Go applications.

Remember that using more goroutines in a program is not a panacea for performance, as more goroutines, in addition to the various calls to `sync.Add()`, `sync.Wait()`, and `sync.Done()`, might slow down your program due to the extra housekeeping that needs to be done by the Go scheduler.



The Go scheduler, as well as most Go components, is always evolving, which means that the people who work on the Go scheduler constantly try to improve its performance by making small changes to the way it works. The core principles, however, remain the same.

You do not need to know all of this information in order to write Go code that uses goroutines. But knowing what occurs behind the scenes can definitely help you when strange things start happening or if you are curious about how the Go scheduler works. It will certainly make you a better developer!

The GOMAXPROCS environment variable

The `GOMAXPROCS` environment variable (and Go function) allows you to limit the number of operating system threads that can execute user-level Go code simultaneously. Starting with Go version 1.5, the default value of `GOMAXPROCS` should be the number of logical cores available in your UNIX machine.

If you decide to assign a value to `GOMAXPROCS` that is less than the number of the cores in your UNIX machine, you might affect the performance of your program. However, using a `GOMAXPROCS` value that is larger than the number of the available cores will not necessarily make your Go programs run faster.

You can programmatically discover the value of the `GOMAXPROCS` environment variable; the relevant code can be found in the following program, which is named `maxprocs.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func getGOMAXPROCS() int {
    return runtime.GOMAXPROCS(0)
}

func main() {
    fmt.Printf("GOMAXPROCS: %d\n", getGOMAXPROCS())
}
```

Executing `maxprocs.go` on a machine with an Intel i7 processor will produce the following output:

```
$ go run maxprocs.go
GOMAXPROCS: 8
```

You can modify the previous output by changing the value of the `GOMAXPROCS` environment variable prior to the execution of the program, however. The following commands are executed in the `bash(1)` UNIX shell:

```
$ go version
go version go1.12.3 darwin/amd64
$ export GOMAXPROCS=800; go run maxprocs.go
GOMAXPROCS: 800
$ export GOMAXPROCS=4; go run maxprocs.go
GOMAXPROCS: 4
```

The select keyword

As you will learn in a short while, the `select` keyword is pretty powerful and can do many things in a variety of situations. The `select` statement in Go looks like a `switch` statement but for channels. In practice, this means that `select` allows a goroutine to wait on multiple communication operations. Therefore, the main benefit that you receive from `select` is that it gives you the power to work with multiple channels using a single `select` block. As a consequence, you can have nonblocking operations on channels, provided that you have appropriate `select` blocks.



The biggest problem when using multiple channels and the `select` keyword is **deadlocks**. This means that you should be extra careful during the design and the development process in order to avoid such deadlocks.

The Go code of `select.go` will clarify the use of the `select` keyword. This program will be presented in five parts. The first part of `select.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

The second code portion from `select.go` is as follows:

```
func gen(min, max int, createNumber chan int, end chan bool) {
    for {
        select {
            case createNumber <- rand.Intn(max-min) + min:
            case <-end:
                close(end)
                return
            case <-time.After(4 * time.Second):
                fmt.Println("\ntime.After()!")
        }
    }
}
```

So, what is really happening in the code of this `select` block? This particular `select` statement has three cases. Notice that `select` statements do not require a default branch. You can consider the third branch of the `select` statement of the aforementioned code as a clever default branch. This happens because `time.After()` waits for the specified duration to elapse and then sends the current time on the returned channel – this will unblock the `select` statement in case all of the other channels are blocked for some reason.

A `select` statement is not evaluated sequentially, as all of its channels are examined simultaneously. If none of the channels in a `select` statement are ready, the `select` statement will block until one of the channels is ready. If multiple channels of a `select` statement are ready, then the Go runtime will make a random selection from the set of these ready channels.

The Go runtime tries to make this random selection between these ready channels as uniformly and as fairly as possible.

The third part of `select.go` is as follows:

```
func main() {
    rand.Seed(time.Now().Unix())
    createNumber := make(chan int)
    end := make(chan bool)

    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }
}
```

The fourth code portion of the `select.go` program contains the following Go code:

```
n, _ := strconv.Atoi(os.Args[1])
fmt.Printf("Going to create %d random numbers.\n", n)
go gen(0, 2*n, createNumber, end)

for i := 0; i < n; i++ {
    fmt.Printf("%d ", <-createNumber)
}
```



The reason for not examining the `error` value returned by `strconv.Atoi()` is to save some space. You should never do this in real applications.

The remaining Go code of the `select.go` program is as follows:

```
time.Sleep(5 * time.Second)
fmt.Println("Exiting...")
end <- true
}
```

The main purpose of the `time.Sleep(5 * time.Second)` statement is to give the `time.After()` function of `gen()` enough time to return and therefore activate the relevant branch of the `select` statement.

The last statement of the `main()` function is what terminates the program by activating the `case <-end` branch of the `select` statement in `gen()` and executing the related Go code.

Executing `select.go` will generate the following output:

```
$ go run select.go 10
```

```
Going to create 10 random numbers.  
13 17 8 14 19 9 2 0 19 5  
time.After()!  
Exiting...
```



The biggest advantage of `select` is that it can connect, orchestrate, and manage multiple channels. As channels connect goroutines, `select` connects channels that connect goroutines. Therefore, the `select` statement is one of the most important, if not the single most important, part of the Go concurrency model.

Timing out a goroutine

This section presents two very important techniques that will help you to time out goroutines. Put simply, these two techniques will save you from having to wait forever for a goroutine to finish its job, and they will give you full control over the amount of time that you want to wait for a goroutine to end. Both techniques use the capabilities of the handy `select` keyword combined with the `time.After()` function that you experienced in the previous section.

Timing out a goroutine – take 1

The source code of the first technique will be saved in `timeOut1.go`, and it will be presented in four parts.

The first part of `timeOut1.go` is shown in the following Go code:

```
package main  
  
import (  
    "fmt"  
    "time"  
)
```

The second code segment from `timeOut1.go` is as follows:

```
func main() {  
    c1 := make(chan string)  
    go func() {  
        time.Sleep(time.Second * 3)  
        c1 <- "c1 OK"  
    }()  
}
```

The `time.Sleep()` call is used to emulate the time it will normally take for the function to finish its job. In this case, the anonymous function that is executed as a goroutine will take about three seconds (`time.Second * 3`) before writing a message to the `c1` channel.

The third code segment from `timeOut1.go` contains the following Go code:

```
select {
case res := <-c1:
    fmt.Println(res)
case <-time.After(time.Second * 1):
    fmt.Println("timeout c1")
}
```

The purpose of the `time.After()` function call is to wait for the chosen time. In this case, you are not interested in the actual value returned by `time.After()` but in the fact that the `time.After()` function call has ended, which means that the available waiting time has passed. In this case, as the value passed to the `time.After()` function is smaller than the value used in the `time.Sleep()` call that was executed as the goroutine in the previous code segment, you will most likely get a timeout message.

The remaining code from `timeOut1.go` is as follows:

```
c2 := make(chan string)
go func() {
    time.Sleep(3 * time.Second)
    c2 <- "c2 OK"
}()

select {
case res := <-c2:
    fmt.Println(res)
case <-time.After(4 * time.Second):
    fmt.Println("timeout c2")
}
```

The preceding code both executes a goroutine that will take about three seconds to execute because of the `time.Sleep()` call and defines a timeout period of four seconds using `time.After(4 * time.Second)`. If the `time.After(4 * time.Second)` call returns after you get a value from the `c2` channel found in the first case of the `select` block, then there will not be any timeout; otherwise, you will get a timeout! However, in this case, the value of the `time.After()` call provides enough time for the `time.Sleep()` call to return, so you will most likely not get a timeout message here.

Executing `timeOut1.go` will generate the following type of output:

```
$ go run timeOut1.go
timeout c1
c2 OK
```

As expected, the first goroutine did not finish its job, whereas the second goroutine had enough time to finish.

Timing out a goroutine – take 2

The source code of the second technique will be saved in `timeOut2.go`, and it will be presented in five parts. This time, the timeout period is provided as a command-line argument to the program.

The first part of `timeOut2.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)
```

The second code segment of `timeOut2.go` is shown in the following Go code:

```
func timeout(w *sync.WaitGroup, t time.Duration) bool {
    temp := make(chan int)
    go func() {
        defer close(temp)
        time.Sleep(5 * time.Second)

        w.Wait()
    }()

    select {
    case <-temp:
        return false
    case <-time.After(t):
        return true
    }
}
```

In the preceding code, the time duration that will be used in the `time.After()` call is a parameter to the `timeout()` function, which means that it can vary. Once again, the `select` block implements the logic of the time out. Additionally, the `w.Wait()` call will make the `timeout()` function wait for a matching `sync.Done()` function indefinitely in order to end. When the `w.Wait()` call returns, the first branch of the `select` statement will be executed.

The third code portion from `timeOut2.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need a time duration!")
        return
    }

    var w sync.WaitGroup
    w.Add(1)

    t, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
        return
    }
```

The fourth part of the `timeOut2.go` program is as follows:

```
    duration := time.Duration(int32(t)) * time.Millisecond
    fmt.Printf("Timeout period is %s\n", duration)

    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
```

The `time.Duration()` function converts an integer value into a `time.Duration` variable that you can use afterward.

The remaining Go code from `timeOut2.go` is as follows:

```
    w.Done()
    if timeout(&w, duration) {
        fmt.Println("Timed out!")
    } else {
        fmt.Println("OK!")
    }
```



```
    }  
}
```

Once the `w.Done()` call is executed, the previous `timeout()` function will return. However, the second call to `timeout()` has no `sync.Done()` statement to wait for.

Executing `timeOut2.go` will generate the following type of output:

```
$ go run timeOut2.go 10000  
Timeout period is 10s  
Timed out!  
OK!
```

In this execution of `timeOut2.go`, the timeout period is longer than the `time.Sleep(5 * time.Second)` call of the anonymous goroutine. However, without the required call to `w.Done()`, the anonymous goroutine cannot return and therefore the `time.After(t)` call will end first, so the `timeout()` function of the first `if` statement will return `true`. In the second `if` statement, the anonymous function does not have to wait for anything, so the `timeout()` function will return `false` because `time.Sleep(5 * time.Second)` will finish before `time.After(t)`.

```
$ go run timeOut2.go 100  
Timeout period is 100ms  
Timed out!  
Timed out!
```

In the second program execution, however, the timeout period is too small, so both executions of `timeout()` will not have enough time to finish; therefore, both will be timed out. So, when defining a timeout period, make sure that you choose an appropriate value, or your results might not be what you expect.

Go channels revisited

Once the `select` keyword comes into play, Go channels can be used in several unique ways to do many more things than those you experienced in Chapter 9, *Concurrency in Go – Goroutines, Channels, and Pipelines*. This section will reveal the many uses of Go channels.

It helps to remember that the zero value of the channel type is `nil`, and that if you send a message to a closed channel, the program will panic. However, if you try to read from a closed channel, you will get the zero value of the type of that channel. So, after closing a channel, you can no longer write to it, but you can still read from it.

In order to be able to close a channel, the channel must not be receive-only. Additionally, a `nil` channel always blocks, which means that trying to read or write from a `nil` channel will block. This property of channels can be very useful when you want to disable a branch of a `select` statement by assigning the `nil` value to a channel variable.

Finally, if you try to close a `nil` channel, your program will panic. This is best illustrated in the `closeNilChannel.go` program, which is presented next:

```
package main

func main() {
    var c chan string
    close(c)
}
```

Executing `closeNilChannel.go` will generate the following output:

```
$ go run closeNilChannel.go
panic: close of nil channel
goroutine 1 [running]:
main.main()
    /Users/mtsouk/closeNilChannel.go:5 +0x2a
exit status 2
```

Signal channels

A signal channel is one that is used just for signaling. Put simply, you can use a signal channel when you want to inform another goroutine about something. Signal channels should not be used for the transferring of data.



You should not confuse signal channels with **UNIX signal handling**, which was discussed in [Chapter 8, *Telling a UNIX System What to Do*](#), because they are totally different things.

You will see a code example that uses signal channels in the section called *Specifying the order of execution for your goroutines* later in this chapter.

Buffered channels

The topic of this subsection is buffered channels. These are channels that allow the Go scheduler to put jobs in the queue quickly in order to be able to deal with more requests.

Moreover, you can use buffered channels as **semaphores** in order to limit the throughput of your application.

The technique presented here works as follows: all incoming requests are forwarded to a channel, which processes them one by one. When the channel is done processing a request, it sends a message to the original caller saying that it is ready to process a new one. So, the capacity of the buffer of the channel restricts the number of simultaneous requests that it can keep.

The technique will be presented with the help of the code found in `bufChannel.go`, which is broken down into four parts.

The first part of the code of `bufChannel.go` is as follows:

```
package main

import (
    "fmt"
)
```

The second code segment of `bufChannel.go` contains the following Go code:

```
func main() {
    numbers := make(chan int, 5)
    counter := 10
```

The definition presented of the `numbers` channel gives it a place to store up to five integers.

The third part of the code of `bufChannel.go` is shown in the following Go code:

```
    for i := 0; i < counter; i++ {
        select {
            case numbers <- i:
            default:
                fmt.Println("Not enough space for", i)
        }
    }
```

In the preceding code, we tried to put 10 integers in the `numbers` channel. However, as the `numbers` channel has room for only five integers, we will not be able to store all 10 integers in it.

The remaining Go code of `bufChannel.go` follows:

```
    for i := 0; i < counter+5; i++ {
        select {
```

```
        case num := <-numbers:
            fmt.Println(num)
        default:
            fmt.Println("Nothing more to be done!")
            break
    }
}
```

In the preceding Go code, we tried to read the contents of the `numbers` channel using a `for` loop and a `select` statement. As long as there is something to read from the `numbers` channel, the first branch of the `select` statement will get executed. As long as the `numbers` channel is empty, the `default` branch will be executed.

Executing `bufChannel.go` will create the following type of output:

```
$ go run bufChannel.go
Not enough space for 5
Not enough space for 6
Not enough space for 7
Not enough space for 8
Not enough space for 9
0
1
2
3
4
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
Nothing more to be done!
```

Nil channels

In this subsection, you will learn about nil channels. These are a special kind of channel because they always block. They are illustrated in `nilChannel.go`, which will be presented in four code segments.

The first part of `nilChannel.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "time"
)
```

The second code portion of `nilChannel.go` is shown in the following Go code:

```
func add(c chan int) {
    sum := 0
    t := time.NewTimer(time.Second)

    for {
        select {
            case input := <-c:
                sum = sum + input
            case <-t.C:
                c = nil
                fmt.Println(sum)
        }
    }
}
```

The `add()` function demonstrates how a nil channel is used. The `<-t.C` statement blocks the `c` channel of the `t` timer for the time that is specified in the `time.NewTimer()` call. Do not confuse channel `c`, which is the parameter of the function, with channel `t.C`, which belongs to timer `t`. When the time expires, the timer sends a value to the `t.C` channel. This will trigger the execution of the relevant branch of the `select` statement, which will assign the value `nil` to channel `c` and print the `sum` variable.

The third code segment of `nilChannel.go` is as follows:

```
func send(c chan int) {
    for {
        c <- rand.Intn(10)
    }
}
```

The purpose of the `send()` function is to generate random numbers and continue sending them to a channel for as long as the channel is open.

The remaining Go code of `nilChannel.go` is as follows:

```
func main() {  
    c := make(chan int)  
    go add(c)  
    go send(c)  
  
    time.Sleep(3 * time.Second)  
}
```

The `time.Sleep()` function is used to give enough time to the two goroutines to operate.

Executing `nilChannel.go` will generate the following output:

```
$ go run nilChannel.go  
13167523  
$ go run nilChannel.go  
12988362
```

Since the number of times that the first branch of the `select` statement in the `add()` function will be executed is not fixed, you get different results from executing `nilChannel.go`.

Channels of channels

A channel of channels is a special kind of channel variable that works with channels instead of other types of variables. Nevertheless, you still have to declare a data type for a channel of channels. You can define a channel of channels using the `chan` keyword twice in a row, as shown in the following statement:

```
c1 := make(chan chan int)
```



The other types of channels presented in this chapter are far more popular and useful than a channel of channels.

The use of channels of channels is illustrated using the code found in `chSquare.go`, which will be presented in four parts.

The first part of `chSquare.go` is as follows:

```
package main  
  
import (
```

```
    "fmt"
    "os"
    "strconv"
    "time"
)

var times int
```

The second code portion from `chSquare.go` is shown in the following Go code:

```
func f1(cc chan chan int, f chan bool) {
    c := make(chan int)
    cc <- c
    defer close(c)

    sum := 0
    select {
    case x := <-c:
        for i := 0; i <= x; i++ {
            sum = sum + i
        }
        c <- sum
    case <-f:
        return
    }
}
```

After declaring a regular `int` channel, you send that to the channel of channels variable. Then, you use a `select` statement in order to be able to read data from the regular `int` channel or exit your function using the `f` signal channel.

Once you read a single value from the `c` channel, you start a `for` loop that calculates the sum of all integers from 0 up to the integer value that you just read. Next, you send the calculated value to the `c int` channel and you are done.

The third part of `chSquare.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Need just one integer argument!")
        return
    }

    times, err := strconv.Atoi(arguments[1])
    if err != nil {
        fmt.Println(err)
    }
}
```

```
        return
    }

    cc := make(chan chan int)
```

The last statement in the preceding code is where you declare a channel of channels variable named `cc`, which is the star of the program because everything depends on that variable. The `cc` variable is passed to the `f1()` function, and it will be used in the `for` loop that is coming next.

The remaining Go code of `chSquare.go` is as follows:

```
    for i := 1; i < times+1; i++ {
        f := make(chan bool)
        go f1(cc, f)
        ch := <-cc
        ch <- i
        for sum := range ch {
            fmt.Print("Sum(", i, ")=", sum)
        }
        fmt.Println()
        time.Sleep(time.Second)
        close(f)
    }
}
```

The `f` channel is a signal channel used to end the goroutine when the real work is finished. The `ch := <-cc` statement allows you to get a regular channel from the channel of channels variable in order to be able to send an `int` value to it using `ch <- i`. After that, you start reading from it using a `for` loop. Although the `f1()` function is programmed to send a single value back, you can also read multiple values. Notice that each value of `i` is served by a different goroutine.

The type of a signal channel can be anything you want, including `bool`, which is used in the preceding code, and `struct{}`, which will be used in the signal channel in the next section. The main advantage of a `struct{}` signal channel is that no data can be sent to it, which can save you from bugs and misconceptions.

Executing `chSquare.go` will generate the following type of output:

```
$ go run chSquare.go 4
Sum(1)=1
Sum(2)=3
Sum(3)=6
Sum(4)=10
$ go run chSquare.go 7
```



```
Sum (1) =1
Sum (2) =3
Sum (3) =6
Sum (4) =10
Sum (5) =15
Sum (6) =21
Sum (7) =28
```

Specifying the order of execution for your goroutines

Although you should not make any assumptions about the order in which your goroutines will be executed, there are times when you need to be able to control this order. This subsection illustrates such a technique using signal channels.



You might ask, "Why choose to execute goroutines in a given order when simple functions could do the same job much more easily?" The answer is simple: goroutines are able to operate concurrently and wait for other goroutines to end, whereas functions executed in sequence cannot do that.

The name of the Go program for this topic is `defineOrder.go`, and it will be presented in five parts. The first part of `defineOrder.go` follows:

```
package main

import (
    "fmt"
    "time"
)

func A(a, b chan struct{}) {
    <-a
    fmt.Println("A()!")
    time.Sleep(time.Second)
    close(b)
}
```

The `A()` function is blocked by the channel stored in the `a` parameter. Once that channel is unblocked in the `main()` function, the `A()` function will start working. Finally, it will close channel `b`, which will unblock another function – in this case, function `B()`.

The second code portion of `defineOrder.go` is shown in the following Go code:

```
func B(a, b chan struct{}) {
```

```
    <-a
    fmt.Println("B()!")
    close(b)
}
```

The logic in `B()` is the same as in function `A()`. The function is blocked until channel `a` is closed. Then, it does its job and closes channel `b`. Notice that channels `a` and `b` refer to the names of the parameters of the function.

The third code segment of `defineOrder.go` follows:

```
func C(a chan struct{}) {
    <-a
    fmt.Println("C()!")
}
```

Function `C()` is blocked and waits for channel `a` to close in order to start working.

The fourth part of `defineOrder.go` contains the following code:

```
func main() {
    x := make(chan struct{})
    y := make(chan struct{})
    z := make(chan struct{})
```

These three channels will be the parameters to the three functions.

The last code segment of `defineOrder.go` contains the following Go code:

```
    go C(z)
    go A(x, y)
    go C(z)
    go B(y, z)
    go C(z)

    close(x)
    time.Sleep(3 * time.Second)
}
```

In this part, we execute the desired goroutines before closing the `x` channel and sleeping for three seconds.

Executing `defineOrder.go` will generate the desired output even though the `C()` function is called multiple times:

```
$ go run defineOrder.go
A()!
B()!
```

```
C()!  
C()!  
C()!
```

Calling the `C()` function multiple times as goroutines will work just fine because `C()` does not close any channels. However, if you call `A()` or `B()` more than once, you will most likely get an error message such as the following:

```
$ go run defineOrder.go  
A()!  
A()!  
B()!  
C()!  
C()!  
C()!  
panic: close of closed channel  
goroutine 7 [running]:  
main.A(0xc420072060, 0xc4200720c0)  
    /Users/mtsouk/Desktop/defineOrder.go:12 +0x9d  
created by main.main  
    /Users/mtsouk/Desktop/defineOrder.go:33 +0xfa  
exit status 2
```

As you can see from the output, function `A()` was called twice. However, as function `A()` closes a channel, one of its goroutines will find that channel already closed and generate a panic situation when it tries to close it again. You will get a similar panic situation if you call `B()` more than once.

How not to use goroutines

In this section, you are going to see a naive way to sort natural numbers using goroutines. The name of the program is `sillySort.go` and it will be presented in two parts. The first part of `sillySort.go` is the following:

```
package main  
  
import (  
    "fmt"  
    "os"  
    "strconv"  
    "sync"  
    "time"  
)  
  
func main() {
```

```

arguments := os.Args

if len(arguments) == 1 {
    fmt.Println(os.Args[0], "n1, n2, [n]")
    return
}

var wg sync.WaitGroup
for _, arg := range arguments[1:] {
    n, err := strconv.Atoi(arg)
    if err != nil || n < 0 {
        fmt.Print(". ")
        continue
    }

```

The second part of `sillySort.go` contains the following Go code:

```

    wg.Add(1)
    go func(n int) {
        defer wg.Done()
        time.Sleep(time.Duration(n) * time.Second)
        fmt.Print(n, " ")
    }(n)
}

wg.Wait()
fmt.Println()
}

```

The sorting takes place with the help of the `time.Sleep()` call – the bigger the natural number, the bigger the wait for the execution of the `fmt.Print()` statement!

Executing `sillySort.go` will generate the following kind of output:

```

$ go run sillySort.go a -1 1 2 3 5 0 100 20 60
. . 0 1 2 3 5 20 60 100
$ go run sillySort.go a -1 1 2 3 5 0 100 -1 a 20 hello 60
. . . . . 0 1 2 3 5 20 60 100
$ go run sillySort.go 0 0 10 2 30 3 4 30
0 0 2 3 4 10 30 30

```

Shared memory and shared variables

Shared memory and **shared variables** are the most common ways for UNIX threads to communicate with each other.

A **mutex** variable, which is an abbreviation of **mutual exclusion** variable, is mainly used for thread synchronization and for protecting shared data when multiple writes can occur at the same time. A mutex works like a buffered channel with a capacity of one, which allows at most one goroutine to access a shared variable at any given time. This means that there is no way for two or more goroutines to try to update that variable simultaneously.

A **critical section** of a concurrent program is the code that cannot be executed simultaneously by all processes, threads, or, in this case, goroutines. It is the code that needs to be protected by mutexes. Therefore, identifying the critical sections of your code will make the whole programming process so much simpler that you should pay particular attention to this task.



A critical section cannot be embedded into another critical section when both critical sections use the same `sync.Mutex` or `sync.RWMutex` variable. Put simply, avoid at almost any cost spreading mutexes across functions because that makes it really hard to see whether you are embedding or not.

The next two subsections will illustrate the use of the `sync.Mutex` and `sync.RWMutex` types.

The `sync.Mutex` type

The `sync.Mutex` type is the Go implementation of a mutex. Its definition, which can be found in the `mutex.go` file of the `sync` directory, is as follows:

```
// A Mutex is a mutual exclusion lock.
// The zero value for a Mutex is an unlocked mutex.
//
// A Mutex must not be copied after first use.
type Mutex struct {
    state int32
    sema  uint32
}
```



If you are interested in what code in the standard library is doing, remember that Go is completely open source and you can go and read it.

The definition of the `sync.Mutex` type is nothing extraordinary. All of the interesting work is done by the `sync.Lock()` and `sync.Unlock()` functions, which can lock and unlock a `sync.Mutex` mutex, respectively. Locking a mutex means that nobody else can lock it until it has been released using the `sync.Unlock()` function.

The `mutex.go` program, which is going to be presented in five parts, illustrates the use of the `sync.Mutex` type.

The first code segment of `mutex.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    m sync.Mutex
    v1 int
)
```

The second part of `mutex.go` is shown in the following Go code:

```
func change(i int) {
    m.Lock()
    time.Sleep(time.Second)
    v1 = v1 + 1
    if v1%10 == 0 {
        v1 = v1 - 10*i
    }
    m.Unlock()
}
```

The critical section of this function is the Go code between the `m.Lock()` and `m.Unlock()` statements.

The third part of `mutex.go` contains the following Go code:

```
func read() int {
    m.Lock()
    a := v1
    m.Unlock()
    return a
}
```

Similarly, the critical section of this function is defined by the `m.Lock()` and `m.Unlock()` statements.

The fourth code segment of `mutex.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give me an integer!")
        return
    }

    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    var waitGroup sync.WaitGroup
```

The last part of `mutex.go` is shown in the following Go code:

```
    fmt.Printf("%d ", read())
    for i := 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(i int) {
            defer waitGroup.Done()
            change(i)
            fmt.Printf("-> %d", read())
        }(i)
    }

    waitGroup.Wait()
    fmt.Printf("-> %d\n", read())
}
```

Executing `mutex.go` will generate the following type of output:

```
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -30-> -29-> -28-> -27-> -26->
-25-> -24-> -23-> -22-> -21-> -210-> -209-> -209
```

```
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -130-> -129-> -128-> -127-> -126->
-125-> -124-> -123-> -122-> -121-> -220-> -219-> -219
$ go run mutex.go 21
0 -> 1-> 2-> 3-> 4-> 5-> 6-> 7-> 8-> 9-> -100-> -99-> -98-> -97-> -96->
-95-> -94-> -93-> -92-> -91-> -260-> -259-> -259
```

If you remove the `m.Lock()` and `m.Unlock()` statements from the `change()` function, the program will generate output similar to the following:

```
$ go run mutex.go 21
0 -> 1-> 6-> 7-> 5-> -60-> -59-> 9-> 2-> -58-> 3-> -52-> 4-> -57-> 8->
-55-> -90-> -54-> -89-> -53-> -56-> -51-> -89
$ go run mutex.go 21
0 -> 1-> 7-> 8-> 9-> 5-> -99-> 4-> 2-> -97-> -96-> 3-> -98-> -95-> -100->
-93-> -94-> -92-> -91-> -230-> 6-> -229-> -229
$ go run mutex.go 21
0 -> 3-> 7-> 8-> 9-> -120-> -119-> -118-> -117-> 1-> -115-> -114-> -116->
4-> 6-> -112-> 2-> -111-> 5-> -260-> -113-> -259-> -259
```

The reason for such a change in the output is that all goroutines are simultaneously changing the shared variable, which is the main reason that the output appears randomly generated.

What happens if you forget to unlock a mutex?

In this section, you will see what happens if you forget to unlock a `sync.Mutex`. You will do this using the Go code of `forgetMutex.go`, which will be presented in two parts.

The first part of `forgetMutex.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "sync"
)

var m sync.Mutex

func function() {
    m.Lock()
    fmt.Println("Locked!")
}
```


All of the problems in this program are caused because the developer forgot to release the lock to `m sync.Mutex`. However, if your program is going to call `function()` only once, then everything will look just fine.

The second part of `forgetMutex.go` is as follows:

```
func main() {
    var w sync.WaitGroup

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    go func() {
        defer w.Done()
        function()
    }()
    w.Add(1)

    w.Wait()
}
```

There is nothing wrong with the `main()` function, which generates just two goroutines and waits for them to finish.

Executing `forgetMutex.go` will produce the following output:

```
$ go run forgetMutex.go
Locked!
fatal error: all goroutines are asleep - deadlock!
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001209c)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420012090)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
    /Users/mtsouk/forgetMutex.go:30 +0xb6
goroutine 5 [semacquire]:
sync.runtime_SemacquireMutex(0x115c6fc, 0x0)
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/sema.go:71 +0x3d
sync.(*Mutex).Lock(0x115c6f8)
    /usr/local/Cellar/go/1.12.3/libexec/src/sync/mutex.go:134 +0xee
main.function()
    /Users/mtsouk/forgetMutex.go:11 +0x2d
main.main.func1(0xc420012090)
```

```
/Users/mtsouk/forgetMutex.go:20 +0x48
created by main.main
/Users/mtsouk/forgetMutex.go:18 +0x58
exit status 2
```

So, forgetting to unlock a `sync.Mutex` mutex will create a panic situation even in the simplest kind of program. The same applies to the `sync.RWMutex` type of mutex, which you are going to work with in the next section.

The `sync.RWMutex` type

The `sync.RWMutex` type is another kind of mutex – actually, it is an improved version of `sync.Mutex`, which is defined in the `rwmutex.go` file of the `sync` directory as follows:

```
type RWMutex struct {
    w          Mutex // held if there are pending writers
    writerSem   uint32 // semaphore for writers to wait for completing
    readers
    readerSem   uint32 // semaphore for readers to wait for completing
    writers
    readerCount int32  // number of pending readers
    readerWait  int32  // number of departing readers
}
```

In other words, `sync.RWMutex` is based on `sync.Mutex` with the necessary additions and improvements.

Now, let us talk about how `sync.RWMutex` improves `sync.Mutex`. Although only one function is allowed to perform write operations using a `sync.RWMutex` mutex, you can have multiple readers owning a `sync.RWMutex` mutex. However, there is one thing that you should be aware of: until all of the readers of a `sync.RWMutex` mutex unlock that mutex, you cannot lock it for writing, which is the small price you have to pay for allowing multiple readers.

The functions that can help you to work with a `sync.RWMutex` mutex are `RLock()` and `RUnlock()`, which are used for locking and unlocking the mutex for reading purposes, respectively. The `Lock()` and `Unlock()` functions used in a `sync.Mutex` mutex should still be used when you want to lock and unlock a `sync.RWMutex` mutex for writing purposes. Hence, an `RLock()` function call that locks for reading purposes should be paired with an `RUnlock()` function call. Finally, it should be apparent that you should not make changes to any shared variables inside an `RLock()` and `RUnlock()` block of code.

The Go code found in `rwMutex.go` illustrates the use and usefulness of the `sync.RWMutex` type. The program will be presented in six parts, and it contains two slightly different versions of the same function. The first one uses a `sync.RWMutex` mutex for reading, and the second one uses a `sync.Mutex` mutex for reading. The performance difference between these two functions will help you to better understand the benefits of the `sync.RWMutex` mutex when used for reading purposes.

The first part of `rwMutex.go` contains the following Go code:

```
package main

import (
    "fmt"
    "os"
    "sync"
    "time"
)

var Password = secret{password: "myPassword"}

type secret struct {
    RWM    sync.RWMutex
    M      sync.Mutex
    password string
}
```

The `secret` structure holds a shared variable, a `sync.RWMutex` mutex, and a `sync.Mutex` mutex.

The second code portion of `rwMutex.go` is shown in the following code:

```
func Change(c *secret, pass string) {
    c.RWM.Lock()
    fmt.Println("LChange")
    time.Sleep(10 * time.Second)
    c.password = pass
    c.RWM.Unlock()
}
```

The `Change()` function modifies a shared variable, which means that you need to use an exclusive lock, which is the reason for using the `Lock()` and `Unlock()` functions. You cannot avoid using exclusive locks when changing things!

The third part of `rwMutex.go` is as follows:

```
func show(c *secret) string {
```

```
c.RWM.RLock()
fmt.Print("show")
time.Sleep(3 * time.Second)
defer c.RWM.RUnlock()
return c.password
}
```

The `show()` function uses the `RLock()` and `RUnlock()` functions because its critical section is used for reading a shared variable. So, although many goroutines can read the shared variable, no one can change it without using the `Lock()` and `Unlock()` functions. However, the `Lock()` function will be blocked for as long as there is someone reading that shared variable using the mutex.

The fourth code segment of `rwMutex.go` contains the following Go code:

```
func showWithLock(c *secret) string {
    c.RWM.Lock()
    fmt.Println("showWithLock")
    time.Sleep(3 * time.Second)
    defer c.RWM.Unlock()
    return c.password
}
```

The only difference between the code of the `showWithLock()` function and the code of the `show()` function is that the `showWithLock()` function uses an exclusive lock for reading, which means that only one `showWithLock()` function can read the `password` field of the `secret` structure.

The fifth part of `rwMutex.go` contains the following Go code:

```
func main() {
    var showFunction = func(c *secret) string { return "" }
    if len(os.Args) != 2 {
        fmt.Println("Using sync.RWMutex!")
        showFunction = show
    } else {
        fmt.Println("Using sync.Mutex!")
        showFunction = showWithLock
    }

    var waitGroup sync.WaitGroup

    fmt.Println("Pass:", showFunction(&Password))
}
```

The remaining code of `rwMutex.go` follows:

```
for i := 0; i < 15; i++ {
```

```
    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        fmt.Println("Go Pass:", showFunction(&Password))
    }()

    go func() {
        waitGroup.Add(1)
        defer waitGroup.Done()
        Change(&Password, "123456")
    }()

    waitGroup.Wait()
    fmt.Println("Pass:", showFunction(&Password))
}
```

Executing `rwMutex.go` twice and using the `time(1)` command-line utility to benchmark the two versions of the program will generate the following type of output:

```
$ time go run rwMutex.go 10 >/dev/null
real    0m51.206s
user    0m0.130s
sys     0m0.074s
$ time go run rwMutex.go >/dev/null
real    0m22.191s
user    0m0.135s
sys     0m0.071s
```

Note that `> /dev/null` at the end of the preceding commands is for omitting the output of the two commands. Hence, the version that uses the `sync.RWMutex` mutex is much faster than the version that uses `sync.Mutex`.

The atomic package

An **atomic operation** is an operation that is completed in a single step relative to other threads or, in this case, to other goroutines. This means that an atomic operation cannot be interrupted in the middle of it.

The Go standard library offers the `atomic` package, which, in some cases, can help you to avoid using a mutex. However, mutexes are more versatile than atomic operations. Using the `atomic` package, you can have atomic counters accessed by multiple goroutines without synchronization issues and race conditions.

Notice that, if you have an atomic variable, all reading and writing must be done using the atomic functions provided by the `atomic` package. The use of the `atomic` package will be illustrated with the `atom.go` program, which will be presented in three parts.

The first part of `atom.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "sync"
    "sync/atomic"
)

type atomCounter struct {
    val int64
}

func (c *atomCounter) Value() int64 {
    return atomic.LoadInt64(&c.val)
}
```

The second part of `atom.go` contains the following code:

```
func main() {
    minusX := flag.Int("x", 100, "Goroutines")
    minusY := flag.Int("y", 200, "Value")
    flag.Parse()
    X := *minusX
    Y := *minusY

    var waitGroup sync.WaitGroup
    counter := atomCounter{}
```

The last part of `atom.go` is as follows:

```
    for i := 0; i < X; i++ {
        waitGroup.Add(1)
        go func(no int) {
            defer waitGroup.Done()
            for i := 0; i < Y; i++ {
                atomic.AddInt64(&counter.val, 1)
            }
        }(i)
    }

    waitGroup.Wait()
```

```
    fmt.Println(counter.Value())  
}
```

The desired variable changes using `atomic.AddInt64()`.

Executing `atom.go` will generate the following output:

```
$ go run atom.go  
20000  
$ go run atom.go -x 4000 -y 10  
40000
```

The output of `atom.go` proves that the counter used in the program is safe. It would be a very interesting exercise to modify the `atom.go` program in order to modify the counter variable using regular arithmetic (`counter.val++`) instead of `atomic.AddInt64()`. In that case, the output of the program would have been similar to the following:

```
$ go run atom.go -x 4000 -y 10  
37613  
$ go run atom.go  
15247
```

The previous output shows that the use of `counter.val++` makes the program thread unsafe.

In Chapter 12, *The Foundations of Network Programming in Go*, you are going to see a similar example using an HTTP server written in Go.

Sharing memory using goroutines

The last subsection of this topic illustrates how you can share data using a dedicated goroutine. Although shared memory is the traditional way that threads communicate with each other, Go comes with built-in synchronization features that allow a single goroutine to own a shared piece of data. This means that other goroutines must send messages to this single goroutine that owns the shared data, which prevents the corruption of the data. Such a goroutine is called a **monitor goroutine**. In Go terminology, this is *sharing by communicating instead of communicating by sharing*.

The technique will be illustrated using the `monitor.go` source file, which will be presented in five parts. The `monitor.go` program generates random numbers using a monitor goroutine.

The first part of `monitor.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
    "sync"
    "time"
)

var readValue = make(chan int)
var writeValue = make(chan int)
```

The `readValue` channel is used for reading random numbers, whereas the `writeValue` channel is used for getting new random numbers.

The second code portion of `monitor.go` is shown in the following code:

```
func set(newValue int) {
    writeValue <- newValue
}

func read() int {
    return <-readValue
}
```

The purpose of the `set()` function is to set the value of the shared variable, whereas the purpose of the `read()` function is to read the value of the saved variable.

The third code segment of the `monitor.go` program is as follows:

```
func monitor() {
    var value int
    for {
        select {
        case newValue := <-writeValue:
            value = newValue
            fmt.Printf("%d ", value)
        case readValue <- value:
        }
    }
}
```


All of the logic of the program can be found in the implementation of the `monitor()` function. More specifically, the `select` statement orchestrates the operation of the entire program.

When you have a read request, the `read()` function attempts to read from the `readValue` channel, which is controlled by the `monitor()` function. This returns the current value that is kept in the `value` variable. On the other hand, when you want to change the stored value, you call `set()`. This writes to the `writeValue` channel, which is also handled by the `select` statement. As a result, no one can deal with the `value` shared variable without using the `monitor()` function.

The fourth code segment of `monitor.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Please give an integer!")
        return
    }
    n, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("Going to create %d random numbers.\n", n)
    rand.Seed(time.Now().Unix())
    go monitor()
```

The last part of `monitor.go` contains the following Go code:

```
var w sync.WaitGroup

for r := 0; r < n; r++ {
    w.Add(1)
    go func() {
        defer w.Done()
        set(rand.Intn(10 * n))
    }()
}
w.Wait()
fmt.Printf("\nLast value: %d\n", read())
}
```

Executing `monitor.go` generates the following output:

```
$ go run monitor.go 20
```

```
Going to create 20 random numbers.  
89 88 166 42 149 89 20 84 44 178 184 28 52 121 62 91 31 117 140 106  
Last value: 106  
$ go run monitor.go 10  
Going to create 10 random numbers.  
30 16 66 70 65 45 31 57 62 26  
Last value: 26
```



Personally, I prefer to use a monitor goroutine instead of traditional shared memory techniques because the implementation that uses the monitor goroutine is safer, closer to the Go philosophy, and much cleaner.

Revisiting the go statement

Although goroutines are fast and you can execute thousands of goroutines on your machine, this comes at a price. In this section, we are going to talk about the go statement, its behavior, and what happens when you start new goroutines in your Go programs.

Notice that closed variables in goroutines are evaluated when the goroutine actually runs and when the go statement is executed in order to create a new goroutine. This means that closed variables are going to be replaced by their values when the Go scheduler decides to execute the relevant code. This is illustrated in the following Go code, which is saved as `cloGo.go`:

```
package main  
  
import (  
    "fmt"  
    "time"  
)  
  
func main() {  
    for i := 0; i <= 20; i++ {  
        go func() {  
            fmt.Print(i, " ")  
        }()  
    }  
    time.Sleep(time.Second)  
    fmt.Println()  
}
```

It would be a good idea to stop reading for a moment and try to guess what the output of that code will be.

Executing `cloGo.go` multiple times will reveal the problem we were talking about:

```
$ go run cloGo.go
9 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21 21
$ go run cloGo.go
4 21 21 21 21 21 21 21 21 6 21 21 21 21 21 21 21 21
$ go run cloGo.go
6 21 6 6 21 21 21 21 21 21 21 21 21 21 21 21 21 6 21
```

The program mostly prints the number 21, which is the last value of the variable of the `for` loop and not the other numbers. As `i` is a **closed variable**, it is evaluated at the time of execution. As the goroutines begin but wait for the Go scheduler to allow them to get executed, the `for` loop ends, so the value of `i` that is being used is 21. Lastly, the same issue also applies to Go channels, so be careful.

There is a pretty funny and unexpected way to solve this problem and it involves some idiomatic Go. The solution can be seen in `cloGoCorrect.go`, which contains the following code:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i <= 20; i++ {
        i := i
        go func() {
            fmt.Print(i, " ")
        }()
    }
    time.Sleep(time.Second)
    fmt.Println()
}
```

The valid yet bizarre `i := i` statement creates a new instance of the variable for the goroutine, which makes the output of `cloGoCorrect.go` similar to the following:

```
$ go run cloGoCorrect.go
1 5 4 3 6 0 13 7 8 9 10 11 12 17 14 15 16 19 18 20 2
$ go run cloGoCorrect.go
5 2 20 13 6 7 1 9 10 11 0 3 17 14 15 16 4 19 18 8 12
```

You are now going to see another strange case that uses the `go` statement. Look at the following Go code, which is saved as `endlessComp.go`:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    var i byte
    go func() {
        for i = 0; i <= 255; i++ {
        }
    }()
    fmt.Println("Leaving goroutine!")
    runtime.Gosched()
    runtime.GC()

    fmt.Println("Good bye!")
}
```

Executing `endlessComp.go` will surprise you as the program never ends because it blocks indefinitely and therefore has to be stopped manually:

```
$ go run endlessComp.go
Leaving goroutine!
^Csignal: interrupt
```

As you might have guessed, the root of the problem is related to the Go garbage collector and the way it works. The call to the `runtime.Gosched()` function asks the scheduler to execute another goroutine and then we invoke the Go garbage collector, which is trying to do its job.

First, the garbage collector needs all goroutines to go to sleep before doing its job. The problem is that the `for` loop will never end because the type of the `for` loop variable is `byte`. This means that the `for` loop prevents the system from doing anything else because the goroutine with that `for` loop will never go to sleep. This unfortunate event will happen even if your machine has multiple cores.

Notice that if the `for` loop was not empty, then the program would have been executed and ended just fine because the garbage collector would have had a place to stop.

Lastly, keep in mind that goroutines need to be explicitly signaled in order to end. You can easily end a goroutine using a simple `return` statement.

Catching race conditions

A **data race condition** is a situation where two or more running elements, such as threads and goroutines, try to take control of or modify a shared resource or a variable of a program. Strictly speaking, a data race occurs when two or more instructions access the same memory address, where at least one of them performs a write operation. If all operations are read operations, then there is no race condition.

Using the `-race` flag when running or building a Go source file will turn on the Go **race detector**, which will make the compiler create a modified version of a typical executable file. This modified version can record all access to shared variables as well as all synchronization events that take place, including calls to `sync.Mutex` and `sync.WaitGroup`. After analyzing the relevant events, the race detector prints a report that can help you to identify potential problems so that you can correct them.

Look at the following Go code, which is saved as `raceC.go`. This program is presented in three parts.

The first part of `raceC.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
)

func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The second part of `raceC.go` contains the following Go code:

```
var waitGroup sync.WaitGroup
var i int
```

```

k := make(map[int]int)
k[1] = 12

for i = 0; i < numGR; i++ {
    waitGroup.Add(1)
    go func() {
        defer waitGroup.Done()
        k[i] = i
    }()
}

```

The remaining Go code of `raceC.go` is as follows:

```

k[2] = 10
waitGroup.Wait()
fmt.Printf("k = %#v\n", k)
}

```

As if it was not enough that many goroutines accessed the `k` map at the same time, I added another statement that accesses the `k` map before calling the `sync.Wait()` function.

If you execute `raceC.go`, you will get the following type of output without any warning or error messages:

```

$ go run raceC.go 10
k = map[int]int{7:10, 2:10, 10:10, 1:12}
$ go run raceC.go 10
k = map[int]int{2:10, 10:10, 1:12, 8:8, 9:9}
$ go run raceC.go 10
k = map[int]int{10:10, 1:12, 6:7, 7:7, 2:10}

```

If you execute `raceC.go` only once, then everything will look normal despite the fact that you do not get what you would expect when printing the contents of the `k` map. However, executing `raceC.go` multiple times tells us that there is something wrong here, mainly because each execution generates a different output.

There are many more things that we can get from `raceC.go` and its unexpected output – if we decide to use the Go race detector to analyze it:

```

$ go run -race raceC.go 10
=====
WARNING: DATA RACE
Read at 0x00c00001a0a8 by goroutine 6:
    main.main.func1()
        /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x66
Previous write at 0x00c00001a0a8 by main goroutine:
    main.main()

```

```

/Users/mtsouk/Desktop/mGo2nd/raceC.go:28 +0x23f
Goroutine 6 (running) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====
=====
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 7:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Previous write at 0x00c0000bc000 by goroutine 6:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Goroutine 7 (running) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
Goroutine 6 (finished) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====
=====
WARNING: DATA RACE
Write at 0x00c0000bc000 by goroutine 8:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Previous write at 0x00c0000bc000 by goroutine 6:
  runtime.mapassign_fast64()
    /usr/local/Cellar/go/1.12.3/libexec/src/runtime/map_fast64.go:92 +0x0
  main.main.func1()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:32 +0x8d
Goroutine 8 (running) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
Goroutine 6 (finished) created at:
  main.main()
    /Users/mtsouk/Desktop/mGo2nd/raceC.go:30 +0x215
=====
k = map[int]int{1:1, 2:10, 3:3, 4:4, 5:5, 6:6, 7:7, 8:8, 9:9, 10:10}
Found 3 data race(s)
exit status 66

```

So, the race detector found three data races. Each one begins with the `WARNING: DATA RACE` message in its output.

The first data race happens inside `main.main.func1()` at line 32, which is called in line 28 by the `for` loop, which is called by a goroutine created at line 30. The problem here is signified by the `Previous write` message. After examining the related code, it is easy to see that the actual problem is that the anonymous function takes no parameters, which means that the value of `i` that is used in the `for` loop cannot be deterministically discerned, as it keeps changing due to the `for` loop, which is a write operation.

The message of the second data race is `Write at 0x00c0000bc000 by goroutine 7`. If you read the relevant output, you will see that the data race is related to a write operation, and it happens on a Go map at line 32 by at least two goroutines that started at line 30. As the two goroutines have the same name (`main.main.func1()`), this is an indication that we are talking about the same goroutine. Two goroutines trying to modify the same variable is a data race condition. The third data race is similar to the second one.



The `main.main.func1()` notation is used by Go in order to name an anonymous function internally. If you have different anonymous functions, their names will be different as well.

You might ask, "What can I do now in order to correct the problems coming from the two data races?"

Well, you can rewrite the `main()` function of `raceC.go`, as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) != 2 {
        fmt.Println("Give me a natural number!")
        return
    }
    numGR, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    var waitGroup sync.WaitGroup
    var i int

    k := make(map[int]int)
    k[1] = 12
```



```

    for i = 0; i < numGR; i++ {
        waitGroup.Add(1)
        go func(j int) {
            defer waitGroup.Done()
            aMutex.Lock()
            k[j] = j
            aMutex.Unlock()
        }(i)
    }

    waitGroup.Wait()
    k[2] = 10
    fmt.Printf("k = %#v\n", k)
}

```

The `aMutex` variable is a global `sync.Mutex` variable defined outside the `main()` function that is accessible from everywhere in the program. Although this is not required, having such a global variable can save you from having to pass it to your functions all of the time.

Saving the new version of `raceC.go` as `noRaceC.go` and executing it will generate the expected output:

```

$ go run noRaceC.go 10
k = map[int]int{1:1, 0:0, 5:5, 3:3, 6:6, 9:9, 2:10, 4:4, 7:7, 8:8}

```

Processing `noRaceC.go` with the Go race detector will generate the following output:

```

$ go run -race noRaceC.go 10
k = map[int]int{5:5, 7:7, 9:9, 1:1, 0:0, 4:4, 6:6, 8:8, 2:10, 3:3}

```

Note that you need to use a locking mechanism while accessing the `k` map. If you do not use such a mechanism and just change the implementation of the anonymous function that is executed as a goroutine, you will get the following output from `go run noRaceC.go`:

```

$ go run noRaceC.go 10
fatal error: concurrent map writes
goroutine 10 [running]:
runtime.throw(0x10ca0bd, 0x15)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/panic.go:605 +0x95
fp=0xc420024738 sp=0xc420024718 pc=0x10276b5
runtime.mapassign_fast64(0x10ae680, 0xc420074180, 0x5, 0x0)
    /usr/local/Cellar/go/1.9.3/libexec/src/runtime/hashtable_fast.go:607
+0x3d2 fp=0xc420024798 sp=0xc420024738 pc=0x100b582
main.main.func1(0xc420010090, 0xc420074180, 0x5)
    /Users/mtsouk/ch10/code/noRaceC.go:35 +0x6b fp=0xc4200247c8
sp=0xc420024798 pc=0x1096f5b
runtime.goexit()

```

```

/usr/local/Cellar/go/1.9.3/libexec/src/runtime/asm_amd64.s:2337 +0x1
fp=0xc4200247d0 sp=0xc4200247c8 pc=0x1050c21
created by main.main
/Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
goroutine 1 [semacquire]:
sync.runtime_Semacquire(0xc42001009c)
/usr/local/Cellar/go/1.9.3/libexec/src/runtime/sema.go:56 +0x39
sync.(*WaitGroup).Wait(0xc420010090)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:131 +0x72
main.main()
/Users/mtsouk/ch10/code/noRaceC.go:40 +0x17a
goroutine 12 [runnable]:
sync.(*WaitGroup).Done(0xc420010090)
/usr/local/Cellar/go/1.9.3/libexec/src/sync/waitgroup.go:99 +0x43
main.main.func1(0xc420010090, 0xc420074180, 0x7)
/Users/mtsouk/ch10/code/noRaceC.go:37 +0x79
created by main.main
/Users/mtsouk/ch10/code/noRaceC.go:32 +0x15a
exit status 2

```

The root of the problem can be seen clearly: `concurrent map writes`.

The context package

The main purpose of the `context` package is to define the `Context` type and support **cancellation**. Yes, you heard that right; there are times when, for some reason, you want to abandon what you are doing. However, it would be very helpful to be able to include some extra information about your cancellation decisions. The `context` package allows you to do exactly that.

If you take a look at the source code of the `context` package, you will realize that its implementation is pretty simple – even the implementation of the `Context` type is pretty simple, yet the `context` package is very important.



The `context` package existed for a while as an external Go package; it first appeared as a standard Go package in Go version 1.7. So, if you have an older Go version, you will not be able to follow this section without first downloading the `context` package or installing a newer Go version.

The `Context` type is an interface with four methods named `Deadline()`, `Done()`, `Err()`, and `Value()`. The good news is that you do not need to implement all of these functions of the `Context` interface – you just need to modify a `Context` variable using functions such as `context.WithCancel()`, `context.WithDeadline()`, and `context.WithTimeout()`.

All three of these functions return a derived `Context` (the child) and a `CancelFunc` function. Calling the `CancelFunc` function removes the parent's reference to the child and stops any associated timers. This means that the Go garbage collector is free to garbage collect the child goroutines that no longer have associated parent goroutines. For garbage collection to work correctly, the parent goroutine needs to keep a reference to each child goroutine. If a child goroutine ends without the parent knowing about it, then a memory leak occurs until the parent is canceled as well.

The following is a simple use of the `context` package using the Go code of the `simpleContext.go` file, which will be presented in six parts.

The first code segment of `simpleContext.go` contains the following code:

```
package main

import (
    "context"
    "fmt"
    "os"
    "strconv"
    "time"
)
```

The second part of `simpleContext.go` is as follows:

```
func f1(t int) {
    c1 := context.Background()
    c1, cancel := context.WithCancel(c1)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()
}
```

The `f1()` function requires just one parameter, which is the time delay, because everything else is defined inside the function. Notice that the type of the `cancel` variable is `context.CancelFunc`.

You need to call the `context.Background()` function in order to initialize an empty `Context`. The `context.WithCancel()` function uses an existing `Context` and creates a child of it with cancellation. The `context.WithCancel()` function also creates a `Done` channel that can be closed, either when the `cancel()` function is called, as shown in the preceding code, or when the `Done` channel of the parent context is closed.

The third code portion of `simpleContext.go` contains the rest of the code of the `f1()` function:

```
select {
case <-c1.Done():
    fmt.Println("f1():", c1.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f1():", r)
}
return
}
```

Here, you see the use of the `Done()` function of a `Context` variable. When this function is called, you have a cancellation. The return value of `Context.Done()` is a channel because, otherwise, you would have not been able to use it in a `select` statement.

The fourth part of `simpleContext.go` contains the following Go code:

```
func f2(t int) {
    c2 := context.Background()
    c2, cancel := context.WithTimeout(c2, time.Duration(t)*time.Second)
    defer cancel()

    go func() {
        time.Sleep(4 * time.Second)
        cancel()
    }()

    select {
    case <-c2.Done():
        fmt.Println("f2():", c2.Err())
        return
    case r := <-time.After(time.Duration(t) * time.Second):
        fmt.Println("f2():", r)
    }
    return
}
```

This part showcases the use of the `context.WithTimeout()` function, which requires two parameters: a `Context` parameter and a `time.Duration` parameter. When the timeout period expires, the `cancel()` function is automatically called.

The fifth part of `simpleContext.go` is as follows:

```
func f3(t int) {
    c3 := context.Background()
```

```
deadline := time.Now().Add(time.Duration(2*t) * time.Second)
c3, cancel := context.WithDeadline(c3, deadline)
defer cancel()

go func() {
    time.Sleep(4 * time.Second)
    cancel()
}()

select {
case <-c3.Done():
    fmt.Println("f3():", c3.Err())
    return
case r := <-time.After(time.Duration(t) * time.Second):
    fmt.Println("f3():", r)
}
return
}
```

The preceding Go code illustrates the use of the `context.WithDeadline()` function, which requires two parameters: a `Context` variable and a time in the future that signifies the deadline of the operation. When the deadline passes, the `cancel()` function is automatically called.

The remaining Go code of `simpleContext.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Println("Need a delay!")
        return
    }

    delay, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Delay:", delay)

    f1(delay)
    f2(delay)
    f3(delay)
}
```

The purpose of the `main()` function is to initialize things.

Executing `simpleContext.go` will generate the following type of output:

```
$ go run simpleContext.go 4
Delay: 4
f1(): context canceled
f2(): 2019-04-11 18:18:43.327345 +0300 EEST m+=8.004588898
f3(): 2019-04-11 18:18:47.328073 +0300 EEST m+=12.005483099
$ go run simpleContext.go 2
Delay: 2
f1(): 2019-04-11 18:18:53.972045 +0300 EEST m+=2.005231943
f2(): context deadline exceeded
f3(): 2019-04-11 18:18:57.974337 +0300 EEST m+=6.007690061
$ go run simpleContext.go 10
Delay: 10
f1(): context canceled
f2(): context canceled
f3(): context canceled
```

The long lines of the output are the return values of the `time.After()` function calls. They denote the normal operation of the program. The point here is that the operation of the program is canceled when there are delays in its execution.

This is as simple as it gets with the use of the `context` package, as the code presented did not do any serious work with the `Context` interface. However, the Go code included in the next section will present a more realistic example.

An advanced example of the context package

The functionality of the `context` package will be illustrated much better and in greater depth by using the Go code of the `useContext.go` program, which is presented in five parts. In this example, you will create an HTTP client that does not want to wait too long for the response of the HTTP server, which is not an unusual scenario. In fact, as almost all HTTP clients support such functionality, you will study another technique for timing out an HTTP request in [Chapter 12, *The Foundations of Network Programming in Go*](#).

The `useContext.go` program requires two command-line arguments: the URL of the server to which it is going to connect and the time for which the presented utility should wait. If the program has only one command-line argument, then the delay will be five seconds.

The first code segment of `useContext.go` follows:

```
package main
```

```
import (
    "context"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "strconv"
    "sync"
    "time"
)

var (
    myUrl string
    delay int = 5
    w      sync.WaitGroup
)

type myData struct {
    r *http.Response
    err error
}
```

Both `myUrl` and `delay` are global variables, so they can be accessed by anything in the code. Additionally, there is a `sync.WaitGroup` variable named `w`, which also has a global scope, and the definition of a structure named `myData` for keeping together the response of the web server, along with an error variable in case there is an error somewhere. The second part of `useContext.go` is shown in the following Go code:

```
func connect(c context.Context) error {
    defer w.Done()
    data := make(chan myData, 1)

    tr := &http.Transport{}
    httpClient := &http.Client{Transport: tr}

    req, _ := http.NewRequest("GET", myUrl, nil)
```

The preceding Go code deals with the HTTP connection.



You will learn more about developing HTTP servers and clients in Go in Chapter 12, *The Foundations of Network Programming in Go*.

The third code portion of `useContext.go` contains the following Go code:

```
go func() {
```

```
    response, err := httpClient.Do(req)
    if err != nil {
        fmt.Println(err)
        data <- myData{nil, err}
        return
    } else {
        pack := myData{response, err}
        data <- pack
    }
}()
```

The fourth code segment of `useContext.go` is shown in the following Go code:

```
select {
case <-c.Done():
    tr.CancelRequest(req)
    <-data
    fmt.Println("The request was cancelled!")
    return c.Err()
case ok := <-data:
    err := ok.err
    resp := ok.r
    if err != nil {
        fmt.Println("Error select:", err)
        return err
    }
    defer resp.Body.Close()

    realHTTPData, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        fmt.Println("Error select:", err)
        return err
    }
    fmt.Printf("Server Response: %s\n", realHTTPData)
}
return nil
}
```

The remaining code of `useContext.go`, which is the implementation of the `main()` function, is as follows:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Need a URL and a delay!")
        return
    }
}
```



```

myUrl = os.Args[1]
if len(os.Args) == 3 {
    t, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
    delay = t
}

fmt.Println("Delay:", delay)
c := context.Background()
c, cancel := context.WithTimeout(c, time.Duration(delay)*time.Second)
defer cancel()

fmt.Printf("Connecting to %s \n", myUrl)
w.Add(1)
go connect(c)
w.Wait()
fmt.Println("Exiting...")
}

```

The timeout period is defined by the `context.WithTimeout()` method. The `connect()` function that is executed as a goroutine will either terminate normally or when the `cancel()` function is executed. Notice that it is considered good practice to use `context.Background()` in the `main()` function or the `init()` function of a package or in tests.

Although it is not necessary to know about the server side of the operation, it is good to see how a Go version of a web server can be slow in a random way. In this case, a random number generator decides how slow your web server will be – real web servers might be too busy to answer or there might be network issues that cause the delay. The name of the source file is `slowWWW.go`, and its contents are as follows:

```

package main

import (
    "fmt"
    "math/rand"
    "net/http"
    "os"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

```

```
}

func myHandler(w http.ResponseWriter, r *http.Request) {
    delay := random(0, 15)
    time.Sleep(time.Duration(delay) * time.Second)

    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Fprintf(w, "Delay: %d\n", delay)
    fmt.Printf("Served: %s\n", r.Host)
}

func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", myHandler)
    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

As you can see, you do not need to use the `context` package in the `slowWWW.go` file because it is the job of the web client to decide how much time it can wait for a response.

The code of the `myHandler()` function is responsible for the slowness of the web server program. The delay can be from 0 seconds to 14 seconds, as introduced by the `random(0, 15)` function call.

If you try to use the `slowWWW.go` web server using a tool such as `wget (1)`, you will receive the following type of output:

```
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 4
$ wget -qO- http://localhost:8001/
Serving: /
Delay: 13
```

This happens because the default timeout value of `wget (1)` is larger. Executing `useContext.go` while `slowWWW.go` is already running in another UNIX shell will create the following type of output when processed with the handy `time (1)` utility:

```
$ time go run useContext.go http://localhost:8001/ 1
Delay: 1
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...
real    0m1.374s
user    0m0.304s
sys     0m0.117s
$ time go run useContext.go http://localhost:8001/ 10
Delay: 10
Connecting to http://localhost:8001/
Get http://localhost:8001/: net/http: request canceled
The request was cancelled!
Exiting...
real    0m10.381s
user    0m0.314s
sys     0m0.125s
$ time go run useContext.go http://localhost:8001/ 15
Delay: 15
Connecting to http://localhost:8001/
Server Response: Serving: /
Delay: 13
Exiting...
real    0m13.379s
user    0m0.309s
sys     0m0.118s
```

The output shows that only the third command actually got an answer from the HTTP server – the first two commands timed out.

Another example of the context package

In this section, you are going to learn even more about the `context` package because it is such a powerful and unique package of the Go standard library. This time, we are going to create a context that uses the `context.TODO()` function instead of the `context.Background()` function. Although both functions return a non-nil, empty `Context`, their purposes differ. We are also going to illustrate the use of the `context.WithValue()` function. The name of the program that will be developed in this section is `moreContext.go`, and it will be presented in four parts.

The first part of `moreContext.go` is as follows:

```
package main

import (
    "context"
    "fmt"
)

type aKey string
```

The second part of `moreContext.go` contains the following Go code:

```
func searchKey(ctx context.Context, k aKey) {
    v := ctx.Value(k)
    if v != nil {
        fmt.Println("found value:", v)
        return
    } else {
        fmt.Println("key not found:", k)
    }
}
```

This is a function that retrieves a value from a context and checks whether that value exists or not.

The third part of `moreContext.go` contains the following Go code:

```
func main() {
    myKey := aKey("mySecretValue")
    ctx := context.WithValue(context.Background(), myKey, "mySecretValue")

    searchKey(ctx, myKey)
```

The `context.WithValue()` function offers a way of associating a value with a Context.

Notice that contexts should be not stored in structures – they should be passed as separate parameters to functions. It is considered good practice to pass them as the first parameter of a function.

The last part of `moreContext.go` is as follows:

```
searchKey(ctx, aKey("notThere"))
emptyCtx := context.TODO()
searchKey(emptyCtx, aKey("notThere"))
}
```

In this case, we declare that although we intend to use an operation context, we are not sure about it yet – this is signified by the use of the `context.TODO()` function. The good thing is that `TODO()` is recognized by static analysis tools, which allows them to determine whether Contexts are propagated correctly in a program or not.

Executing `moreContext.go` will generate the following output:

```
$ go run moreContext.go
found value: mySecretValue
key not found: notThere
key not found: notThere
```

Remember that you should never pass a `nil` context – use the `context.TODO()` function to create a suitable context – and remember that the `context.TODO()` function should be used when you are not sure about the `Context` that you want to use.

Worker pools

Generally speaking, a **worker pool** is a set of threads that are about to process jobs assigned to them. The Apache web server and the `net/http` package of Go more or less work this way: the main process accepts all incoming requests, which are forwarded to the worker processes to get served. Once a worker process has finished its job, it is ready to serve a new client.

Nevertheless, there is a central difference here because our worker pool is going to use goroutines instead of threads. Additionally, threads do not usually die after serving a request because the cost of ending a thread and creating a new one is too high, whereas goroutines do die after finishing their job. As you will see shortly, worker pools in Go are implemented with the help of buffered channels, because they allow you to limit the number of goroutines running at the same time.

The next program, `workerPool.go`, will be presented in five parts. The program will implement a simple task: it will process integers and print their square values using a single goroutine to serve each request. Despite the simplicity of `workerPool.go`, the Go code of the program can be easily used as a template for implementing much more difficult tasks.



This is an advanced technique that can help you to create server processes in Go that can accept and serve multiple clients using goroutines.

The first part of `workerPool.go` follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
    "sync"
    "time"
)

type Client struct {
    id      int
    integer int
}

type Data struct {
    job      Client
    square int
}
```

Here, you can see a technique that uses the `Client` structure to assign a unique ID to each request that you are going to process. The `Data` structure is used to group the data of a `Client` with the actual results generated by the program. Put simply, the `Client` structure holds the input data of each request, whereas the `Data` structure holds the results of a request.

The second code portion of `workerPool.go` is shown in the following Go code:

```
var (
    size      = 10
    clients = make(chan Client, size)
    data      = make(chan Data, size)
)

func worker(w *sync.WaitGroup) {
    for c := range clients {
        square := c.integer * c.integer
        output := Data{c, square}
        data <- output
        time.Sleep(time.Second)
    }
    w.Done()
}
```

The preceding code has two interesting parts. The first part creates three global variables. The `clients` and `data` buffered channels are used to get new client requests and write the results, respectively. If you want your program to run faster, you can increase the value of the `size` parameter.

The second part is the implementation of the `worker()` function, which reads the `clients` channel in order to get new requests to serve. Once the processing is complete, the result is written to the `data` channel. The delay that is introduced using the `time.Sleep(time.Second)` statement is not necessary, but it gives you a better sense of the way that the generated output will be printed.

Finally, remember to use a pointer for the `sync.WaitGroup` parameter in the `worker()` function because, otherwise, the `sync.WaitGroup` variable is copied, which means that it will be useless.

The third part of `workerPool.go` contains the following Go code:

```
func makeWP(n int) {
    var w sync.WaitGroup
    for i := 0; i < n; i++ {
        w.Add(1)
        go worker(&w)
    }
    w.Wait()
    close(data)
}

func create(n int) {
    for i := 0; i < n; i++ {
        c := Client{i, i}
        clients <- c
    }
    close(clients)
}
```

The preceding code implements two functions, named `makeWP()` and `create()`. The purpose of the `makeWP()` function is to generate the required number of `worker()` goroutines to process all requests. Although the `w.Add(1)` function is called in `makeWP()`, `w.Done()` is called in the `worker()` function once a worker has finished its job. The purpose of the `create()` function is to create all requests properly using the `Client` type and then write them to the `clients` channel for processing. Note that the `clients` channel is read by the `worker()` function.

The fourth code segment of `workerPool.go` is as follows:

```
func main() {
    fmt.Println("Capacity of clients:", cap(clients))
    fmt.Println("Capacity of data:", cap(data))

    if len(os.Args) != 3 {
        fmt.Println("Need #jobs and #workers!")
        os.Exit(1)
    }

    nJobs, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    nWorkers, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

In the preceding code, you read your command-line parameters. First, however, you see that you can use the `cap()` function to find the capacity of a channel.

If the number of workers is greater than the size of the `clients` buffered channel, then the number of goroutines that are going to be created will be equal to the size of the `clients` channel. Similarly, if the number of jobs is greater than the number of workers, the jobs will be served in smaller sets.

The program allows you to define the number of workers and the number of jobs using its command-line arguments. However, in order to change the size of the `clients` and `data` channels, you will need to make changes to the source code of the program.

The remaining code of `workerPool.go` follows:

```
go create(nJobs)
finished := make(chan interface{})
go func() {
    for d := range data {
        fmt.Printf("Client ID: %d\tint: ", d.job.id)
        fmt.Printf("%d\tsquare: %d\n", d.job.integer, d.square)
    }
    finished <- true
}()
```



```
    makeWP(nWorkers)
    fmt.Printf(" : %v\n", <-finished)
}
```

First, you call the `create()` function to mimic the client requests that you will have to process. An anonymous goroutine is used to read the `data` channel and print the output to the screen. The `finished` channel is used to block the program until the anonymous goroutine is done reading the `data` channel. Therefore, the `finished` channel needs no particular type.

Next, you call the `makeWP()` function to actually process the requests. The `<-finished` statement in `fmt.Printf()` blocks means that it does not allow the program to end until somebody writes something to the `finished` channel. That somebody is the anonymous goroutine of the `main()` function. Additionally, although the anonymous function writes the `true` value to the `finished` channel, you could have written `false` to it and had the same result, which is unblocking the `main()` function. Try it on your own!

Executing `workerPool.go` will generate the following type of output:

```
$ go run workerPool.go 15 5
Capacity of clients: 10
Capacity of data: 10
Client ID: 0      int: 0      square: 0
Client ID: 4      int: 4      square: 16
Client ID: 1      int: 1      square: 1
Client ID: 3      int: 3      square: 9
Client ID: 2      int: 2      square: 4
Client ID: 5      int: 5      square: 25
Client ID: 6      int: 6      square: 36
Client ID: 7      int: 7      square: 49
Client ID: 8      int: 8      square: 64
Client ID: 9      int: 9      square: 81
Client ID: 10     int: 10     square: 100
Client ID: 11     int: 11     square: 121
Client ID: 12     int: 12     square: 144
Client ID: 13     int: 13     square: 169
Client ID: 14     int: 14     square: 196
: true
```

When you want to serve each individual request without expecting an answer from it in the `main()` function, as happened with `workerPool.go`, you have fewer things to worry about. A simple way to use goroutines for processing your requests and to get an answer from them in the `main()` function is by using shared memory or a monitor process that will collect the data instead of just printing it on the screen.

Finally, the work of the `workerPool.go` program is much simpler because the `worker()` function cannot fail. This will not be the case when you have to work over computer networks or with other kinds of resources that can fail.

Additional resources

The following are very useful resources:

- Visit the documentation page of the `sync` package, which can be found at <https://golang.org/pkg/sync/>.
- Visit the documentation page of the `context` package, which can be found at <https://golang.org/pkg/context/>.
- You can learn more about the implementation of the Go scheduler by visiting <https://golang.org/src/runtime/proc.go>.
- You can find the documentation page of the `atomic` package at <https://golang.org/pkg/sync/atomic/>.
- You can view the design document of the Go scheduler at <https://golang.org/s/go11sched>.

Exercises

- Try to implement a concurrent version of `wc(1)` that uses a buffered channel.
- Next, try to implement a concurrent version of `wc(1)` that uses shared memory.
- Finally, try to implement a concurrent version of `wc(1)` that uses a monitor goroutine.
- Modify the Go code of `workerPool.go` in order to save the results in a file. Use a mutex and a critical section while dealing with the file or a monitor goroutine that will keep writing your data on the disk.
- What will happen to the `workerPool.go` program when the value of the `size` global variable becomes 1? Why?
- Modify the Go code of `workerPool.go` in order to implement the functionality of the `wc(1)` command-line utility.
- Modify the Go code of `workerPool.go` so that the size of the `clients` and `data` buffered channels can be defined using command-line arguments.

- Try to write a concurrent version of the `find(1)` command-line utility that uses a monitor goroutine.
- Modify the code of `simpleContext.go` so that the anonymous function used in all `f1()`, `f2()`, and `f3()` functions becomes a separate function. What is the main challenge of this code change?
- Modify the Go code of `simpleContext.go` so that all `f1()`, `f2()`, and `f3()` functions use an externally-created `Context` variable instead of defining their own.
- Modify the Go code of `useContext.go` in order to use either `context.WithDeadline()` or `context.WithCancel()` instead of `context.WithTimeout()`.
- Finally, try to implement a concurrent version of the `find(1)` command-line utility using a `sync.Mutex` `mutex`.

Summary

This chapter addressed many important topics related to goroutines and channels. Mainly, however, it clarified the power of the `select` statement. Due to the capabilities of the `select` statement, channels are the preferred Go way for interconnecting the components of a concurrent Go program that utilizes multiple goroutines. Additionally, the chapter demonstrated the use of the `context` standard Go package, which, when needed, is irreplaceable.

There are many rules in concurrent programming; however, the most important rule is that you should avoid sharing things unless you have a pretty important reason to do so! Shared data is the root of all nasty bugs in concurrent programming.

What you must remember from this chapter is that, although shared memory used to be the only way of exchanging data over the threads of the same process, Go offers better ways for goroutines to communicate with each other, so think in Go terms before deciding to use shared memory in your Go code. Nonetheless, if you really have to use shared memory, you might want to use a monitor goroutine instead.

The primary subjects of the next chapter will be code testing, code optimization, and code profiling with Go. Apart from these topics, you will learn about benchmarking Go code, cross-compilation, and finding unreachable Go code.

At the end of the next chapter, you will also learn how to document your Go code and how to generate HTML output using the `godoc` utility.

11

Code Testing, Optimization, and Profiling

The previous chapter discussed concurrency in Go, mutexes, the `atomic` package, the various types of channels, race conditions, and how the `select` statement allows you to use channels as glue to control goroutines and allow them to communicate.

The Go topics in this chapter are very practical and important, especially if you are interested in improving the performance of your Go programs and discovering bugs quickly. This chapter primarily addresses code optimization, code testing, code documentation, and code profiling.

Code optimization is a process where one or more developers try to make certain parts of a program run faster, be more efficient, or use fewer resources. Put simply, code optimization is about eliminating the bottlenecks of a program.

Code testing is about making sure that your code does what you want it to do. In this chapter, you will experience the Go way of testing code. The best time to write code to test your programs is during the development phase, as this can help to reveal bugs in your code as early as possible.

Code profiling relates to measuring certain aspects of a program in order to get a detailed understanding of the way the code works. The results of code profiling may help you to decide which parts of your code need to change.

I hope that you already recognize the importance of documenting your code in order to describe the decisions you made while developing the implementation of your program. In this chapter, you will see how Go can help you to generate documentation for the modules that you implement.



Documentation is so important that some developers write the documentation first and the code afterward! However, what is really important is that the documentation and the functionality of the program say and do the same thing, respectively.

In this chapter, you will learn about the following topics:

- Profiling Go code
- The `go tool pprof` utility
- Using the web interface of the Go profiler
- Testing Go code
- The `go test` command
- The `go tool trace` utility
- The handy `testing/quick` package
- Benchmarking Go code
- Cross-compilation
- Testing code coverage
- Generating documentation for your Go code
- Creating example functions
- Finding unreachable Go code in your programs

About optimization

Code optimization is both an art and a science. This means that there is no deterministic way to help you optimize your Go code, or any other code in any programming language, and that you should use your brain and try many things if you want to make your code run faster.



You should make sure that you optimize code that does not contain any bugs, because there is no point in optimizing a bug. If you have any bugs in your program, you should debug it first.

If you are really into code optimization, you might want to read *Compilers: Principles, Techniques, and Tools* by Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman (Pearson Education Limited, 2014), which focuses on compiler construction. Additionally, all volumes in *The Art of Computer Programming* series by Donald Knuth (Addison-Wesley Professional, 1998) are great resources for all aspects of programming.

Always remember what Knuth said about optimization:

"The real problem is that programmers have spent far too much time worrying about efficiency in the wrong places and at the wrong times; premature optimization is the root of all evil (or at least most of it) in programming."

Also, remember what Joe Armstrong, one of the developers of Erlang, said about optimization:

"Make it work, then make it beautiful, then if you really, really have to, make it fast. 90 percent of the time, if you make it beautiful, it will already be fast. So really, just make it beautiful!"

Additionally, generally speaking, only a small percentage of a program needs to be optimized. In such cases, the assembly programming language, which can be used to implement certain Go functions, is a really good candidate and will have a huge impact on the performance of your programs.

Optimizing Go code

As mentioned, code optimization is the process where you try to discover the parts of your code that have a big impact on the performance of the entire program in order to make them run faster or use fewer resources.

The *benchmarking* section that appears later in this chapter will greatly help you to understand what is going on with your code behind the scenes and which parameters of your program have the greatest impact on the performance of your program. However, do not underestimate the importance of common sense. Put simply, if one of your functions is executed 10,000 times more than the rest of the functions of a program, try to optimize that function first.



The general advice for optimization is that you must optimize bug-free code only. This means that you must optimize working code only. Therefore, you should first try to write correct code even if that code is slow. Finally, the single most frequent mistake that programmers make is trying to optimize the first version of their code, which is the root of most bugs!

Again, code optimization is both an art and a science, which means that it is a pretty difficult task. The next section about profiling Go code will definitely help you with code optimization because the main purpose of profiling is to find the bottlenecks in your code in order to optimize the most important parts of your program.

Profiling Go code

Profiling is a process of dynamic program analysis that measures various values related to program execution in order to give you a better understanding of the behavior of your program. In this section, you are going to learn how to profile Go code in order to understand your code better and improve its performance. Sometimes, code profiling can even reveal bugs!

First, we are going to use the command-line interface of the Go profiler. Then, we will use the web interface of the Go profiler.

The single most important thing to remember is that if you want to profile Go code, you will need to import the `runtime/pprof` standard Go package, either directly or indirectly. You can find the help page of the `pprof` tool by executing the `go tool pprof -help` command, which will generate lots of output.

The `net/http/pprof` standard Go package

Although Go comes with the low-level `runtime/pprof` standard Go package, there is also the high-level `net/http/pprof` package, which should be used when you want to profile a web application written in Go. As this chapter will not talk about creating HTTP servers in Go, you will learn more about the `net/http/pprof` package in Chapter 12, *The Foundations of Network Programming in Go*.

A simple profiling example

Go supports two kinds of profiling: **CPU profiling** and **memory profiling**. It is not recommended that you profile an application for both kinds at the same time, because these two different kinds of profiling do not work well with each other. The `profileMe.go` application is an exception, however, because it is used to illustrate the two techniques.

The Go code to be profiled is saved as `profileMe.go`, and it will be presented in five parts. The first part of `profileMe.go` is shown in the following Go code:

```
package main
import (
    "fmt"
    "math"
    "os"
    "runtime"
    "runtime/pprof"
```

```

        "time"
    )
    func fibo1(n int) int64 {
        if n == 0 || n == 1 {
            return int64(n)
        }
        time.Sleep(time.Millisecond)
        return int64(fibo2(n-1)) + int64(fibo2(n-2))
    }

```

Notice that it is compulsory to import the `runtime/pprof` package directly or indirectly for your program to create profiling data. The reason for calling `time.Sleep()` in the `fibo1()` function is to slow it down a bit. You will learn why near the end of this section.

The second code segment of `profileMe.go` follows:

```

func fibo2(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    time.Sleep(50 * time.Millisecond)
    return fn[n]
}

```

The preceding code contains the implementation of another Go function that uses a different algorithm for calculating numbers of the Fibonacci sequence.

The third part of `profileMe.go` contains the following Go code:

```

func N1(n int) bool {
    k := math.Floor(float64(n/2 + 1))
    for i := 2; i < int(k); i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}

func N2(n int) bool {
    for i := 2; i < n; i++ {

```



```
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

Both the `N1()` and `N2()` functions are used to find out whether a given integer is a prime number or not. The first function is optimized because its `for` loop iterates over approximately half the numbers used in the `for` loop of `N2()`. As both functions are relatively slow, there is no need for a call to `time.Sleep()` here.

The fourth code segment of `profileMe.go` is as follows:

```
func main() {
    cpuFile, err := os.Create("/tmp/cpuProfile.out")
    if err != nil {
        fmt.Println(err)
        return
    }
    pprof.StartCPUProfile(cpuFile)
    defer pprof.StopCPUProfile()
    total := 0
    for i := 2; i < 100000; i++ {
        n := N1(i)
        if n {
            total = total + 1
        }
    }
    fmt.Println("Total primes:", total)
    total = 0
    for i := 2; i < 100000; i++ {
        n := N2(i)
        if n {
            total = total + 1
        }
    }
    fmt.Println("Total primes:", total)
    for i := 1; i < 90; i++ {
        n := fibo1(i)
        fmt.Print(n, " ")
    }
    fmt.Println()
    for i := 1; i < 90; i++ {
        n := fibo2(i)
        fmt.Print(n, " ")
    }
}
```

```
fmt.Println()
runtime.GC()
```

The call to `os.Create()` is used to have a file to which to write the profiling data. The `pprof.StartCPUProfile()` call begins the **CPU profiling** of the program, and the call to `pprof.StopCPUProfile()` stops it.

If you want to create and use temporary files and directories multiple times, then you should have a look at `ioutil.TempFile()` and `ioutil.TempDir()` respectively.

The last part of `profileMe.go` follows:

```
// Memory profiling!
memory, err := os.Create("/tmp/memoryProfile.out")
if err != nil {
    fmt.Println(err)
    return
}
defer memory.Close()
for i := 0; i < 10; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(50 * time.Millisecond)
}
err = pprof.WriteHeapProfile(memory)
if err != nil {
    fmt.Println(err)
    return
}
}
```

In the last part, you can see how the **memory profiling** technique works. It is pretty similar to CPU profiling, and once again, you will need a file to write out the profiling data.

Executing `profileMe.go` will generate the following output:

```
$ go run profileMe.go
Total primes: 9592
Total primes: 9592
1 2 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
```

```

956722026041 1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853 72723460248141
117669030460994 190392490709135 308061521170129 498454011879264
806515533049393 1304969544928657 2111485077978050 3416454622906707
5527939700884757 8944394323791464 14472334024676221 23416728348467685
37889062373143906 61305790721611591 99194853094755497 160500643816367088
259695496911122585 420196140727489673 679891637638612258
1100087778366101931 1779979416004714189
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584 4181 6765 10946
17711 28657 46368 75025 121393 196418 317811 514229 832040 1346269 2178309
3524578 5702887 9227465 14930352 24157817 39088169 63245986 102334155
165580141 267914296 433494437 701408733 1134903170 1836311903 2971215073
4807526976 7778742049 12586269025 20365011074 32951280099 53316291173
86267571272 139583862445 225851433717 365435296162 591286729879
956722026041 1548008755920 2504730781961 4052739537881 6557470319842
10610209857723 17167680177565 27777890035288 44945570212853 72723460248141
117669030460994 190392490709135 308061521170129 498454011879264
806515533049393 1304969544928657 2111485077978050 3416454622906707
5527939700884757 8944394323791464 14472334024676221 23416728348467685
37889062373143906 61305790721611591 99194853094755497 160500643816367088
259695496911122585 420196140727489673 679891637638612258
1100087778366101931 1779979416004714189

```

Apart from the output, the program will also collect the profiling data in two files:

```

$ cd /tmp
$ ls -l *Profile*
-rw-r--r-- 1 mtsouk wheel 1557 Apr 24 16:37 cpuProfile.out
-rw-r--r-- 1 mtsouk wheel 438 Apr 24 16:37 memoryProfile.out

```

It is only after collecting the profiling data that you can start to inspect it. Thus, you can now start the command-line profiler to examine the CPU data as follows:

```

$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)

```

Pressing `help` while in the profiler shell will generate the following output:

```

(pprof) help
Commands:
  callgrind      Outputs a graph in callgrind format
  comments       Output all profile comments
  disasm         Output assembly listings annotated with samples
  dot            Outputs a graph in DOT format

```

eog	Visualize graph through eog
evince	Visualize graph through evince
gif	Outputs a graph image in GIF format
gv	Visualize graph through gv
kcachegrind	Visualize report in KCachegrind
list	Output annotated source for functions matching regexp
pdf	Outputs a graph in PDF format
peek	Output callers/callees of functions matching regexp
png	Outputs a graph image in PNG format
proto	Outputs the profile in compressed protobuf format
ps	Outputs a graph in PS format
raw	Outputs a text representation of the raw profile
svg	Outputs a graph in SVG format
tags	Outputs all tags in the profile
text	Outputs top entries in text form
top	Outputs top entries in text form
topproto	Outputs top entries in compressed protobuf format
traces	Outputs all profile samples in text form
tree	Outputs a text rendering of call graph
web	Visualize graph through web browser
weblist	Display annotated source in a web browser
o/options	List options and their current values
quit/exit/^D	Exit pprof
Options:	
call_tree	Create a context-sensitive call tree
compact_labels	Show minimal headers
divide_by	Ratio to divide all samples before visualization
drop_negative	Ignore negative differences
edgefraction	Hide edges below <f>*total
focus	Restricts to samples going through a node matching
regexp	
hide	Skips nodes matching regexp
ignore	Skips paths going through any nodes matching regexp
mean	Average sample value over first value (count)
nodecount	Max number of nodes to show
nodefraction	Hide nodes below <f>*total
noinlines	Ignore inlines.
normalize	Scales profile based on the base profile.
output	filename for file-based outputs
prune_from	Drops any functions below the matched frame.
relative_percentages	Show percentages relative to focused subgraph
sample_index	Sample value to report (0-based index or name)
show	Only show nodes matching regexp
show_from	Drops functions above the highest matched frame.
source_path	Search path for source files
tagfocus	Restricts to samples with tags in range or matched by
regexp	
taghide	Skip tags matching this regexp

```

    tagignore      Discard samples with tags in range or matched by
regexp
    tagshow        Only consider tags matching this regexp
    trim          Honor nodefraction/edgefraction/nodecount defaults
    trim_path      Path to trim from source paths before search
    unit           Measurement units to display
Option groups (only set one per group):
    cumulative
        cum        Sort entries based on cumulative weight
        flat       Sort entries based on own weight
    granularity
        addresses   Aggregate at the address level.
        filefunctions Aggregate at the function level.
        files       Aggregate at the file level.
        functions   Aggregate at the function level.
        lines       Aggregate at the source code line level.
: Clear focus/ignore/hide/tagfocus/tagignore
type "help <cmd|option>" for more information
(pprof)

```



Do find the time to try out all of the commands of the `go tool pprof` utility and familiarize yourself with them.

The `top` command returns the top 10 entries in text form:

```

(pprof) top
Showing nodes accounting for 4.42s, 99.10% of 4.46s total
Dropped 14 nodes (cum <= 0.02s)
Showing top 10 nodes out of 19

```

flat	flat%	sum%	cum	cum%	
2.69s	60.31%	60.31%	2.69s	60.31%	main.N2
1.41s	31.61%	91.93%	1.41s	31.61%	main.N1
0.19s	4.26%	96.19%	0.19s	4.26%	runtime.nanotime
0.10s	2.24%	98.43%	0.10s	2.24%	runtime.usleep
0.03s	0.67%	99.10%	0.03s	0.67%	runtime.memclrNoHeapPointers
0	0%	99.10%	4.14s	92.83%	main.main
0	0%	99.10%	0.03s	0.67%	runtime.(*mheap).alloc
0	0%	99.10%	0.03s	0.67%	runtime.largeAlloc
0	0%	99.10%	4.14s	92.83%	runtime.main
0	0%	99.10%	0.03s	0.67%	runtime.makeslice

As the first line of the output states, the functions presented are responsible for 99.10% of the total execution time of the program.

The `main.N2` function in particular is responsible for 60.31% of the execution time of the program.

The `top10 --cum` command returns the cumulative time for each function:

```
(pprof) top10 --cum
Showing nodes accounting for 4390ms, 98.43% of 4460ms total
Dropped 14 nodes (cum <= 22.30ms)
Showing top 10 nodes out of 19
```

	flat	flat%	sum%	cum	cum%	
	0	0%	0%	4140ms	92.83%	main.main
	0	0%	0%	4140ms	92.83%	runtime.main
	2690ms	60.31%	60.31%	2690ms	60.31%	main.N2
	1410ms	31.61%	91.93%	1410ms	31.61%	main.N1
	0	0%	91.93%	290ms	6.50%	runtime.mstart
	0	0%	91.93%	270ms	6.05%	runtime.mstart1
	0	0%	91.93%	270ms	6.05%	runtime.sysmon
	190ms	4.26%	96.19%	190ms	4.26%	runtime.nanotime
	100ms	2.24%	98.43%	100ms	2.24%	runtime.usleep
	0	0%	98.43%	50ms	1.12%	runtime.systemstack

What if you want to find out what is happening with a particular function? You can use the `list` command followed by the function name, combined with the package name, and you'll get more detailed information about the performance of that function:

```
(pprof) list main.N1
Total: 4.18s
ROUTINE ===== main.N1 in /Users/mtsouk/ch11/code/profileMe.go
  1.41s      1.41s (flat, cum) 31.61% of Total
    .          .      32:    return fn[n]
    .          .      33:}
    .          .      34:
    .          .      35:func N1(n int) bool {
    .          .      36:    k := math.Floor(float64(n/2 + 1))
  60ms      60ms      37:    for i := 2; i < int(k); i++ {
  1.35s      1.35s      38:        if (n % i) == 0 {
    .          .      39:            return false
    .          .      40:        }
    .          .      41:    }
    .          .      42:    return true
    .          .      43:}
(pprof)
```

The output shows that the `for` loop of `main.N1` is responsible for almost all of the execution time of the entire function. Specifically, the `if (n % i) == 0` statement is responsible for 1.35s out of 1.41s of the execution time of the entire function.

You can also create PDF output of the profiling data from the shell of the Go profiler using the `pdf` command:

```
(pprof) pdf
Generating report in profile001.pdf
```

Note that you will need **Graphviz** in order to generate a PDF file that can be viewed using your favorite PDF reader.

Finally, a warning: if your program executes too quickly, then the profiler will not have enough time to get its required samples and you might see the `Total samples = 0` output when you load the data file. In that case, you will not be able to get any useful information from the profiling process. That is the reason for using the `time.Sleep()` function in some of the functions of `profileMe.go`:

```
$ go tool pprof /tmp/cpuProfile.out
Type: cpu
Time: Apr 24, 2019 at 4:37pm (EEST)
Duration: 19.59s, Total samples = 4.46s (22.77%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

A convenient external package for profiling

In this subsection, you will see the use of an external Go package that sets up the profiling environment much more conveniently than by using the `runtime/pprof` standard Go package. This point is illustrated in `betterProfile.go`, which will be presented in three parts.

The first part of `betterProfile.go` is as follows:

```
package main
import (
    "fmt"
    "github.com/pkg/profile"
)
var VARIABLE int
func N1(n int) bool {
    for i := 2; i < n; i++ {
        if (n % i) == 0 {
            return false
        }
    }
    return true
}
```

In the preceding code, you can see the use of an external Go package that can be found at github.com/pkg/profile. You can download it with the help of the `go get` command, as follows:

```
$ go get github.com/pkg/profile
```

The second code segment of `betterProfile.go` contains the following Go code:

```
func Multiply(a, b int) int {
    if a == 1 {
        return b
    }
    if a == 0 || b == 0 {
        return 0
    }
    if a < 0 {
        return -Multiply(-a, b)
    }
    return b + Multiply(a-1, b)
}
func main() {
    defer profile.Start(profile.ProfilePath("/tmp")).Stop()
```

The `github.com/pkg/profile` package by Dave Cheney requires that you insert just a single statement in order to enable **CPU profiling** in your Go application. If you want to enable **memory profiling**, you should insert the following statement instead:

```
defer profile.Start(profile.MemProfile).Stop()
```

The remaining Go code of the program is as follows:

```
total := 0
for i := 2; i < 200000; i++ {
    n := N1(i)
    if n {
        total++
    }
}
fmt.Println("Total: ", total)
total = 0
for i := 0; i < 5000; i++ {
    for j := 0; j < 400; j++ {
        k := Multiply(i, j)
        VARIABLE = k
        total++
    }
}
fmt.Println("Total: ", total)
```



```
}
```

Executing `betterProfile.go` generates the following output:

```
$ go run betterProfile.go
2019/04/24 16:44:05 profile: cpu profiling enabled, /tmp/cpu.pprof
Total: 17984
Total: 2000000
2019/04/24 16:44:33 profile: cpu profiling disabled, /tmp/cpu.pprof
```



The `github.com/pkg/profile` package will help you with the data capturing portion; the processing part is the same as before.

```
$ go tool pprof /tmp/cpu.pprof
Type: cpu
Time: Apr 24, 2019 at 4:44pm (EEST)
Duration: 27.40s, Total samples = 25.10s (91.59%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof)
```

The web interface of the Go profiler

The good news is that with Go version 1.10, the `go tool pprof` command comes with a web user interface.



For the web user interface feature to work, you will need to have **Graphviz** installed and your web browser must support JavaScript. If you want to play it safe, use either **Chrome** or **Firefox**.

You can start the interactive Go profiler as follows:

```
$ go tool pprof -http=[host]:[port] aProfile
```

A profiling example that uses the web interface

We will use the data captured from the execution of `profileMe.go` to study the web interface of the Go profiler, as there is no need to create a specific code example to do this. As you learned in the previous subsection, you will first need to execute the following command:

```
$ go tool pprof -http=localhost:8080 /tmp/cpuProfile.out
```


Similarly, the following figure shows the `http://localhost:8080/ui/source` URL of the Go profiler, which displays analytical information for each function of the program:

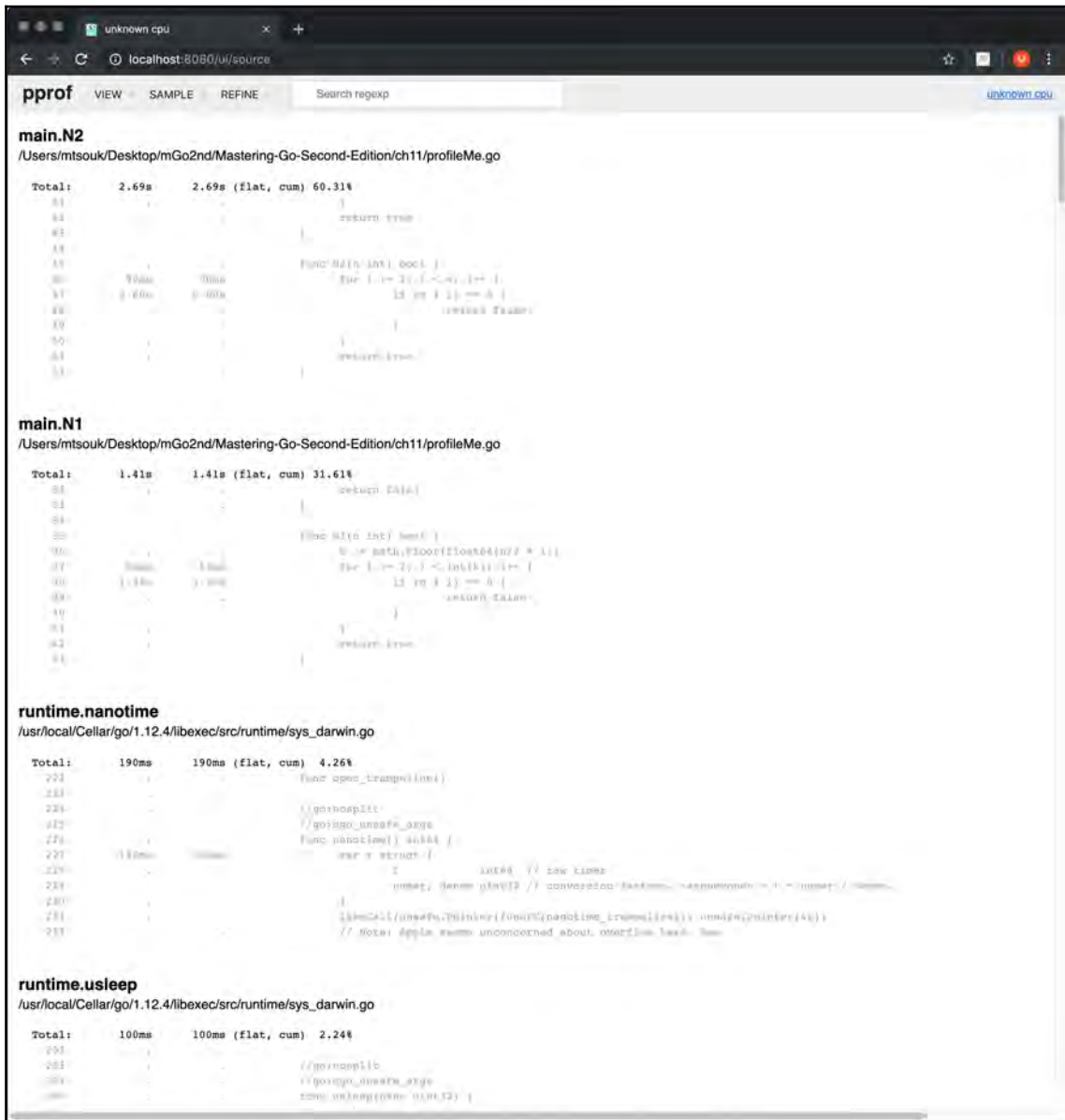


Figure 11.2: Using the `/source` URL of the Go profiler



As I cannot possibly show every single page of the Go profiler web interface, you should start to familiarize yourself with it on your own, as it is a great tool for examining the operations of your programs.

A quick introduction to Graphviz

Graphviz is a very handy compilation of utilities and a computer language that allows you to draw complex graphs. Strictly speaking, Graphviz is a collection of tools for manipulating both directed and undirected **graph** structures and generating graph layouts. Graphviz has its own language, named **DOT**, which is simple, elegant, and powerful. The good thing about Graphviz is that you can write its code using a simple plain text editor. A wonderful side effect of this feature is that you can easily develop scripts that generate Graphviz code. Also, most programming languages, including **Python**, **Ruby**, **C++**, and **Perl**, provide their own interfaces for creating Graphviz files using native code.



You do not need to know all of these things about Graphviz in order to use the web interface of the Go profiler. It is just useful to know how Graphviz works and what its code looks like.

The following Graphviz code, which is saved as `graph.dot`, briefly illustrates the way that Graphviz works and the look of the Graphviz language:

```
digraph G
{
    graph [dpi = 300, bgcolor = "gray"];
    rankdir = LR;
    node [shape=record, width=.2, height=.2, color="white" ];
    node0 [label = "<p0>; |<p1>|<p2>|<p3>|<p4>| | ", height = 3];
    node[ width=2 ];
    node1 [label = "{<e> r0 | 123 | <p> }", color="gray" ];
    node2 [label = "{<e> r10 | 13 | <p> }" ];
    node3 [label = "{<e> r11 | 23 | <p> }" ];
    node4 [label = "{<e> r12 | 326 | <p> }" ];
    node5 [label = "{<e> r13 | 1f3 | <p> }" ];
    node6 [label = "{<e> r20 | 143 | <p> }" ];
    node7 [label = "{<e> r40 | b23 | <p> }" ];
    node0:p0 -> node1:e [dir=both color="red:blue"];
    node0:p1 -> node2:e [dir=back arrowhead=diamond];
    node2:p -> node3:e;
    node3:p -> node4:e [dir=both arrowtail=box color="red"];
    node4:p -> node5:e [dir=forward];
    node0:p2 -> node6:e [dir=none color="orange"];
```

```

    node0:p4 -> node7:e;
}

```

The `color` attribute changes the color of a node, whereas the `shape` attribute changes the shape of a node. Additionally, the `dir` attribute, which can be applied to edges, defines whether an edge is going to have two arrows, one, or none. Furthermore, the style of the arrowhead can be specified using the `arrowhead` and `arrowtail` attributes.

Compiling the preceding code using one of the Graphviz command-line tools in order to create a PNG image requires the execution of the following command in your favorite UNIX shell:

```

$ dot -T png graph.dot -o graph.png
$ ls -l graph.png
-rw-r--r--@ 1 mtsouk staff 94862 Apr 24 16:48 graph.png

```

The next figure shows the graphics file generated from the execution of the preceding command:

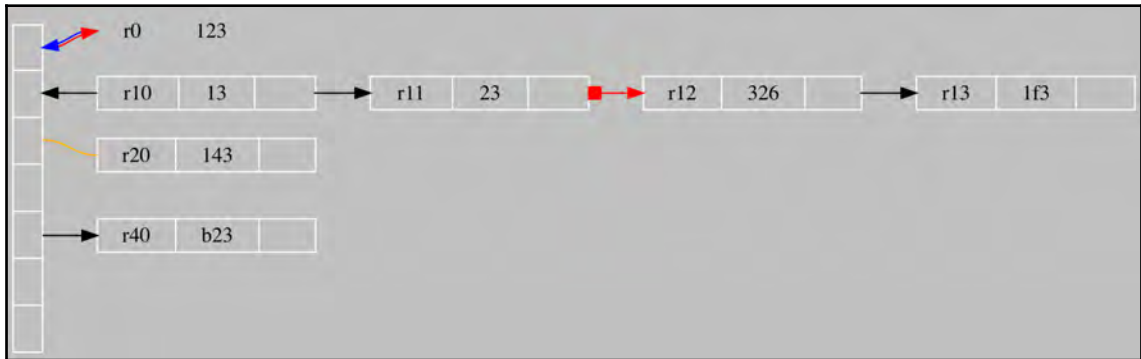


Figure 11.3: Using Graphviz to create graphs

Thus, if you want to visualize any kind of structure, you should definitely consider using Graphviz and its tools, especially if you want to automate things with your own scripts.

The go tool trace utility

The `go tool trace` utility is a tool for viewing trace files that can be generated in any one of the following three ways:

- Using the `runtime/trace` package

- Using the `net/http/pprof` package
- Executing the `go test -trace` command

This section will use the first technique only. The output of the following command will greatly help you to understand what the **Go execution tracer** does:

```
$ go doc runtime/trace
package trace // import "runtime/trace"
Package trace contains facilities for programs to generate traces for
the Go
execution tracer.
Tracing runtime activities
The execution trace captures a wide range of execution events such as
goroutine creation/blocking/unblocking, syscall enter/exit/block, GC-
related
events, changes of heap size, processor start/stop, etc. A precise
nanosecond-precision timestamp and a stack trace is captured for most
events. The generated trace can be interpreted using `go tool trace`.
The trace tool computes the latency of a task by measuring the time
between the task creation and the task end and provides latency
distributions for each task type found in the trace.
func IsEnabled() bool
func Log(ctx context.Context, category, message string)
func Logf(ctx context.Context, category, format string, args
...interface{})
func Start(w io.Writer) error
func Stop()
func WithRegion(ctx context.Context, regionType string, fn func())
type Region struct{ ... }
    func StartRegion(ctx context.Context, regionType string) *Region
type Task struct{ ... }
    func NewTask(pctx context.Context, taskType string) (ctx
context.Context, task *Task)
```

In Chapter 2, *Understanding Go Internals*, we talked about the **Go garbage collector** and presented a Go utility, `gColl.go`, which allowed us to see some of the variables of the Go garbage collector. In this section, we are going to gather even more information about the operation of `gColl.go` using the `go tool trace` utility.

First, let's examine the modified version of the `gColl.go` program, which tells Go to collect performance data. It is saved as `goGC.go`, and it will be presented in three parts.

The first part of `goGC.go` is as follows:

```
package main
import (
```

```
    "fmt "  
    "os "  
    "runtime "  
    "runtime/trace "  
    "time "  
)  
  
func printStats(mem runtime.MemStats) {  
    runtime.ReadMemStats(&mem)  
    fmt.Println("mem.Alloc:", mem.Alloc)  
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)  
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)  
    fmt.Println("mem.NumGC:", mem.NumGC)  
    fmt.Println("-----")  
}
```

As you already know, you first need to import the `runtime/trace` standard Go package in order to collect data for the `go tool trace` utility.

The second code segment of `goGC.go` is shown in the following Go code:

```
func main() {  
    f, err := os.Create("/tmp/traceFile.out")  
    if err != nil {  
        panic(err)  
    }  
    defer f.Close()  
    err = trace.Start(f)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    defer trace.Stop()
```

This part is all about acquiring data for the `go tool trace` utility, and it has nothing to do with the functionality of the actual program. First, you create a new file that will hold the tracing data for the `go tool trace` utility. Then, you start the tracing process using `trace.Start()`. When you are done, you call the `trace.Stop()` function. The `defer` call to this function means that you want to terminate tracing when your program ends.



Using the `go tool trace` utility is a process with two phases that requires extra Go code. First, you collect the data, and then you display and process it.

The remaining Go code is as follows:

```
var mem runtime.MemStats
printStats(mem)
for i := 0; i < 3; i++ {
    s := make([]byte, 50000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
}
printStats(mem)
for i := 0; i < 5; i++ {
    s := make([]byte, 100000000)
    if s == nil {
        fmt.Println("Operation failed!")
    }
    time.Sleep(time.Millisecond)
}
printStats(mem)
}
```

Executing `goGC.go` produces the following output, as well as a new file named `/tmp/traceFile.out` with the tracing information:

```
$ go run goGC.go
mem.Alloc: 108592
mem.TotalAlloc: 108592
mem.HeapAlloc: 108592
mem.NumGC: 0
-----
mem.Alloc: 109736
mem.TotalAlloc: 150127000
mem.HeapAlloc: 109736
mem.NumGC: 3
-----
mem.Alloc: 114672
mem.TotalAlloc: 650172952
mem.HeapAlloc: 114672
mem.NumGC: 8
-----
$ cd /tmp
$ ls -l traceFile.out
-rw-r--r-- 1 mtsouk wheel 10108 Apr 24 16:51 /tmp/traceFile.out
$ file /tmp/traceFile.out
/tmp/traceFile.out: data
```


The `go tool trace` utility uses a web interface that starts automatically when you execute the next command:

```
$ go tool trace /tmp/traceFile.out
2019/04/24 16:52:06 Parsing trace...
2019/04/24 16:52:06 Splitting trace...
2019/04/24 16:52:06 Opening browser. Trace viewer is listening on
http://127.0.0.1:50383
```

The figure that follows shows the initial image of the web interface of the `go tool trace` utility when examining the `/tmp/traceFile.out` trace file.

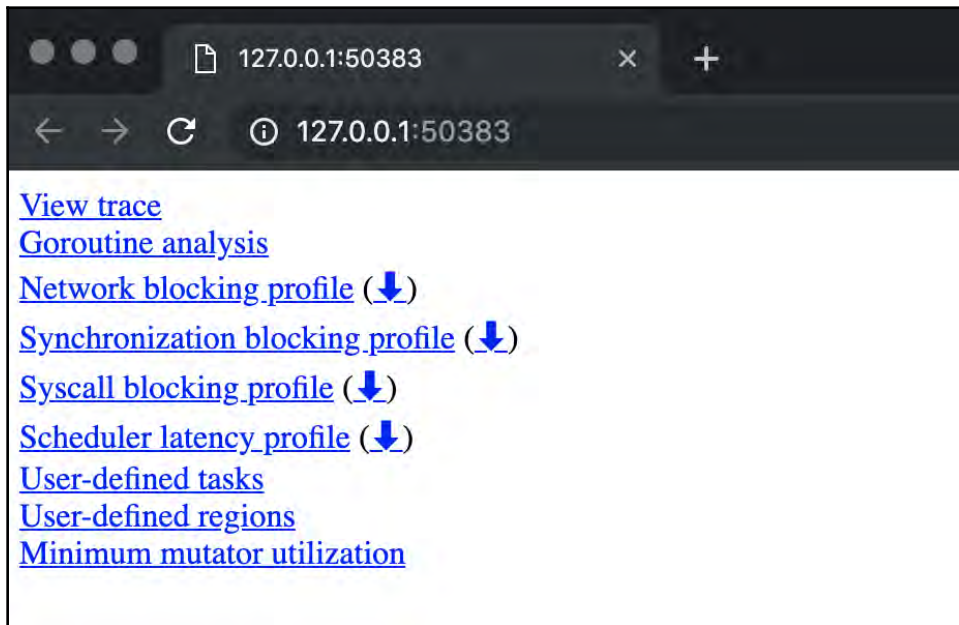
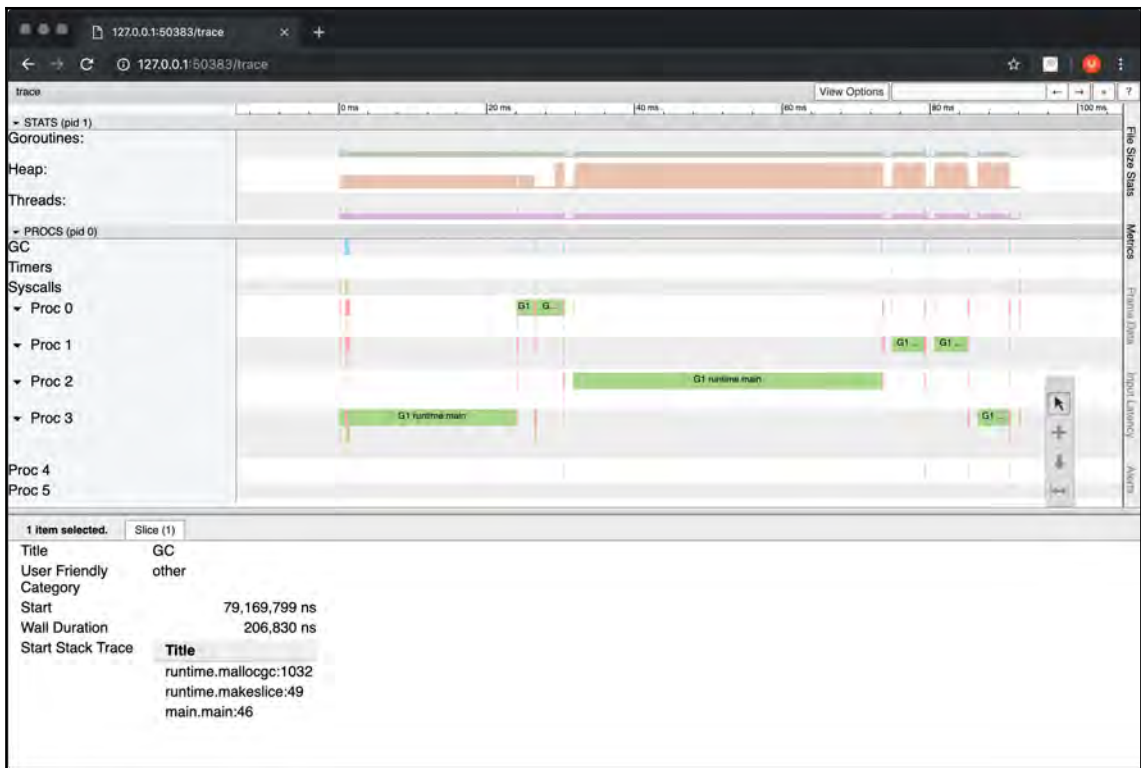


Figure 11.4: The initial screen of the web interface of the `go tool trace` utility

You should now select the `View trace` link. This will take you to the next figure, which shows you another view of the web interface of the `go tool trace` utility that uses the data from `/tmp/traceFile.out`.

Figure 11.5: Examining the operation of the Go garbage collector using `go tool trace`

In the preceding figure, you can see that the Go garbage collector runs on its own goroutine, but that it does not run all of the time. Additionally, you can see the number of goroutines used by the program. You can learn more about this by selecting certain parts of the interactive view. As we are interested in the operation of the garbage collector, a useful piece of information that is displayed is how often and for how long the garbage collector runs.

Notice that although `go tool trace` is a very handy and powerful utility, it cannot solve every kind of performance problem. There are times when `go tool pprof` is more appropriate, especially when you want to reveal where your program spends most of its time by individual function.

Testing Go code

Software testing is a very large subject and cannot be covered in a single section of a chapter in a book. So, this brief section will try to present as much practical information as possible.

Go allows you to write tests for your Go code in order to detect bugs. Strictly speaking, this section is about **automated testing**, which involves writing extra code to verify whether the real code – that is, the production code – works as expected or not. Thus, the result of a test function is either `PASS` or `FAIL`. You will see how this works shortly.

Although the Go approach to testing might look simple at first, especially if you compare it with the testing practices of other programming languages, it is very efficient and effective because it does not require too much of the developer's time.

Go follows certain conventions regarding testing. First of all, testing functions should be included in Go source files that end with `_test.go`. So, if you have a package named `aGoPackage.go`, then your tests should be placed in a file named `aGoPackage_test.go`. A test function begins with `Test`, and it checks the correctness of the behavior of a function of the production package. Finally, you will need to import the `testing` standard Go package for the `go test` subcommand to work correctly. As you will soon see, this import requirement also applies to two additional cases.

Once the testing code is correct, the `go test` subcommand does all the dirty work for you, which includes scanning all `*_test.go` files for special functions, generating a temporary `main` package properly, calling these special functions, getting the results, and generating the final output.



Always put the testing code in a different source file. There is no need to create a huge source file that is hard to read and maintain.

Writing tests for existing Go code

In this section, you will learn how to write tests for an existing Go application that includes two functions: one for calculating numbers of the Fibonacci sequence and one for finding out the length of a string. The main reason for using these two functions that implement such relatively trivial tasks is simplicity. The tricky point here is that each function will have two different implementations: one that works well and another one that has some issues.

The Go package in this example is named `testMe` and it is saved as `testMe.go`. The code of this package will be presented in three parts.

The first part of `testMe.go` includes the following Go code:

```
package testMe
func f1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 1
    }
    return f1(n-1) + f1(n-2)
}
```

In the preceding code, you can see the definition of a function named `f1()`, which calculates natural numbers of the Fibonacci sequence.

The second part of `testMe.go` is shown in the following Go code:

```
func f2(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 {
        return 2
    }
    return f2(n-1) + f2(n-2)
}
```

In this code, you can see the implementation of another function that calculates the numbers of the Fibonacci sequence, named `f2()`. However, this function contains a bug because it does not return 1 when the value of `n` is 1, which destroys the entire functionality of the function.

The remaining code of `testMe.go` is shown in the following Go code:

```
func s1(s string) int {
    if s == "" {
        return 0
    }
    n := 1
    for range s {
        n++
    }
    return n
}
```

```
}

func s2(s string) int {
    return len(s)
}
```

In this part, we implement two functions named `s1()` and `s2()`, which work on strings. Both of these functions find the length of a string. However, the implementation of `s1()` is incorrect because the initial value of `n` is 1 instead of 0.

It is now time to start thinking about tests and test cases. First of all, you should create a `testMe_test.go` file, which will be used to store your testing functions. Next, it is important to realize that you do not need to make any code changes to `testMe.go`. Finally, remember that you should try to write as many tests as required to cover all potential inputs and outputs.

The first part of `testMe_test.go` is shown in the following Go code:

```
package testMe

import "testing"

func TestS1(t *testing.T) {
    if s1("123456789") != 9 {
        t.Error(`s1("123456789") != 9`)
    }
    if s1("") != 0 {
        t.Error(`s1("") != 0`)
    }
}
```

The preceding function performs two tests on the `s1()` function: one using "123456789" as input and another one using "" as input.

The second part of `testMe_test.go` is as follows:

```
func TestS2(t *testing.T) {
    if s2("123456789") != 9 {
        t.Error(`s2("123456789") != 9`)
    }
    if s2("") != 0 {
        t.Error(`s2("") != 0`)
    }
}
```

The preceding testing code performs the same two tests on the `s2()` function.

The remaining code of `testMe_test.go` comes next:

```
func TestF1(t *testing.T) {
    if f1(0) != 0 {
        t.Error(`f1(0) != 0`)
    }
    if f1(1) != 1 {
        t.Error(`f1(1) != 1`)
    }
    if f1(2) != 1 {
        t.Error(`f1(2) != 1`)
    }
    if f1(10) != 55 {
        t.Error(`f1(10) != 55`)
    }
}

func TestF2(t *testing.T) {
    if f2(0) != 0 {
        t.Error(`f2(0) != 0`)
    }
    if f2(1) != 1 {
        t.Error(`f2(1) != 1`)
    }
    if f2(2) != 1 {
        t.Error(`f2(2) != 1`)
    }
    if f2(10) != 55 {
        t.Error(`f2(10) != 55`)
    }
}
```

The previous code tests the operation of the `f1()` and `f2()` functions.

Executing the tests will generate the following type of output:

```
$ go test testMe.go testMe_test.go -v
=== RUN TestS1
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
=== RUN TestS2
--- PASS: TestS2 (0.00s)
=== RUN TestF1
--- PASS: TestF1 (0.00s)
=== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
```

```

    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments    0.005s

```

If you do not include the `-v` parameter, which produces richer output, you will get the following output:

```

$ go test testMe.go testMe_test.go
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments    0.005s

```

If you want to run a test multiple times in succession, you can use the `-count` option as follows:

```

$ go test testMe.go testMe_test.go -count 2
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
--- FAIL: TestS1 (0.00s)
    testMe_test.go:7: s1("123456789") != 9
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL
FAIL    command-line-arguments    0.005s

```

Should you wish to execute specific tests, you should use the `-run` command-line option, which accepts a regular expression and executes all tests that have a function name that matches the given regular expression:

```

$ go test testMe.go testMe_test.go -run='F2' -v
=== RUN TestF2
--- FAIL: TestF2 (0.00s)
    testMe_test.go:50: f2(1) != 1
    testMe_test.go:54: f2(2) != 1
    testMe_test.go:58: f2(10) != 55
FAIL

```

```
FAIL command-line-arguments 0.005s
$ go test testMe.go testMe_test.go -run='F1'
ok command-line-arguments (cached)
```

The last command verifies that the `go test` command used caching.



CAUTION: Software testing can only show the presence of one or more bugs, not the absence of bugs. This means that you can never be absolutely sure that your code has no bugs!

Test code coverage

In this section, you will learn how to find more information about the code coverage of your programs. There are times when seeing the code coverage of your programs can reveal issues and bugs with your code, so do not underestimate its usefulness.

The Go code of `codeCover.go` is as follows:

```
package codeCover

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}

func fibo2(n int) int {
    if n >= 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```

The implementation of `fibo2()` is erroneous because it returns 0 all the time. A result of this bug is that most of the Go code of `fibo2()` will never get executed.

The test code that can be found in `codeCover_test.go` is as follows:

```
package codeCover

import (
    "testing"
)

func TestFibo1(t *testing.T) {
    if fibo1(1) != 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}

func TestFibo2(t *testing.T) {
    if fibo2(0) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}

func TestFibo1_10(t *testing.T) {
    if fibo1(10) == 1 {
        t.Errorf("Error fibo1(1): %d\n", fibo1(1))
    }
}

func TestFibo2_10(t *testing.T) {
    if fibo2(10) != 0 {
        t.Errorf("Error fibo2(0): %d\n", fibo1(0))
    }
}
```

The implementations of these functions are pretty naive because their purpose is to illustrate the use of the code coverage tool and not the generation of test functions.

Now we are going to check the coverage of the previous code. The main way to do this is by executing `go test` with the `-cover` parameter:

```
$ go test -cover -v
=== RUN   TestFibo1
--- PASS: TestFibo1 (0.00s)
=== RUN   TestFibo2
--- PASS: TestFibo2 (0.00s)
=== RUN   TestFibo1_10
--- PASS: TestFibo1_10 (0.00s)
=== RUN   TestFibo2_10
--- PASS: TestFibo2_10 (0.00s)
```

```
PASS
coverage: 70.0% of statements
ok      _/Users/mtsouk/cover    0.005s
```

So, the code coverage is 70.0%, which means that there might be a problem somewhere. Notice that the `-v` flag is not required for `-cover` to work.

However, there is a variation of the main command for test coverage that can generate a coverage profile:

```
$ go test -coverprofile=coverage.out
PASS
coverage: 70.0% of statements
ok      _/Users/mtsouk/cover    0.005s
```

After generating that special file, you can analyze it as follows:

```
$ go tool cover -func=coverage.out
/Users/mtsouk/cover/codeCover.go:3: fibo1    100.0%
/Users/mtsouk/cover/codeCover.go:13: fibo2   40.0%
total: (statements)    70.0%
```

Additionally, you can use a web browser to analyze that code coverage file as follows:

```
$ go tool cover -html=coverage.out
```

In the browser window that automatically opens, you will see lines of code colored either in green or in red. Red-colored lines of code are not covered by tests, which means that you should either cover them with more test cases or that there is something wrong with the Go code that is being tested. One way or another, you should look into these lines of code and reveal the problem.

Lastly, you can save that HTML output as follows:

```
$ go tool cover -html=coverage.out -o output.html
```

Testing an HTTP server with a database backend

In this section, you are going to see how to test a database server, which in this case is going to be a **PostgreSQL** server that works with an HTTP server written in Go. This is a special situation because you will have to get your database data in order to be able to test the correctness of the HTTP server.

For the purposes of this section, I am going to show two Go files named `webServer.go` and `webServer_test.go`. Before executing the code, you will need to download the Go package that will help you to work with PostgreSQL from Go, which requires the execution of the following command:

```
$ go get github.com/lib/pq
```

The Go code of `webServer.go` is as follows:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}

func getData(w http.ResponseWriter, r *http.Request) {
    fmt.Printf("Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
        return
    }

    rows, err := db.Query("SELECT * FROM users")
    if err != nil {
```

```

        fmt.Fprintf(w, "<h3 align=\"center\">%s</h3>\n", err)
        return
    }
    defer rows.Close()

    for rows.Next() {
        var id int
        var firstName string
        var lastName string
        err = rows.Scan(&id, &firstName, &lastName)
        if err != nil {
            fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>\n", err)
            return
        }
        fmt.Fprintf(w, "<h3 align=\"center\">%d, %s, %s</h3>\n", id,
firstName, lastName)
    }

    err = rows.Err()
    if err != nil {
        fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", err)
        return
    }
}

func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
    fmt.Println("Using port number: ", PORT)
    http.HandleFunc("/time", timeHandler)
    http.HandleFunc("/getdata", getData)
    http.HandleFunc("/", myHandler)

    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

The Go code of `webServer_test.go` will be presented in six parts. The thing with testing a web server that gets its data from a database is that you will need to write lots of utility code to support the testing of the database.

The first part of `webServer_test.go` is as follows:

```
package main

import (
    "database/sql"
    "fmt"
    _ "github.com/lib/pq"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

The second part of `webServer_test.go` contains the following Go code:

```
func create_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
    }

    const query = `
        CREATE TABLE IF NOT EXISTS users (
            id SERIAL PRIMARY KEY,
            first_name TEXT,
            last_name TEXT
        )`

    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}
```

This function communicates with PostgreSQL and creates a table named `users` inside the database, which will be used for testing purposes only.

The third part of `webServer_test.go` is as follows:

```
func drop_table() {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

```

    }

    _, err = db.Exec("DROP TABLE IF EXISTS users")
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

func insert_record(query string) {
    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }
    _, err = db.Exec(query)
    if err != nil {
        fmt.Println(err)
        return
    }
    db.Close()
}

```

These are two helper functions. The first one deletes the table that is created by `create_table()`, whereas the second one inserts a record in PostgreSQL.

The fourth part of `webServer_test.go` is the following:

```

func Test_count(t *testing.T) {
    var count int
    create_table()

    insert_record("INSERT INTO users (first_name, last_name) VALUES
('Epifanios', 'Doe')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES
('Mihalis', 'Tsoukalos')")
    insert_record("INSERT INTO users (first_name, last_name) VALUES
('Mihalis', 'Unknown')")

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }
}

```

```
    row := db.QueryRow("SELECT COUNT(*) FROM users")
    err = row.Scan(&count)
    db.Close()

    if count != 3 {
        t.Errorf("Select query returned %d", count)
    }
    drop_table()
}
```

This is the first test function of the package and it operates in two stages. Firstly, it inserts three records into a database table. Secondly, it verifies that the database table has exactly three records in it.

The fifth part of `webServer_test.go` contains the following Go code:

```
func Test_queryDB(t *testing.T) {
    create_table()

    connStr := "user=postgres dbname=s2 sslmode=disable"
    db, err := sql.Open("postgres", connStr)
    if err != nil {
        fmt.Println(err)
        return
    }

    query := "INSERT INTO users (first_name, last_name) VALUES ('Random
Text', '123456')"
    insert_record(query)

    rows, err := db.Query(`SELECT * FROM users WHERE last_name=$1`,
`123456`)
    if err != nil {
        fmt.Println(err)
        return
    }
    var col1 int
    var col2 string
    var col3 string
    for rows.Next() {
        rows.Scan(&col1, &col2, &col3)
    }
    if col2 != "Random Text" {
        t.Errorf("first_name returned %s", col2)
    }

    if col3 != "123456" {
        t.Errorf("last_name returned %s", col3)
    }
}
```

```

    }

    db.Close()
    drop_table()
}

```

This is another test function that inserts a record into a database table and verifies that the data was written correctly.

The last part of `webServer_test.go` is as follows:

```

func Test_record(t *testing.T) {
    create_table()
    insert_record("INSERT INTO users (first_name, last_name) VALUES
('John', 'Doe')")

    req, err := http.NewRequest("GET", "/getdata", nil)
    if err != nil {
        fmt.Println(err)
        return
    }
    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(getData)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("Handler returned %v", status)
    }

    if rr.Body.String() != "<h3 align=\"center\">1, John, Doe</h3>\n" {
        t.Errorf("Wrong server response!")
    }
    drop_table()
}

```

The last test function of the package interacts with the web server and visits the `/getdata` URL. Then, it verifies that the return value is the expected one.

At this point, you should create a PostgreSQL database named `s2` because this is the one that is used by the testing code. This can be done as follows:

```

$ psql -p 5432 -h localhost -U postgres -c "CREATE DATABASE s2"
CREATE DATABASE

```

If everything is OK, you will get the following kind of output from `go test`:

```

$ go test webServer* -v

```



```

=== RUN   Test_count
--- PASS: Test_count (0.05s)
=== RUN   Test_queryDB
--- PASS: Test_queryDB (0.04s)
=== RUN   Test_record
Serving: /getdata
Served:
--- PASS: Test_record (0.04s)
PASS
ok      command-line-arguments    0.138s

```

Omitting the `-v` option will generate a shorter output:

```

$ go test webServer*
ok      command-line-arguments    0.160s

```

Notice that you should not start the web server before the `go test` command, as it is automatically started by `go test`.

If the PostgreSQL server is not running, the test will fail with the following error messages:

```

$ go test webServer* -v
=== RUN   Test_count
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- FAIL: Test_count (0.01s)
    webServer_test.go:85: Select query returned 0
=== RUN   Test_queryDB
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
--- PASS: Test_queryDB (0.00s)
=== RUN   Test_record
dial tcp [::1]:5432: connect: connection refused
dial tcp [::1]:5432: connect: connection refused
Serving: /getdata
Served:
dial tcp [::1]:5432: connect: connection refused
--- FAIL: Test_record (0.00s)
    webServer_test.go:145: Wrong server response!
FAIL
FAIL    command-line-arguments    0.024s

```

In Chapter 12, *The Foundations of Network Programming in Go*, you are going to see an example where you test the handlers and the HTTP code of an HTTP server written in Go.

The testing/quick package

The Go standard library offers the `testing/quick` package, which can be used for **black-box testing** and is somewhat related to the `QuickCheck` package found in the **Haskell** programming language – both packages implement utility functions to help you with black-box testing. Go can generate random values of built-in types. The `testing/quick` package allows you to use these random values for testing, which saves you from having to generate all these values on your own.

A small program, which is saved as `randomBuiltin.go` and illustrates how Go can generate random values, is the following:

```
package main

import (
    "fmt"
    "math/rand"
    "reflect"
    "testing/quick"
    "time"
)

func main() {
    type point3D struct {
        X, Y, Z int8
        S        float32
    }
    ran := rand.New(rand.NewSource(time.Now().Unix()))

    myValues := reflect.TypeOf(point3D{})
    x, _ := quick.Value(myValues, ran)
    fmt.Println(x)
}
```

First, we create a new random generator using `rand.New()` and then we use **reflection** to get information about the type of `point3D`. After that, we call `quick.Value()` from the `testing/quick` package with a type descriptor and a random number generator in order to put some random data into the `myValues` variable. Notice that in order to generate arbitrary values for structures, all the fields of the structure must be exported.

Executing `randomBuiltin.go` will generate the following kind of output:

```
$ go run randomBuiltin.go
{65 8 75 -3.3435536e+38}
$ go run randomBuiltin.go
{-38 33 36 3.2604468e+38}
```

Notice that if you decide to create random strings, you will end up having Unicode strings with strange characters.

Now that you know how to create random values for built-in types, let us continue with the `testing/quick` package. For the purposes of this subsection, we are going to use two Go source files named `quick.go` and `quick_test.go`.

The Go code of `quick.go` is as follows:

```
package main

import (
    "fmt"
)

func Add(x, y uint16) uint16 {
    var i uint16
    for i = 0; i < x; i++ {
        y++
    }
    return y
}

func main() {
    fmt.Println(Add(0, 0))
}
```

The `add()` function in `quick.go` implements the addition of unsigned integers (`uint16`) using a `for` loop and it is used to illustrate black-box testing using the `testing/quick` package.

The code of `quick_test.go` will be the following:

```
package main

import (
    "testing"
    "testing/quick"
)

var N = 1000000

func TestWithSystem(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == (b + a)
    }
}
```

```

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

func TestWithItself(t *testing.T) {
    condition := func(a, b uint16) bool {
        return Add(a, b) == Add(b, a)
    }

    err := quick.Check(condition, &quick.Config{MaxCount: N})
    if err != nil {
        t.Errorf("Error: %v", err)
    }
}

```

The two calls to `quick.Check()` automatically generate random numbers based on the signature of their first argument, which is a function defined earlier. There is no need to create these random input numbers on your own, which makes the code easy to read and write. The actual tests happen in the `condition` function in both cases.

Executing the tests will generate the following output:

```

$ go test -v quick*
=== RUN   TestWithSystem
--- PASS: TestWithSystem (8.36s)
=== RUN   TestWithItself
--- PASS: TestWithItself (17.41s)
PASS
ok      command-line-arguments    (cached)

```

If you do not want to use cached testing, you should execute `go test` as follows:

```

$ go test -v quick* -count=1
=== RUN   TestWithSystem
--- PASS: TestWithSystem (8.15s)
=== RUN   TestWithItself
--- PASS: TestWithItself (15.95s)
PASS
ok      command-line-arguments    24.104s

```



The idiomatic way to bypass test caching is to use `-count=1` with the `go test` command because `GOCACHE=off` no longer works with Go 1.12.



The error message you will get if you try to use `GOCACHE=off` is `build cache is disabled by GOCACHE=off, but required as of Go 1.12`. You can learn more about it by executing `go help testflag`.

What if testing takes too long or never finishes?

If the `go test` tool takes too much time to finish or for some reason it never ends, there is the `-timeout` parameter. In order to illustrate that, we are going to create a new Go program along with its tests.

The Go code of the `main` package, which is saved as `too_long.go`, is the following:

```
package main

import (
    "time"
)

func sleep_with_me() {
    time.Sleep(5 * time.Second)
}

func get_one() int {
    return 1
}

func get_two() int {
    return 2
}

func main() {
}
```

The testing functions, which are saved in `too_long_test.go`, will be as follows:

```
package main

import (
    "testing"
)

func Test_test_one(t *testing.T) {
    sleep_with_me()
    value := get_one()
}
```

```

    if value != 1 {
        t.Errorf("Function returned %v", value)
    }
    sleep_with_me()
}

func Test_test_two(t *testing.T) {
    sleep_with_me()
    value := get_two()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}

func Test_that_will_fail(t *testing.T) {
    value := get_one()
    if value != 2 {
        t.Errorf("Function returned %v", value)
    }
}

```

The `Test_that_will_fail()` test function will always fail, whereas the other two functions are correct but slow.

Executing `go test` with and without the `-timeout` parameter will generate the following kinds of output:

```

$ go test too_long* -v
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
--- PASS: Test_test_two (5.00s)
=== RUN   Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL    command-line-arguments    15.019s
$ go test too_long* -v -timeout 20s
=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
--- PASS: Test_test_two (5.01s)
=== RUN   Test_that_will_fail
--- FAIL: Test_that_will_fail (0.00s)
    too_long_test.go:27: Function returned 1
FAIL
FAIL    command-line-arguments    15.021s
$ go test too_long* -v -timeout 15s

```

```

=== RUN   Test_test_one
--- PASS: Test_test_one (10.01s)
=== RUN   Test_test_two
panic: test timed out after 15s
goroutine 34 [running]:
testing.(*M).startAlarm.func1()
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1334 +0xdf
created by time.goFunc
    /usr/local/Cellar/go/1.12.4/libexec/src/time/sleep.go:169 +0x44
goroutine 1 [chan receive]:
testing.(*T).Run(0xc0000dc000, 0x113a46d, 0xd, 0x1141a08, 0x1069b01)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:917 +0x381
testing.runTests.func1(0xc0000c0000)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1157 +0x78
testing.tRunner(0xc0000c0000, 0xc00009fe30)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
testing.runTests(0xc0000ba000, 0x1230280, 0x3, 0x3, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1155 +0x2a9
testing.(*M).Run(0xc0000a8000, 0x0)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:1072 +0x162
main.main()
    _testmain.go:46 +0x13e
goroutine 5 [runnable]:
runtime.goparkunlock(...)
    /usr/local/Cellar/go/1.12.4/libexec/src/runtime/proc.go:307
time.Sleep(0x12a05f200)
    /usr/local/Cellar/go/1.12.4/libexec/src/runtime/time.go:105 +0x159
command-line-arguments.sleep_with_me(...)
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch11/too_long.go:8
command-line-arguments.Test_test_two(0xc0000dc000)
    /Users/mtsouk/Desktop/mGo2nd/Mastering-Go-Second-
Edition/ch11/too_long_test.go:17 +0x31
testing.tRunner(0xc0000dc000, 0x1141a08)
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:865 +0xc0
created by testing.(*T).Run
    /usr/local/Cellar/go/1.12.4/libexec/src/testing/testing.go:916 +0x35a
FAIL    command-line-arguments    15.015s

```

Only the third go test command fails because it takes longer than 15 seconds to finish.

Benchmarking Go code

Benchmarking measures the performance of a function or program, allowing you to compare implementations and to understand the impact of changes you make to your code.

Using that information, you can easily reveal the part of the Go code that needs to be rewritten in order to improve its performance.



Never benchmark your Go code on a busy UNIX machine that is currently being used for other, more important, purposes unless you have a very good reason to do so! Otherwise, you will interfere with the benchmarking process and get inaccurate results.

Go follows certain conventions regarding benchmarking. The most important convention is that the name of a benchmark function must begin with `Benchmark`.

Once again, the `go test` subcommand is responsible for benchmarking a program. As a result, you still need to import the `testing` standard Go package and include benchmarking functions in Go files that end with `_test.go`.

A simple benchmarking example

In this section, I will show you a basic benchmarking example that will measure the performance of three algorithms that generate numbers belonging to the Fibonacci sequence. The good news is that such algorithms require lots of mathematical calculations, which makes them perfect candidates for benchmarking.

For the purposes of this section, I will create a new `main` package, which will be saved as `benchmarkMe.go` and presented in three parts.

The first part of `benchmarkMe.go` is as follows:

```
package main

import (
    "fmt"
)

func fibo1(n int) int {
    if n == 0 {
        return 0
    } else if n == 1 {
        return 1
    } else {
        return fibo1(n-1) + fibo1(n-2)
    }
}
```


The preceding code contains the implementation of the `fibonacci()` function, which uses recursion in order to calculate numbers of the Fibonacci sequence. Although the algorithm works fine, this is a relatively simple and somewhat slow approach.

The second code segment of `benchmarkMe.go` is shown in the following Go code:

```
func fibo2(n int) int {
    if n == 0 || n == 1 {
        return n
    }
    return fibo2(n-1) + fibo2(n-2)
}
```

In this part, you see the implementation of the `fibo2()` function, which is almost identical to the `fibonacci()` function that you saw earlier. However, it will be interesting to see whether a small code change – a single `if` statement as opposed to an `if else if` block – has any impact on the performance of the function.

The third code portion of `benchmarkMe.go` contains yet another implementation of a function that calculates numbers that belong to the Fibonacci sequence:

```
func fibo3(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

The `fibo3()` function presented here uses a totally new approach that requires a Go **map** and has a `for` loop. It remains to be seen whether this approach is indeed faster than the other two implementations. The algorithm presented in `fibo3()` will also be used in Chapter 13, *Network Programming – Building Your Own Servers and Clients*, where it will be explained in greater detail. As you will see in a while, choosing an efficient algorithm can save you a lot of trouble!

The remaining code of `benchmarkMe.go` follows:

```
func main() {
    fmt.Println(fibonacci(40))
}
```

```
    fmt.Println(fibo2(40))
    fmt.Println(fibo3(40))
}
```

Executing `benchmarkMe.go` will generate the following output:

```
$ go run benchmarkMe.go
102334155
102334155
102334155
```

The good news is that all three implementations returned the same number. Now it is time to add some benchmarks to `benchmarkMe.go` in order to understand the efficiency of each one of the three algorithms.

As the Go rules require, the version of `benchmarkMe.go` containing the benchmark functions is going to be saved as `benchmarkMe_test.go`. This program is presented in five parts. The first code segment of `benchmarkMe_test.go` contains the following Go code:

```
package main
import (
    "testing"
)

var result int

func benchmarkfibo1(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo1(n)
    }
    result = r
}
```

In the preceding code, you can see the implementation of a function with a name that begins with the `benchmark` string instead of the `Benchmark` string. As a result, this function will not run automatically because it begins with a lowercase `b` instead of an uppercase `B`.

The reason for storing the result of `fibo1(n)` in a variable named `r` and using another global variable named `result` afterward is tricky: this technique is used to prevent the compiler from performing any optimizations that will exclude the function that you want to measure from being executed because its results are never used. The same technique will be applied to the `benchmarkfibo2()` and `benchmarkfibo3()` functions, which will be presented next.

The second part of `benchmarkMe_test.go` is shown in the following Go code:

```
func benchmarkfibo2(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo2(n)
    }
    result = r
}

func benchmarkfibo3(b *testing.B, n int) {
    var r int
    for i := 0; i < b.N; i++ {
        r = fibo3(n)
    }
    result = r
}
```

The preceding code defines two more benchmark functions that will not run automatically because they begin with a lowercase `b` instead of an uppercase `B`.

Now, I will tell you a big secret: even if these three functions were named `BenchmarkFibo1()`, `BenchmarkFibo2()`, and `BenchmarkFibo3()`, they would not have been invoked automatically by the `go test` command because their signature is not `func(*testing.B)`. So, that is the reason for naming them with a lowercase `b`. However, there is nothing that prevents you from invoking them from other benchmark functions afterward, as you will see shortly.

The third part of `benchmarkMe_test.go` follows:

```
func Benchmark30fibo1(b *testing.B) {
    benchmarkfibo1(b, 30)
}
```

This is a correct benchmark function with the correct name and the correct signature, which means that it will be executed by `go tool`.

Notice that although `Benchmark30fibo1()` is a valid benchmark function name, `BenchmarkfiboIII()` is not because there is no uppercase letter or a number after the `Benchmark` string. This is very important because a benchmark function with an incorrect name will not get executed automatically. The same rule applies to test functions.

The fourth code segment of `benchmarkMe_test.go` contains the following Go code:

```
func Benchmark30fibo2(b *testing.B) {
    benchmarkfibo2(b, 30)
```

```

}

func Benchmark30fibo3(b *testing.B) {
    benchmarkfibo3(b, 30)
}

```

Both the `Benchmark30fibo2()` and `Benchmark30fibo3()` benchmark functions are similar to `Benchmark30fibo1()`.

The last part of `benchmarkMe_test.go` is as follows:

```

func Benchmark50fibo1(b *testing.B) {
    benchmarkfibo1(b, 50)
}
func Benchmark50fibo2(b *testing.B) {
    benchmarkfibo2(b, 50)
}
func Benchmark50fibo3(b *testing.B) {
    benchmarkfibo3(b, 50)
}

```

In this part, you see three additional benchmark functions that calculate the 50th number in the Fibonacci sequence.



Remember that each benchmark is executed for at least one second by default. If the benchmark function returns in a time that is less than one second, the value of `b.N` is increased, and the function is run again. The first time the value of `b.N` is one, then it becomes two, then five, then 10, then 20, then 50, and so on. This happens because the faster the function is, the more times you need to run it to get accurate results.

Executing `benchmarkMe_test.go` will generate the following output:

```

$ go test -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibo1-8      300      4494213 ns/op
Benchmark30fibo2-8      300      4463607 ns/op
Benchmark30fibo3-8     500000      2829 ns/op
Benchmark50fibo1-8       1     67272089954 ns/op
Benchmark50fibo2-8       1     67300080137 ns/op
Benchmark50fibo3-8     300000      4138 ns/op
PASS
ok      command-line-arguments 145.827s

```

There are two important points here: first, the value of the `-bench` parameter specifies the benchmark functions that are going to be executed. The `.` value used is a regular expression that matches all valid benchmark functions. The second point is that if you omit the `-bench` parameter, no benchmark function will be executed.

So, what does this output tell us? First of all, the `-8` at the end of each benchmark function (`Benchmark10fibonacci8`) signifies the number of goroutines used during its execution, which is essentially the value of the `GOMAXPROCS` environment variable. You will recall that we talked about the `GOMAXPROCS` environment variable back in Chapter 10, *Concurrency in Go – Advanced Topics*. Similarly, you can see the values of `GOOS` and `GOARCH`, which show the operating system and the architecture of your machine.

The second column in the output displays the number of times that the relevant function was executed. Faster functions are executed more times than slower functions. As an example, the `Benchmark30fibonacci3()` function was executed 500,000 times, while the `Benchmark50fibonacci2()` function was executed only once! The third column in the output shows the average time of each run.

As you can see, the `fibonacci1()` and `fibonacci2()` functions are really slow compared to the `fibonacci3()` function. Should you wish to include memory allocation statistics in the output, you can execute the following command:

```
$ go test -benchmem -bench=. benchmarkMe.go benchmarkMe_test.go
goos: darwin
goarch: amd64
Benchmark30fibonacci1-8 300      4413791 ns/op      0 B/op      0 allocs/op
Benchmark30fibonacci2-8 300      4430097 ns/op      0 B/op      0 allocs/op
Benchmark30fibonacci3-8 500000  2774 ns/op        2236 B/op    6 allocs/op
Benchmark50fibonacci1-8 1       71534648696 ns/op    0 B/op      0 allocs/op
Benchmark50fibonacci2-8 1       72551120174 ns/op    0 B/op      0 allocs/op
Benchmark50fibonacci3-8 300000  4612 ns/op        2481 B/op   10 allocs/op
PASS
ok      command-line-arguments  150.500s
```

The preceding output is similar to the one without the `-benchmem` command-line parameter, but it includes two more columns in its output. The fourth column shows the amount of memory that was allocated on average in each execution of the benchmark function. The fifth column shows the number of allocations used to allocate the memory value of the fourth column. So, `Benchmark50fibonacci3()` allocated 2,481 bytes in 10 allocations on average.

As you can understand, the `fib01()` and `fib02()` functions do not need any special kind of memory apart from what is expected, because neither of them use any kind of data structure, which is not the case with `fib03()`, which uses a map variable; hence the larger-than-zero values in both the fourth and fifth columns of the output of `Benchmark10fib03-8`.

Wrongly defined benchmark functions

Look at the Go code of the following benchmark function:

```
func BenchmarkFiboI(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fib01(i)
    }
}
```

The `BenchmarkFibo()` function has a valid name and the correct signature. The bad news, however, is that this benchmark function is wrong, and you will not get any output from it after executing the `go test` command.

The reason for this is that as the `b.N` value grows in the way described earlier, the runtime of the benchmark function will also increase because of the `for` loop. This fact prevents `BenchmarkFiboI()` from converging to a stable number, which prevents the function from completing and therefore returning.

For analogous reasons, the next benchmark function is also wrongly implemented:

```
func BenchmarkfiboII(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fib02(b.N)
    }
}
```

On the other hand, there is nothing wrong with the implementation of the following two benchmark functions:

```
func BenchmarkFiboIV(b *testing.B) {
    for i := 0; i < b.N; i++ {
        _ = fib03(10)
    }
}

func BenchmarkFiboIII(b *testing.B) {
```

```
    _ = fibo3(b.N)
}
```

Benchmarking buffered writing

In this section, we will explore how the size of the write buffer affects the performance of the entire writing operation using the Go code of `writingBU.go`, which will be presented in five parts.

The `writingBU.go` program generates dummy files with randomly generated data. The variables of the program are the size of the buffer and the size of the output file.

The first part of `writingBU.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "os"
    "strconv"
)

var BUFFERSIZE int
var FILESIZE int

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second code portion of `writingBU.go` contains the following Go code:

```
func createBuffer(buf *[]byte, count int) {
    *buf = make([]byte, count)
    if count == 0 {
        return
    }
    for i := 0; i < count; i++ {
        intByte := byte(random(0, 100))
        if len(*buf) > count {
            return
        }
        *buf = append(*buf, intByte)
    }
}
```

The third part of `writingBU.go` is shown in the following Go code:

```
func Create(dst string, b, f int) error {
    _, err := os.Stat(dst)
    if err == nil {
        return fmt.Errorf("File %s already exists.", dst)
    }
    destination, err := os.Create(dst)
    if err != nil {
        return err
    }
    defer destination.Close()
    if err != nil {
        panic(err)
    }
    buf := make([]byte, 0)
    for {
        createBuffer(&buf, b)
        buf = buf[:b]
        if _, err := destination.Write(buf); err != nil {
            return err
        }
        if f < 0 {
            break
        }
        f = f - len(buf)
    }
    return err
}
```

The `Create()` function does all of the work in the program, and it is this function that needs to be benchmarked.

Note that if the buffer size and the file size were not part of the signature of the `Create()` function, you would have a problem writing a benchmark function for `Create()` because you would be required to use the `BUFFERSIZE` and `FILESIZE` global variables, which are both initialized in the `main()` function of `writingBU.go`. This would be difficult to do in the `writingBU_test.go` file. This means that in order to create a benchmark for a function, you should think about it when you are writing your code.

The fourth code segment of `writingBU.go` is as follows:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need BUFFERSIZE FILESIZE!")
        return
    }
}
```



```

output := "/tmp/randomFile"
BUFSIZE, _ = strconv.Atoi(os.Args[1])
FILESIZE, _ = strconv.Atoi(os.Args[2])
err := Create(output, BUFSIZE, FILESIZE)
if err != nil {
    fmt.Println(err)
}

```

The remaining Go code of `writingBU.go` follows:

```

err = os.Remove(output)
if err != nil {
    fmt.Println(err)
}
}

```

Although the `os.Remove()` call that deletes the temporary file is inside of the `main()` function, which is not called by the benchmark functions, it is easy to call `os.Remove()` from the benchmark functions, so there is no problem here.

Executing `writingBU.go` twice on a macOS Mojave machine with an SSD hard disk using the `time(1)` utility to check the speed of the program generates the following output:

```

$ time go run writingBU.go 1 100000
real    0m1.193s
user    0m0.349s
sys     0m0.809s
$ time go run writingBU.go 10 100000
real    0m0.283s
user    0m0.195s
sys     0m0.228s

```

So, although it is obvious that the size of the write buffer plays a key role in the performance of the program, we need to be much more specific and accurate. Therefore, we will write benchmark functions that will be stored in `writingBU_test.go`.

The first part of `writingBU_test.go` is shown in the following Go code:

```

package main
import (
    "fmt"
    "os"
    "testing"
)
var ERR error

func benchmarkCreate(b *testing.B, buffer, filesize int) {

```

```

    var err error
    for i := 0; i < b.N; i++ {
        err = Create("/tmp/random", buffer, filesize)
    }
    ERR = err
    err = os.Remove("/tmp/random")
    if err != nil {
        fmt.Println(err)
    }
}

```

As you will recall, this is not a valid benchmark function.

The second code segment of `writingBU_test.go` is as follows:

```

func Benchmark1Create(b *testing.B) {
    benchmarkCreate(b, 1, 1000000)
}

func Benchmark2Create(b *testing.B) {
    benchmarkCreate(b, 2, 1000000)
}

```

The remaining code of `writingBU_test.go` is as follows:

```

func Benchmark4Create(b *testing.B) {
    benchmarkCreate(b, 4, 1000000)
}

func Benchmark10Create(b *testing.B) {
    benchmarkCreate(b, 10, 1000000)
}

func Benchmark1000Create(b *testing.B) {
    benchmarkCreate(b, 1000, 1000000)
}

```

Here, we created five benchmark functions that are going to check the performance of the `benchmarkCreate()` function, which checks the performance of the `Create()` function for various write buffer sizes.

Executing `go test` on both the `writingBU.go` and `writingBU_test.go` files will generate the following type of output:

```

$ go test -bench=. writingBU.go writingBU_test.go
goos: darwin
goarch: amd64
Benchmark1Create-8          1      6001864841 ns/op

```

```

Benchmark2Create-8      1      3063250578 ns/op
Benchmark4Create-8      1      1557464132 ns/op
Benchmark10Create-8    100000      11136 ns/op
Benchmark1000Create-8  200000       5532 ns/op
PASS
ok      command-line-arguments  21.847s

```

The following output also checks the memory allocations of the benchmark functions:

```

$ go test -bench=. writingBU.go writingBU_test.go -benchmem
goos: darwin
goarch: amd64
Benchmark1Create-8 1      6209493161 ns/op      16000840 B/op      2000017
allocs/op
Benchmark2Create-8 1      3177139645 ns/op      8000584 B/op      1000013
allocs/op
Benchmark4Create 1      1632772604 ns/op      4000424 B/op      500011
allocs/op
Benchmark10Create-8 100000      11238 ns/op      336 B/op      7 allocs/op
Benchmark1000Create-8 200000 5122 ns/op      303 B/op      5 allocs/op
PASS
ok      command-line-arguments  24.031s

```

It is now time to interpret the output of the two `go test` commands.

It is extremely obvious that using a write buffer with a size of 1 byte is totally inefficient and slows everything down. Additionally, such a buffer size requires too many memory operations, which slows down the program even more.

Using a write buffer with 2 bytes makes the entire program twice as fast, which is a good thing. However, it is still very slow. The same applies to a write buffer with a size of 4 bytes. Where things get much faster is when we decide to use a write buffer with a size of 10 bytes. Finally, the results show that using a write buffer with a size of 1,000 bytes does not make things 100 times faster than when using a buffer size of 10 bytes, which means that the sweet spot between speed and write buffer size is somewhere between these two buffer size values.

Finding unreachable Go code

Go code that cannot be executed is a logical error, and therefore it is pretty difficult to reveal it with developers or a normal execution of the Go compiler. Put simply, there is nothing wrong with unreachable code, apart from the fact that there is no way for this code to get executed.

Take a look at the following Go code, which is saved as `cannotReach.go`:

```
package main
import (
    "fmt"
)

func f1() int {
    fmt.Println("Entering f1()")
    return -10
    fmt.Println("Exiting f1()")
    return -1
}

func f2() int {
    if true {
        return 10
    }
    fmt.Println("Exiting f2()")
    return 0
}

func main() {
    fmt.Println(f1())
    fmt.Println("Exiting program...")
}
```

There is nothing syntactically incorrect with the Go code of `cannotReach.go`. As a result, you can execute `cannotReach.go` without getting any error messages from the compiler:

```
$ go run cannotReach.go
Entering f1()
-1
Exiting program...
```

Note that `f2()` is never used in the program. However, it is obvious that the following Go code of `f2()` never gets executed because the condition in the preceding `if` is always `true`:

```
    fmt.Println("Exiting f2()")
    return 0
```

So, what can you do about it? You can execute `go vet` as follows:

```
$ go vet cannotReach.go
# command-line-arguments
./cannotReach.go:10:2: unreachable code
```

The output tells us that there is unreachable code in line 10 of the program. Now let's remove the `return -10` statement from the `f1()` function and rerun `go vet`:

```
$ go vet cannotReach.go
```

Here, there are no new error messages despite the fact that there is still unreachable code in the `f2()` function. This means that `go vet` cannot catch every possible type of logical error.

Cross-compilation

Cross-compilation is the process of generating a binary executable file for a different architecture than the one on which you are working.

The main benefit that you receive from cross-compilation is that you do not need a second or third machine to create executable files for different architectures. This means that you basically need just a single machine for your development. Fortunately, Go has built-in support for cross-compilation.

For the purpose of this section, we are going to use the Go code of `xCompile.go` to illustrate the cross-compilation process. The Go code of `xCompile.go` is as follows:

```
package main

import (
    "fmt"
    "runtime"
)

func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("with Go version", runtime.Version())
}
```

Running `xCompile.go` on a macOS Mojave machine generates the following output:

```
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.12.4
```

In order to cross compile a Go source file, you will need to set the `GOOS` and `GOARCH` environment variables to the target operating system and architecture, respectively, which is not as difficult as it sounds.

So, the cross-compilation process goes like this:

```
$ env GOOS=linux GOARCH=arm go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
statically linked, not stripped
$ ./xCompile
-bash: ./xCompile: cannot execute binary file
```

The first command generates a binary file that works on Linux machines that use the ARM architecture, while the output of `file(1)` verifies that the generated binary file is indeed for a different architecture.

As the Debian Linux machine that will be used for this example has an Intel processor, we will have to execute the `go build` command once more using the correct `GOARCH` value:

```
$ env GOOS=linux GOARCH=386 go build xCompile.go
$ file xCompile
xCompile: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, with debug_info, not stripped
```

Executing the generated binary executable file on a Linux machine will produce the following expected output:

```
$ ./xCompile
You are using gc on a 386 machine
with Go version go1.12.4
$ go version
go version go1.3.3 linux/amd64
$ go run xCompile.go
You are using gc on a amd64 machine
with Go version go1.3.3
```

One thing to notice here is that the cross-compiled binary file of `xCompile.go` prints the Go version of the machine used for compiling it. The second thing to notice is that the architecture of the Linux machine is actually `amd64` instead of `386`, which was used in the cross-compilation process.



You can find a list of the available values for the GOOS and GOARCH environment variables at <https://golang.org/doc/install/source>. Keep in mind, however, that not all GOOS and GOARCH pairs are valid.

Creating example functions

Part of the documentation process is generating example code that showcases the use of some or all of the functions and types of a package. Example functions have many benefits, including the fact that they are executable tests that are executed by `go test`. Therefore, if an example function contains an `// Output :` line, the `go test` tool will check whether the calculated output matches the values found after the `// Output :` line.

Additionally, examples are really useful when seen in the documentation of the package, which is the subject of the next section. Finally, example functions that are presented on the Go documentation server (https://golang.org/pkg/io/#example_Copy) allow the reader of the documentation to experiment with the example code. The Go playground at <https://play.golang.org/> also supports this functionality.

As the `go test` subcommand is responsible for the examples of a program, you will need to import the testing standard Go package and include example functions in Go files that end with `_test.go`. Moreover, the name of each example function must begin with `Example`. Lastly, **example functions** take no input parameters and return no results.

Now let's create some example functions for the following package, which is saved as `ex.go`:

```
package ex
func F1(n int) int {
    if n == 0 {
        return 0
    }
    if n == 1 || n == 2 {
        return 1
    }
    return F1(n-1) + F1(n-2)
}
func S1(s string) int {
    return len(s)
}
```

The `ex.go` source file contains the implementation of two functions named `F1()` and `S1()`.

Notice that `ex.go` does not need to import the `fmt` package.

As you know, the example functions will be included in the `ex_test.go` file, which will be presented in three parts.

The first part of `ex_test.go` is as follows:

```
package ex
import (
    "fmt"
)
```

The second code portion of `ex_test.go` is shown in the following Go code:

```
func ExampleF1() {
    fmt.Println(F1(10))
    fmt.Println(F1(2))
    // Output:
    // 55
    // 1
}
```

The remaining Go code of `ex_test.go` follows:

```
func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 8
    // 0
}
```

Executing the `go test` command on the `ex.go` package will generate the following type of output:

```
$ go test ex.go ex_test.go -v
=== RUN ExampleF1
--- PASS: ExampleF1 (0.00s)
=== RUN ExampleS1
--- FAIL: ExampleS1 (0.00s)
got:
9
0
want:
8
0
FAIL
FAIL    command-line-arguments    0.006s
```


You will observe that the preceding output tells us that there is something wrong with the `S1()` function based on the data after the `// Output: comment`.

From Go code to machine code

In this section, you will learn how Go code is converted into machine code in much more detail. The Go program that is going to be used in this example is named `machineCode.go` and it is going to be presented in two parts.

The first part of `machineCode.go` is as follows:

```
package main

import (
    "fmt"
)

func hello() {
    fmt.Println("Hello!")
}
```

The second and final part of `machineCode.go` is as follows:

```
func main() {
    hello()
}
```

After that, you can see `machineCode.go` translated into machine code as follows:

```
$ GOSSAFUNC=main GOOS=linux GOARCH=amd64 go build -gcflags "-S"
machineCode.go
# runtime
dumped SSA to /usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html
# command-line-arguments
dumped SSA to ./ssa.html
os.(*File).close TEXT dupok nosplit size=26 args=0x18 locals=0x0
    0x0000 00000 (<autogenerated>:1) TEXT os.(*File).close(SB),
DUPOK|NOSPLIT|ABIInternal, $0-24
    0x0000 00000 (<autogenerated>:1) FUNCDATA $0, gcllocals
e6397a44f8e1b6e77d0f200b4fba5269(SB)
    0x0000 00000 (<autogenerated>:1) FUNCDATA $1, gcllocals
69c1753bd5f81501d95132d08af04464(SB)
    0x0000 00000 (<autogenerated>:1) FUNCDATA $3, gcllocals
9fb7f0986f647f17cb5
...
```

The first lines of the output tell us that there are two files that contain all the generated output: `/usr/local/Cellar/go/1.12.4/libexec/src/runtime/ssa.html` and `./ssa.html`. Notice that if you have a different Go version or a different Go installation, the first file will be located at a different place. **Static Single Assignment (SSA)** form is a method for describing low-level operations in a way that is pretty close to machine instructions. However, notice that SSA acts as if it has an infinite number of registers, which is not the case with machine code.

Opening that file on your favorite web browser will present a plethora of useful yet low-level information about the program. Lastly, notice that the parameter to `GOSSAFUNC` is the Go function that you want to **disassemble**.

Unfortunately, talking more about SSA is beyond the scope of this book.

Using assembly with Go

In this section, we are going to talk about **assembly** and Go, and how you can use assembly to implement Go functions. We will begin with the following Go program, which is saved as `add_me.go`:

```
package main

import (
    "fmt"
)

func add(x, y int64) int64 {
    return x + y
}

func main() {
    fmt.Println(add(1, 2))
}
```

Executing the following command will reveal the assembly implementation of the `add()` function, which is the following:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S add_me.go
"".add STEXT nosplit size=19 args=0x18 locals=0x0
    0x0000 00000 (add_me.go:7)    TEXT    "".add(SB), NOSPLIT|ABIInternal,
$0-24
    0x0000 00000 (add_me.go:7)    FUNCDATA    $0, gcllocals
33cdeccccebe80329f1fdbee7f5874cb(SB)
    0x0000 00000 (add_me.go:7)    FUNCDATA    $1, gcllocals
```

```

33cdecccccebe80329f1fdbee7f5874cb(SB)
0x0000 00000 (add_me.go:7)   FUNCDATA    $3, gcllocals
33cdecccccebe80329f1fdbee7f5874cb(SB)
0x0000 00000 (add_me.go:8)   PCDATA    $2, $0
0x0000 00000 (add_me.go:8)   PCDATA    $0, $0
0x0000 00000 (add_me.go:8)   MOVQ      ".y+16(SP), AX
0x0005 00005 (add_me.go:8)   MOVQ      ".x+8(SP), CX
0x000a 00010 (add_me.go:8)   ADDQ      CX, AX
0x000d 00013 (add_me.go:8)   MOVQ      AX, ".~r2+24(SP)
0x0012 00018 (add_me.go:8)   RET
0x0000 48 8b 44 24 10 48 8b 4c 24 08 48 01 c8 48 89 44
H.D$.H.L$.H..H.D
0x0010 24 18 c3                                     $..
".main TEXT size=150 args=0x0 locals=0x58

```

The last line is not part of the assembly implementation of the `add()` function. Additionally, the `FUNCDATA` lines have nothing to do with the assembly implementation of the function and are added by the Go compiler.

Now, we are going to make some changes to the previous assembly code and make it look as follows:

```

TEXT    add(SB), $0
    MOVQ x+0(FP), BX
    MOVQ y+8(FP), BP
    ADDQ BP, BX
    MOVQ BX, ret+16(FP)
    RET

```

The assembly code will be saved in a file named `add_amd64.s` in order to be used as the implementation of the `add()` function.

The new version of `add_me.go` will be as follows:

```

package main

import (
    "fmt"
)

func add(x, y int64) int64

func main() {
    fmt.Println(add(1, 2))
}

```

This means that `add_me.go` will use the assembly implementation of the `add()` function. Using the assembly implementation of the `add()` function is as simple as this:

```
$ go build
$ ls -l
total 4136
-rw-r--r--@ 1 mtsouk  staff      93 Apr 18 22:49 add_amd64.s
-rw-r--r--@ 1 mtsouk  staff     101 Apr 18 22:59 add_me.go
-rwxr-xr-x  1 mtsouk  staff 2108072 Apr 18 23:00 asm
$ ./asm
3
$ file asm
asm: Mach-O 64-bit executable x86_64
```

The only tricky point here is that the assembly code is not portable. Unfortunately, talking more about assembly and Go is beyond the scope of the book – you should look at the assembly reference of your own CPU and architecture.

Generating documentation

Go offers the `godoc` tool, which allows you to view the documentation of your packages – provided that you have included some extra information in your files.



The general advice is that you should try to document everything but leave out obvious things. Put simply, do not say, "Here, I create a new `int` variable." It would be better to state the use of that `int` variable!

Nevertheless, really good code does not usually need documentation.

The rule about writing documentation in Go is pretty simple and straightforward: in order to document something, you have to put one or more regular comment lines that start with `//` just before its declaration. This convention can be used to document functions, variables, constants, or even packages.

Additionally, you will notice that the first line of the documentation of a package of any size will appear in the package list of `godoc`, as happens in <https://golang.org/pkg/>. This means that your description should be pretty expressive and complete.

Keep in mind that comments that begin with `BUG(something)` will appear in the `Bugs` section of the documentation of a package, even if they do not precede a declaration.

If you are looking for such an example, you can visit the source code and the documentation page of the `bytes` package, which can be found at:

<https://golang.org/src/bytes/bytes.go> and <https://golang.org/pkg/bytes/>, respectively. Lastly, all comments that are not related to a top-level declaration are omitted from the output that is generated by the `godoc` utility.

Take a look at the following Go code, which is saved as `documentMe.go`:

```
// This package is for showcasing the documentation capabilities of Go
// It is a naive package!
package documentMe
// Pie is a global variable
// This is a silly comment!
const Pie = 3.1415912
// The S1() function finds the length of a string
// It iterates over the string using range
func S1(s string) int {
    if s == "" {
        return 0
    }
    n := 0
    for range s {
        n++
    }
    return n
}
// The F1() function returns the double value of its input integer
// A better function name would have been Double()!
func F1(n int) int {
    return 2 * n
}
```

As discussed in the previous section, we will need to create a `documentMe_test.go` file in order to develop example functions for it. The contents of `documentMe_test.go` follow:

```
package documentMe
import (
    "fmt"
)
func ExampleS1() {
    fmt.Println(S1("123456789"))
    fmt.Println(S1(""))
    // Output:
    // 9
    // 0
}
```

```
func ExampleF1() {  
    fmt.Println(F1(10))  
    fmt.Println(F1(2))  
    // Output:  
    // 1  
    // 55  
}
```

In order to be able to see the documentation of `documentMe.go`, you will need to install the package on your local machine as you learned back in Chapter 6, *What You Might Not Know About Go Packages and Functions*. This will require the execution of the following commands from your favorite UNIX shell:

```
$ mkdir ~/go/src/documentMe  
$ cp documentMe* ~/go/src/documentMe/  
$ ls -l ~/go/src/documentMe/  
total 16  
-rw-r--r--@ 1 mtsouk staff 542 Mar 6 21:11 documentMe.go  
-rw-r--r--@ 1 mtsouk staff 223 Mar 6 21:11 documentMe_test.go  
$ go install documentMe  
$ cd ~/go/pkg/darwin_amd64  
$ ls -l documentMe.a  
-rw-r--r-- 1 mtsouk staff 1626 Mar 6 21:11 documentMe.a
```

Next, you should execute the `godoc` utility as follows:

```
$ godoc -http=":8080"
```

Note that if the port is already in use and the user has root privileges, the error message you will get will be the following:

```
$ godoc -http=":22"  
2019/08/19 15:18:21 ListenAndServe :22: listen tcp :22: bind: address  
already in use
```

On the other hand, if the user does not have root privileges, the error message will be the following (even if the port is already in use):

```
$ godoc -http=":22"  
2019/03/06 21:03:05 ListenAndServe :22: listen tcp :22: bind: permission  
denied
```

After taking care of that, you will be able to browse the HTML documentation created using your favorite web browser. The URL that will take you to the documentation is `http://localhost:8080/pkg/`.

The following figure shows the root directory of the `godoc` server that we just started. There, you can see the `documentMe` package that you created in `documentMe.go` among the other Go packages:

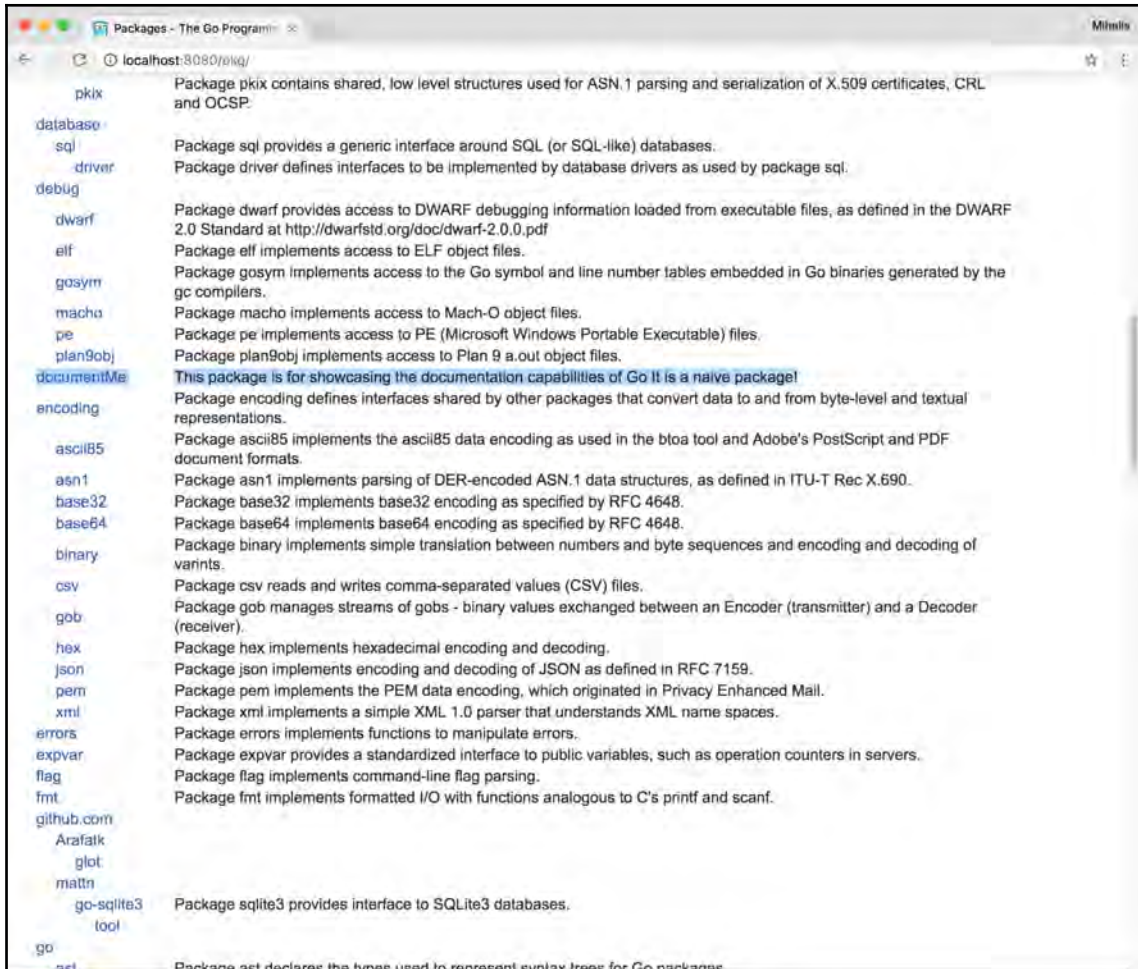


Figure 11.6: The root directory of the `godoc` server

The following figure shows the root directory of the documentation of the `documentMe` package implemented in the `documentMe.go` source file:

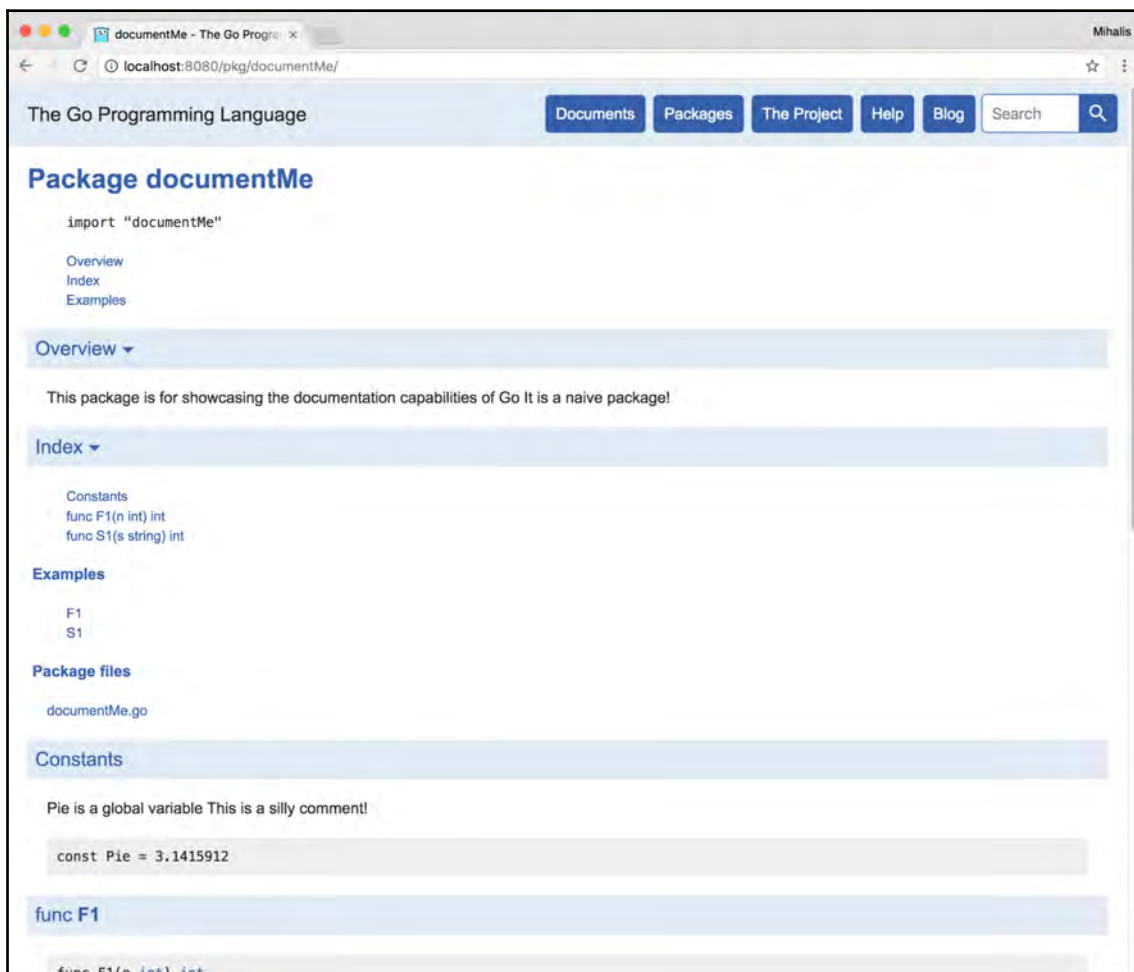


Figure 11.7: The root page of the `documentMe.go` file

Similarly, the following figure shows the documentation of the `S1()` function of the `documentMe.go` package in greater detail, which also includes the example code:

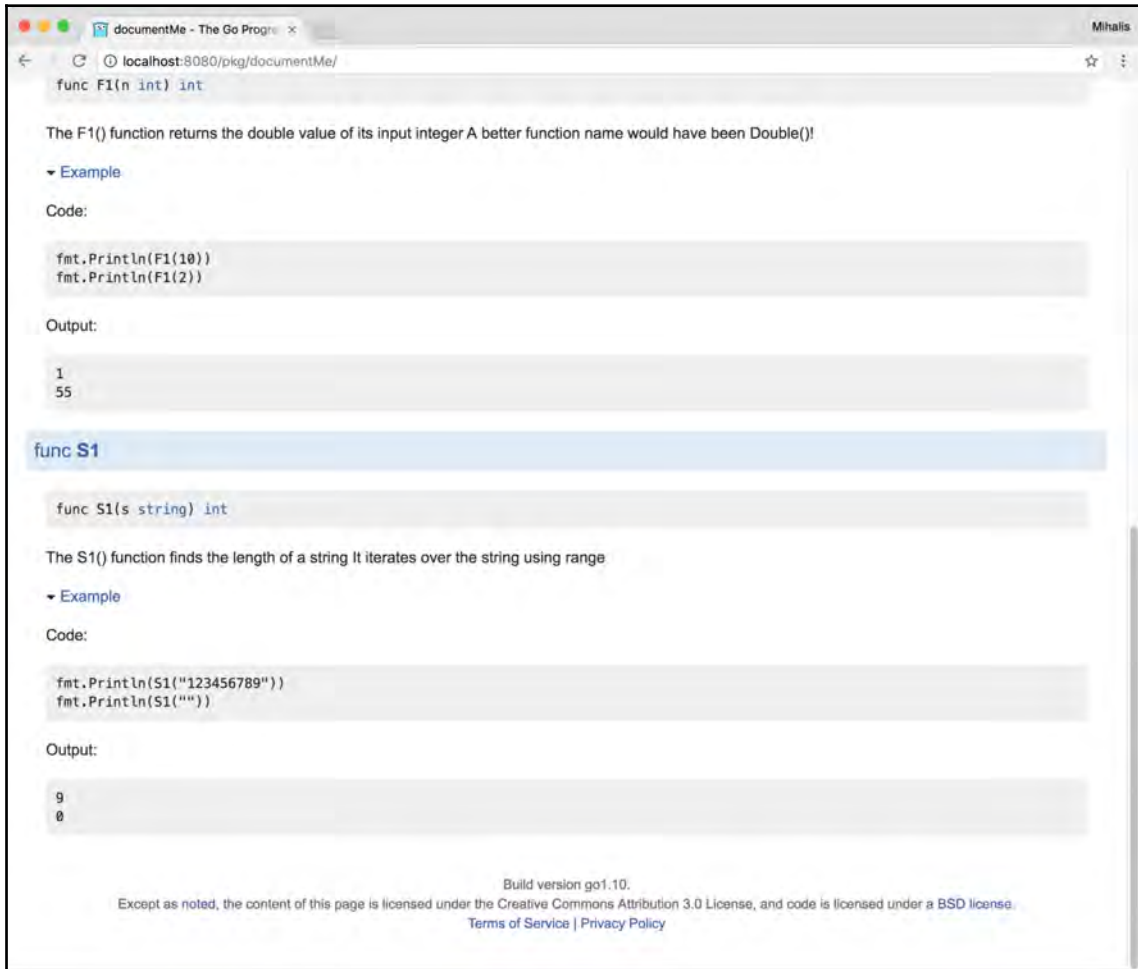


Figure 11.8: The documentation page and the example of the `S1()` function

Executing the `go test` command will generate the following output, which might reveal potential problems and bugs in our code:

```
$ go test -v documentMe*
=== RUN ExampleS1
--- PASS: ExampleS1 (0.00s)
=== RUN ExampleF1
```

```

--- FAIL: ExampleF1 (0.00s)
got:
20
4
want:
1
55
FAIL
FAIL    command-line-arguments    0.005s

```

Using Docker images

In this section, you will learn how to create a Docker image that contains both PostgreSQL and Go. The image will not be perfect, but it will illustrate how this can be done. As you already know, everything should begin with a `Dockerfile`, which in this case will be the following:

```

FROM ubuntu:18.04

RUN apt-get update && apt-get install -y gnupg
RUN apt-key adv --keyserver hkp://p80.pool.sks-keyservers.net:80 --recv-
keys B97B0AFCAA1A47F044F244A07FCC7D46ACCC4CF8
RUN echo "deb http://apt.postgresql.org/pub/repos/apt/ precise-pgdg main" >
/etc/apt/sources.list.d/pgdg.list
RUN apt-get update && apt-get install -y software-properties-common
postgresql-9.3 postgresql-client-9.3 postgresql-contrib-9.3
RUN apt-get update && apt-get install -y git golang vim

USER postgres
RUN /etc/init.d/postgresql start &&\
    psql --command "CREATE USER docker WITH SUPERUSER PASSWORD 'docker';"
&&\
    createdb -O docker docker

RUN echo "host all    all    0.0.0.0/0    md5" >>
/etc/postgresql/9.3/main/pg_hba.conf
RUN echo "listen_addresses='*'" >> /etc/postgresql/9.3/main/postgresql.conf

USER root
RUN mkdir files
COPY webServer.go files
WORKDIR files
RUN go get github.com/lib/pq
RUN go build webServer.go
RUN ls -l

```

```
USER postgres
CMD ["/usr/lib/postgresql/9.3/bin/postgres", "-D",
"/var/lib/postgresql/9.3/main", "-c",
"config_file=/etc/postgresql/9.3/main/postgresql.conf"]
```

This is a relatively complex `Dockerfile` that begins with a base Docker image from Docker Hub and downloads additional software – this is more or less the way to build a custom Docker image that fits your needs.

Executing `Dockerfile` will generate lots of output, including the following:

```
$ docker build -t go_postgres .
Sending build context to Docker daemon 9.216kB
Step 1/19 : FROM ubuntu:18.04
----> 94e814e2efa8
Step 2/19 : RUN apt-get update && apt-get install -y gnupg
...
Step 15/19 : RUN go get github.com/lib/pq
----> Running in 17aede1c97d8
Removing intermediate container 17aede1c97d8
----> 1878408c6e06
Step 16/19 : RUN go build webServer.go
----> Running in 39cfe8af63d5
Removing intermediate container 39cfe8af63d5
----> 1b4d638242ae
...
Removing intermediate container 9af6a391c1e4
----> c24937079367
Successfully built c24937079367
Successfully tagged go_postgres:latest
```

In the last part of the `Dockerfile`, we also build the executable file from `webServer.go` using the operating system of the Docker image, which is Ubuntu Linux.

Executing `docker images` will generate the following kind of output:

```
$ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
go_postgres	latest	c24937079367	34 seconds ago	831MB

Now we can execute that Docker image and connect to its `bash (1)` shell as follows:

```
$ docker run --name=my_go -dt go_postgres:latest
23de1ab0d3f5517d5dbf8c599a68075574a8ed9217aa3cb4899ea2f92412a833
$ docker exec -it my_go bash
postgres@23de1ab0d3f5:/files$
```

Additional resources

Visit the following web links:

- Visit the web site of Graphviz at <http://graphviz.org>.
- Visit the documentation page of the testing package, which can be found at <https://golang.org/pkg/testing/>.
- If you are not familiar with Donald Knuth and his work, you can learn more about him at https://en.wikipedia.org/wiki/Donald_Knuth.
- You can learn more about the godoc utility by visiting its documentation page at <https://godoc.org/golang.org/x/tools/cmd/godoc>.
- Visit the documentation page of the runtime/pprof standard Go package, which can be found at <https://golang.org/pkg/runtime/pprof/>.
- You can view the Go code of the net/http/pprof package by visiting <https://golang.org/src/net/http/pprof/pprof.go>.
- You can find the documentation page of the net/http/pprof package at <https://golang.org/pkg/net/http/pprof/>.
- You can learn more about the pprof tool by visiting its development page at <https://github.com/google/pprof>.
- Watch the "Advanced Testing with Go" video from GopherCon 2017 by Mitchell Hashimoto: <https://www.youtube.com/watch?v=8hQG7QlcLBk>.
- You can find the source code of the testing package at <https://golang.org/src/testing/testing.go>.
- You can find more information about the testing/quick package by visiting <https://golang.org/pkg/testing/quick/>.
- You can learn more about the profile package by visiting its web page at <https://github.com/pkg/profile>.
- A "Manual for the Plan 9 assembler": <https://9p.io/sys/doc/asm.html>.
- The documentation page of the arm64 package can be found at <https://golang.org/pkg/cmd/internal/obj/arm64/>.
- You can learn more about Go and SSA by visiting <https://golang.org/src/cmd/compile/internal/ssa/>.
- SSA bibliography: <http://www.dcs.gla.ac.uk/~jsinger/ssa.html>.
- You can learn more about the go fix tool by visiting its web page at <https://golang.org/cmd/fix/>.

Exercises

- Write test functions for the `byWord.go` program that we developed in Chapter 8, *Telling a UNIX System What to Do*.
- Write benchmark functions for the `readSize.go` program that we developed in Chapter 8, *Telling a UNIX System What to Do*.
- Try to fix the problems in the Go code of `documentMe.go` and `documentMe_test.go`.
- Use the text interface of the `go tool pprof` utility to examine the `memoryProfile.out` file generated by `profileMe.go`.
- Modify `webServer.go` and `webServer_test.go` in order to work with other databases such as MySQL and SQLite3.
- Modify the `Dockerfile` file from the last section in order to include `webServer_test.go`.
- Modify `Dockerfile` in order to be able to execute `go test -v` for the web server. The changes have to do with creating the right PostgreSQL users and databases.
- Use the web interface of the `go tool pprof` utility to examine the `memoryProfile.out` file generated by `profileMe.go`.

Summary

In this chapter, we talked about code testing, code optimization, and code profiling. Near the end of the chapter, you learned how to find unreachable code and how to cross-compile Go code. The `go test` command is used to test and benchmark Go code, as well as offering extra documentation with the use of example functions.

Although the discussion of the Go profiler and `go tool trace` is far from complete, you should understand that with topics such as profiling and code tracing, nothing can replace experimenting and trying new techniques on your own!

In the next chapter, Chapter 12, *The Foundations of Network Programming in Go*, we will start talking about network programming in Go, which involves programming applications that work over TCP/IP computer networks, which includes the Internet. Some of the subjects in the next chapter are the `net/http` package, creating web clients and web servers in Go, the `http.Response` and `http.Request` structures, profiling HTTP servers, gRPC, and timing out network connections.

Additionally, the next chapter will discuss the **IPv4** and **IPv6** protocols, as well as the Wireshark and tshark tools, which are used to capture and analyze network traffic.

12

The Foundations of Network Programming in Go

The previous chapter discussed benchmarking Go code using benchmark functions, testing in Go, example functions, code coverage, cross-compilation, and profiling Go code, as well as generating documentation in Go and creating Docker images that contain the software that you want.

This chapter is all about network and web programming, which means that you will learn how to create web applications that work over computer networks and the Internet. However, you will have to wait for *Chapter 13, Network Programming – Building Your Own Servers and Clients*, to learn how to develop TCP and UDP applications.

Please note that in order to follow this chapter and the next one successfully, you will need to know some basic information about HTTP, networking, and how computer networks work.

In this chapter, you will learn about the following topics:

- What TCP/IP is and why it is important
- The IPv4 and IPv6 protocols
- The **netcat** command-line utility
- Performing DNS lookups in Go
- The `net/http` package
- The `http.Response`, `http.Request`, and `http.Transport` structures
- Creating web servers in Go
- Programming web clients in Go
- Creating websites in Go

- **gRPC** and **Go**
- The `http.ServeMux` type
- **Wireshark** and **tshark**
- Timing out HTTP connections that take too long to finish either on the server or client end

About net/http, net, and http.RoundTripper

The star of this chapter will be the `net/http` package, which offers functions that allow you to develop powerful web servers and web clients. The `http.Set()` and `http.Get()` methods can be used to make HTTP and HTTPS requests, whereas the `http.ListenAndServe()` function can be used to create web servers by specifying the IP address and the TCP port to which the server will listen, as well as the functions that will handle incoming requests.

Apart from `net/http`, we will use the `net` package in some of the programs presented in this chapter. The functionality of the `net` package, however, will be exposed in greater detail in Chapter 13, *Network Programming – Building Your Own Servers and Clients*.

Finally, it will come in handy to know that an `http.RoundTripper` is an **interface** that is used to make sure that a Go element is capable of executing HTTP transactions. Put simply, this means that a Go element can get an `http.Response` for a given `http.Request`. You will learn about `http.Response` and `http.Request` shortly.

The http.Response type

The definition of the `http.Response` structure, which can be found in the <https://golang.org/src/net/http/response.go> file, is as follows:

```
type Response struct {  
    Status      string // e.g. "200 OK"  
    StatusCode  int   // e.g. 200  
    Proto       string // e.g. "HTTP/1.0"  
    ProtoMajor  int    // e.g. 1  
    ProtoMinor  int    // e.g. 0  
    Header      Header  
    Body         io.ReadCloser  
    ContentLength int64  
    TransferEncoding []string  
    Close        bool
```



```
    Uncompressed bool
    Trailer Header
    Request *Request
    TLS *tls.ConnectionState
}
```

The goal of this pretty complex `http.Response` type is to represent the response of an HTTP request. The source file contains more information about the purpose of each field of the structure, which is the case with most `struct` types found in the standard Go library.

The `http.Request` type

The purpose of the `http.Request` type is to represent an HTTP request as received by a server, or as about to be sent to a server by an HTTP client.

The `http.Request` structure type as defined in

<https://golang.org/src/net/http/request.go> is as follows:

```
type Request struct {
    Method string
    URL *url.URL
    Proto      string // "HTTP/1.0"
    ProtoMajor int    // 1
    ProtoMinor int    // 0
    Header Header
    Body io.ReadCloser
    GetBody func() (io.ReadCloser, error)
    ContentLength int64
    TransferEncoding []string
    Close bool
    Host string
    Form url.Values
    PostForm url.Values
    MultipartForm *multipart.Form
    Trailer Header
    RemoteAddr string
    RequestURI string
    TLS *tls.ConnectionState
    Cancel <-chan struct{}
    Response *Response
    ctx context.Context
}
```

The http.Transport type

The definition of the `http.Transport` structure, which can be found in <https://golang.org/src/net/http/transport.go>, is as follows:

```
type Transport struct {
    idleMu      sync.Mutex
    wantIdle    bool
    idleConn    map[connectMethodKey][]*persistConn
    idleConnCh  map[connectMethodKey]chan *persistConn
    idleLRU     connLRU
    reqMu       sync.Mutex
    reqCanceler map[*Request]func(error)
    altMu       sync.Mutex
    altProto    atomic.Value
    Proxy       func(*Request) (*url.URL, error)
    DialContext func(ctx context.Context, network, addr string)
              (net.Conn, error)
    Dial        func(network, addr string) (net.Conn, error)
    DialTLS     func(network, addr string) (net.Conn, error)
    TLSClientConfig *tls.Config
    TLSHandshakeTimeout time.Duration
    DisableKeepAlives bool
    DisableCompression bool
    MaxIdleConns int
    MaxIdleConnsPerHost int
    IdleConnTimeout time.Duration
    ResponseHeaderTimeout time.Duration
    ExpectContinueTimeout time.Duration
    TLSNextProto map[string]func(authority string, c *tls.Conn)
    RoundTripper
    ProxyConnectHeader Header
    MaxResponseHeaderBytes int64
    nextProtoOnce sync.Once
    h2transport    *http2Transport
}
```

As you can see, the `http.Transport` structure is pretty complex and contains a very large number of fields. The good news is that you will not need to use the `http.Transport` structure in all of your programs and that you are not required to deal with all of its fields each time that you use it.

The `http.Transport` structure implements the `http.RoundTripper` interface and supports HTTP, HTTPS, and HTTP proxies. Note that `http.Transport` is pretty low-level, whereas the `http.Client` structure, which is also used in this chapter, implements a high-level HTTP client.

About TCP/IP

TCP/IP is a family of protocols that help the internet to operate. Its name comes from its two most well-known protocols: **TCP** and **IP**.

TCP stands for **Transmission Control Protocol**. TCP software transmits data between machines using segments, which are also called **TCP packets**. The main characteristic of TCP is that it is a reliable protocol, which means that it makes sure that a packet was delivered without needing any extra code from the programmer. If there is no proof of packet delivery, TCP resends that particular packet. Among other things, a TCP packet can be used for establishing connections, transferring data, sending acknowledgements, and closing connections.

When a TCP connection is established between two machines, a full duplex virtual circuit, similar to a telephone call, is created between those two machines. The two machines constantly communicate to make sure that data is sent and received correctly. If the connection fails for some reason, the two machines try to find the problem and report to the relevant application.

IP stands for **Internet Protocol**. The main characteristic of IP is that it is not a reliable protocol by nature. IP encapsulates the data that travels over a TCP/IP network because it is responsible for delivering packets from the source host to the destination host according to the IP addresses. IP has to find an addressing method to send a packet to its destination effectively. Although there are dedicated devices, called routers, that perform IP routing, every TCP/IP device has to perform some basic routing.

The **UDP (User Datagram Protocol)** protocol is based on IP, which means that it is also unreliable. Generally speaking, the UDP protocol is simpler than the TCP protocol, mainly because UDP is not reliable by design. As a result, UDP messages can be lost, duplicated, or arrive out of order. Furthermore, packets can arrive faster than the recipient can process them. So, UDP is used when speed is more important than reliability.

About IPv4 and IPv6

The first version of the **IP protocol** is now called **IPv4** in order to differentiate it from the latest version of the IP protocol, which is called **IPv6**.

The main problem with IPv4 is that it is about to run out of IP addresses, which is the main reason for creating the IPv6 protocol. This happened because an IPv4 address is represented using 32 bits only, which allows a total number of 2^{32} (4,294,967,296) different IP addresses. On the other hand, IPv6 uses 128 bits to define each one of its addresses.

The format of an IPv4 address is 10.20.32.245 (four parts separated by dots), while the format of an IPv6 address is 3fce:1706:4523:3:150:f8ff:fe21:56cf (eight parts separated by colons).

The nc(1) command-line utility

The `nc(1)` utility, which is also called `netcat(1)`, comes in very handy when you want to test TCP/IP servers and clients. This section will present some of its more common uses.

You can use `nc(1)` as a client for a TCP service that runs on a machine with the 10.10.1.123 IP address and listens to port number 1234, as follows:

```
$ nc 10.10.1.123 1234
```

By default, `nc(1)` uses the TCP protocol. However, if you execute `nc(1)` with the `-u` flag, then `nc(1)` will use the UDP protocol.

The `-l` option tells `netcat(1)` to act as a server, which means that `netcat(1)` will start listening for connections at the given port number.

Finally, the `-v` and `-vv` options tell `netcat(1)` to generate verbose output, which can come in handy when you want to troubleshoot network connections.

Although `netcat(1)` can help you to test HTTP applications, it will be niftier in [Chapter 13, Network Programming – Building Your Own Servers and Clients](#), when we will develop our own TCP and UDP clients and servers. As a result, the `netcat(1)` utility will be used only once in this chapter.

Reading the configuration of network interfaces

There are four core elements in a network configuration: the IP address of the interface, the network mask of the interface, the DNS servers of the machine, and the default gateway or default router of the machine. However, there is a problem here: you cannot find every piece of information using native, portable Go code. This means that there is no portable way to discover the DNS configuration and the default gateway information of a UNIX machine.

As a result, in this section, you will learn how to read the configuration of the network interfaces of a UNIX machine with Go. For that purpose, I will present two portable utilities that allow you to find out information about your network interfaces.

The source code of the first utility, which is called `netConfig.go`, is presented in three parts. The first part of `netConfig.go` is shown in the following Go code:

```
package main

import (
    "fmt"
    "net"
)

func main() {
    interfaces, err := net.Interfaces()
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `net.Interfaces()` function returns all of the interfaces of the current machine as a slice with elements of the `net.Interface` type. This slice will be used shortly to acquire the desired information.

The second code portion of `netConfig.go` contains the following Go code:

```
for _, i := range interfaces {
    fmt.Printf("Interface: %v\n", i.Name)
    byName, err := net.InterfaceByName(i.Name)
    if err != nil {
        fmt.Println(err)
    }
}
```

In the preceding code, you visit each element of the slice with the `net.Interface` elements to retrieve the desired information.

The remaining Go code of `netConfig.go` is as follows:

```
addresses, err := byName.Addrs()
for k, v := range addresses {
    fmt.Printf("Interface Address #%v: %v\n", k, v.String())
}
fmt.Println()
}
```

Executing `netConfig.go` on a macOS Mojave machine with Go version 1.12.4 generates the following output:

```
$ go run netConfig.go
Interface: lo0
Interface Address #0 : 127.0.0.1/8
Interface Address #1 : ::1/128
Interface Address #2 : fe80::1/64
Interface: gif0
Interface: stf0
Interface: XHC20
Interface: en0
Interface Address #0 : fe80::1435:19cd:ece8:f532/64
Interface Address #1 : 10.67.93.23/24
Interface: p2p0
Interface: awdl0
Interface Address #0 : fe80::888:68ff:fe01:99c/64
Interface: en1
Interface: en2
Interface: bridge0
Interface: utun0
Interface Address #0 : fe80::7fd3:e1ba:a4b1:fe22/64
```

As you can see, the `netConfig.go` utility returns a rather large output, because today's computers tend to have lots of network interfaces, and the program supports both the IPv4 and IPv6 protocols.

Executing `netConfig.go` on a Debian Linux machine with Go version 1.7.4 will generate the following output:

```
$ go run netConfig.go
Interface: lo
Interface Address #0: 127.0.0.1/8
Interface Address #1: ::1/128
Interface: dummy0
Interface: eth0
Interface Address #0: 10.74.193.253/24
Interface Address #1: 2a01:7e00::f03c:91ff:fe69:1381/64
Interface Address #2: fe80::f03c:91ff:fe69:1381/64
Interface: teql0
Interface: tunl0
Interface: gre0
Interface: gretap0
Interface: erspan0
Interface: ip_vti0
Interface: ip6_vti0
Interface: sit0
```

```
Interface: ip6tnl0
```

```
Interface: ip6gre0
```

Please note that the main reason why a network interface may not have a network address is that it is down, which essentially means that it is not currently configured.



Not all of the listed interfaces have a real hardware network device attached to them. A more representative example is the `lo0` interface, which is a loopback device. The **loopback device** is a special, virtual network interface that is used by your computer in order to communicate with itself over a network.

The Go code for the next utility, `netCapabilities.go`, is also presented in three parts. The purpose of the `netCapabilities.go` utility is to reveal the capabilities of each network interface found on your UNIX system.

The `netCapabilities.go` utility uses the fields of the `net.Interface` structure, which is defined as follows:

```
type Interface struct {  
    Index      int  
    MTU        int  
    Name       string  
    HardwareAddr HardwareAddr  
    Flags      Flags  
}
```

The first part of the Go code for `netCapabilities.go` is as follows:

```
package main  
  
import (  
    "fmt"  
    "net"  
)
```

The second code portion of `netCapabilities.go` contains the following Go code:

```
func main() {  
    interfaces, err := net.Interfaces()  
  
    if err != nil {  
        fmt.Print(err)  
        return  
    }  
}
```

The last part of `netCapabilities.go` comes in the following Go code:

```
for _, i := range interfaces {
    fmt.Printf("Name: %v\n", i.Name)
    fmt.Println("Interface Flags:", i.Flags.String())
    fmt.Println("Interface MTU:", i.MTU)
    fmt.Println("Interface Hardware Address:", i.HardwareAddr)

    fmt.Println()
}
}
```

Running `netCapabilities.go` on a macOS Mojave machine will generate the following output:

```
$ go run netCapabilities.go
Name : lo0
Interface Flags: up|loopback|multicast
Interface MTU: 16384
Interface Hardware Address:
Name : gif0
Interface Flags: pointtopoint|multicast
Interface MTU: 1280
Interface Hardware Address:
Name : stf0
Interface Flags: 0
Interface MTU: 1280
Interface Hardware Address:
Name : XHC20
Interface Flags: 0
Interface MTU: 0
Interface Hardware Address:
Name : en0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: b8:e8:56:34:a1:c8
Name : p2p0
Interface Flags: up|broadcast|multicast
Interface MTU: 2304
Interface Hardware Address: 0a:e8:56:34:a1:c8
Name : awdl0
Interface Flags: up|broadcast|multicast
Interface MTU: 1484
Interface Hardware Address: 0a:88:68:01:09:9c
Name : en1
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0
```



```
Name : en2
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b1
Name : bridge0
Interface Flags: up|broadcast|multicast
Interface MTU: 1500
Interface Hardware Address: 72:00:00:9d:b2:b0
Name : utun0
Interface Flags: up|pointtopoint|multicast
Interface MTU: 2000
Interface Hardware Address:
```

Executing `netCapabilities.go` on a Debian Linux machine will generate similar output.

Finally, if you are really interested in finding out the default gateway of the machine, you can execute the `netstat -nr` command either externally or using `exec.Command()`, taking its output using a pipe or `exec.CombinedOutput()` and processing it as text using Go. This solution, however, is neither elegant nor perfect!

Performing DNS lookups

DNS stands for **Domain Name System**, which relates to the way an IP address is translated into a name, such as `packt.com`, and vice versa. The logic behind the `DNS.go` utility, which will be developed in this section, is pretty simple: if the given command-line argument is a valid IP address, the program will process it as an IP address; otherwise, it will assume that it is dealing with a hostname that needs to be translated into one or more IP addresses.

The code for the `DNS.go` utility will be presented in three parts. The first part of the program contains the following Go code:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func lookIP(address string) ([]string, error) {
    hosts, err := net.LookupAddr(address)
    if err != nil {
        return nil, err
    }
}
```

```
        return hosts, nil
    }

    func lookHostname(hostname string) ([]string, error) {
        IPs, err := net.LookupHost(hostname)
        if err != nil {
            return nil, err
        }
        return IPs, nil
    }
}
```

The `lookIP()` function gets an IP address as input and returns a list of names that match that IP address with the help of the `net.LookupAddr()` function.

On the other hand, the `lookHostname()` function gets a hostname as input and returns a list with the associated IP addresses using the `net.LookupHost()` function.

The second part of `DNS.go` is the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide an argument!")
        return
    }

    input := arguments[1]
    IPaddress := net.ParseIP(input)
```

The `net.ParseIP()` function parses a string as an IPv4 or an IPv6 address. If the IP address is not valid, `net.ParseIP()` returns `nil`.

The remaining Go code for the `DNS.go` utility is as follows:

```
    if IPaddress == nil {
        IPs, err := lookHostname(input)
        if err == nil {
            for _, singleIP := range IPs {
                fmt.Println(singleIP)
            }
        }
    } else {
        hosts, err := lookIP(input)
        if err == nil {
            for _, hostname := range hosts {
                fmt.Println(hostname)
            }
        }
    }
}
```

```
    }  
  }  
}
```

Executing `DNS.go` with various kinds of input will generate the following output:

```
$ go run DNS.go 127.0.0.1  
localhost  
$ go run DNS.go 192.168.1.1  
cisco  
$ go run DNS.go packtpub.com  
83.166.169.231  
$ go run DNS.go google.com  
2a00:1450:4001:816::200e  
216.58.210.14  
$ go run DNS.go www.google.com  
2a00:1450:4001:816::2004  
216.58.214.36  
$ go run DNS.go cnn.com  
2a04:4e42::323  
2a04:4e42:600::323  
2a04:4e42:400::323  
2a04:4e42:200::323  
151.101.193.67  
151.101.1.67  
151.101.129.67  
151.101.65.67
```

Please note that the output of the `go run DNS.go 192.168.1.1` command is taken from my `/etc/hosts` file, because the `cisco` hostname is an alias for the `192.168.1.1` IP address in my `/etc/hosts` file.

The output of the last command shows that, sometimes, a single hostname (`cnn.com`) might have multiple public IP addresses. Please pay special attention to the word *public* here, because although `www.google.com` has multiple IP addresses, it uses just a single public IP address (`216.58.214.36`).

Getting the NS records of a domain

A very popular DNS request has to do with finding out the **name servers** of a domain, which are stored in the **NS records** of that domain. This functionality will be illustrated in the code for `NSrecords.go`.

The code for `NSrecords.go` will be presented in two parts.

The first part of `NSrecords.go` is as follows:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
}
```

In this part, you check whether you have at least one command-line argument in order to have something with which you can work.

The remaining Go code for `NSrecords.go` is as follows:

```
    domain := arguments[1]
    NSs, err := net.LookupNS(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, NS := range NSs {
        fmt.Println(NS.Host)
    }
}
```

All the work is done by the `net.LookupNS()` function, which returns the NS records of a domain as a slice variable of the `net.NS` type. This is the reason for printing the `Host` field of each `net.NS` element of the slice. Executing `NSrecords.go` will generate the following type of output:

```
$ go run NSrecords.go mtsoukalos.eu
ns5.linode.com.
ns4.linode.com.
ns1.linode.com.
ns2.linode.com.
ns3.linode.com.
$ go run NSrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

You can verify the correctness of the preceding output with the help of the `host (1)` utility:

```
$ host -t ns www.mtsoukalos.eu
www.mtsoukalos.eu has no NS record
$ host -t ns mtsoukalos.eu
mtsoukalos.eu name server ns3.linode.com.
mtsoukalos.eu name server ns1.linode.com.
mtsoukalos.eu name server ns4.linode.com.
mtsoukalos.eu name server ns2.linode.com.
mtsoukalos.eu name server ns5.linode.com.
```

Getting the MX records of a domain

Another very popular DNS request has to do with getting the **MX records** of a domain. The MX records specify the mail servers of a domain. The code for the `MXrecords.go` utility will perform this task with Go. The first part of the `MXrecords.go` utility is shown in the following Go code:

```
package main

import (
    "fmt"
    "net"
    "os"
)

func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Need a domain name!")
        return
    }
}
```

The second part of `MXrecords.go` contains the following Go code:

```
    domain := arguments[1]
    MXs, err := net.LookupMX(domain)
    if err != nil {
        fmt.Println(err)
        return
    }

    for _, MX := range MXs {
        fmt.Println(MX.Host)
    }
}
```

The code for `MXrecords.go` works in a similar way to the code for `NXrecords.go`, presented in the previous section. The only difference is that `MXrecords.go` uses the `net.LookupMX()` function instead of the `net.LookupNS()` function.

Executing `MXrecords.go` will generate the following type of output:

```
$ go run MXrecords.go golang.com
aspmx.1.google.com.
alt3.aspmx.1.google.com.
alt1.aspmx.1.google.com.
alt2.aspmx.1.google.com.
$ go run MXrecords.go www.mtsoukalos.eu
lookup www.mtsoukalos.eu on 8.8.8.8:53: no such host
```

Once again, you can verify the validity of the preceding output with the help of the `host(1)` utility:

```
$ host -t mx golang.com
golang.com mail is handled by 2 alt3.aspmx.1.google.com.
golang.com mail is handled by 1 aspmx.1.google.com.
golang.com mail is handled by 2 alt1.aspmx.1.google.com.
golang.com mail is handled by 2 alt2.aspmx.1.google.com.
$ host -t mx www.mtsoukalos.eu
www.mtsoukalos.eu has no MX record
```

Creating a web server in Go

Go allows you to create web servers on your own using some of the functions of its standard library.



Although a web server programmed in Go can do many things efficiently and securely, if what you really need is a powerful web server that will support modules, multiple websites, and virtual hosts, then you would be better off using a web server such as **Apache**, **Nginx** or **Candy**, which is written in Go.

The name of the Go program for this example will be `www.go`, and it will be presented in five parts. The first part of `www.go` contains the expected `import` statements:

```
package main

import (
    "fmt"
    "net/http"
```

```
    "os"  
    "time"  
)
```

The `time` package is not necessary for a web server to operate. However, it is needed in this case because the server is going to send the time and date to its clients.

The second code segment of `www.go` is shown in the following Go code:

```
func myHandler(w http.ResponseWriter, r *http.Request) {  
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)  
    fmt.Printf("Served: %s\n", r.Host)  
}
```

This is the implementation of the first **handler function** of the program. A handler function serves one or more URLs depending on the configuration that is stated in the Go code – you can have as many handler functions as you want.

The third part of `www.go` contains the following Go code:

```
func timeHandler(w http.ResponseWriter, r *http.Request) {  
    t := time.Now().Format(time.RFC1123)  
    Body := "The current time is:"  
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)  
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)  
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)  
    fmt.Printf("Served time for: %s\n", r.Host)  
}
```

In the preceding Go code, you can see the implementation of the program's second handler function. This function generates dynamic content.

The fourth section of code for our web server deals with the command-line arguments and the definition of the supported URLs:

```
func main() {  
    PORT := ":8001"  
    arguments := os.Args  
    if len(arguments) == 1 {  
        fmt.Println("Using default port number: ", PORT)  
    } else {  
        PORT = ":" + arguments[1]  
    }  
  
    http.HandleFunc("/time", timeHandler)  
    http.HandleFunc("/", myHandler)
```

The `http.HandleFunc()` function associates a URL with a handler function. What is really important here is that all URLs apart from `/time` are served by the `myHandler()` function because its first argument, which is `/`, matches every URL not matched by another handler.

The last part of the `www.go` program is as follows:

```
    err := http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

You should start the web server with the help of the `http.ListenAndServe()` function using the desired port number.

Executing `www.go` and connecting to its web server will generate the following type of output:

```
$ go run www.go
Using default port number:  :8001
Served: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served time for: localhost:8001
Served: localhost:8001
Served: localhost:8001
Served: localhost:8001
```


Although the output of the program provides some handy material, I think that you would prefer to see the real output of the program using your favorite web browser. The next screenshot shows the output of the `myHandler()` function of our web server as displayed in **Google Chrome**:

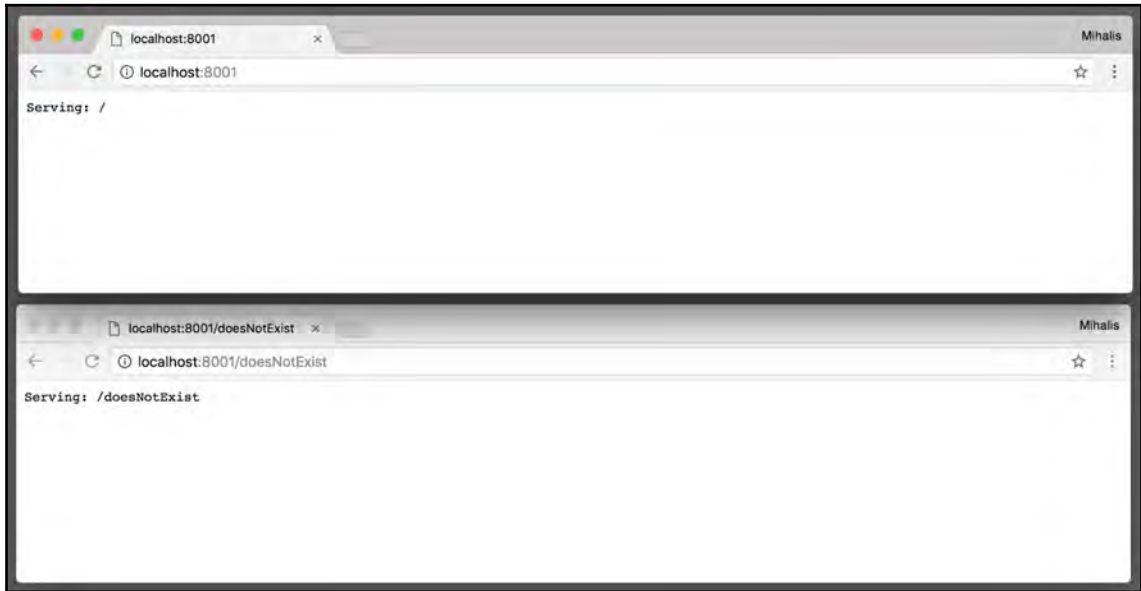


Figure 12.1: The home page of the `www.go` web server

The following screenshot shows that `www.go` can generate dynamic pages as well. In this case, it is a web page registered in `/time`, which displays the current date and time:

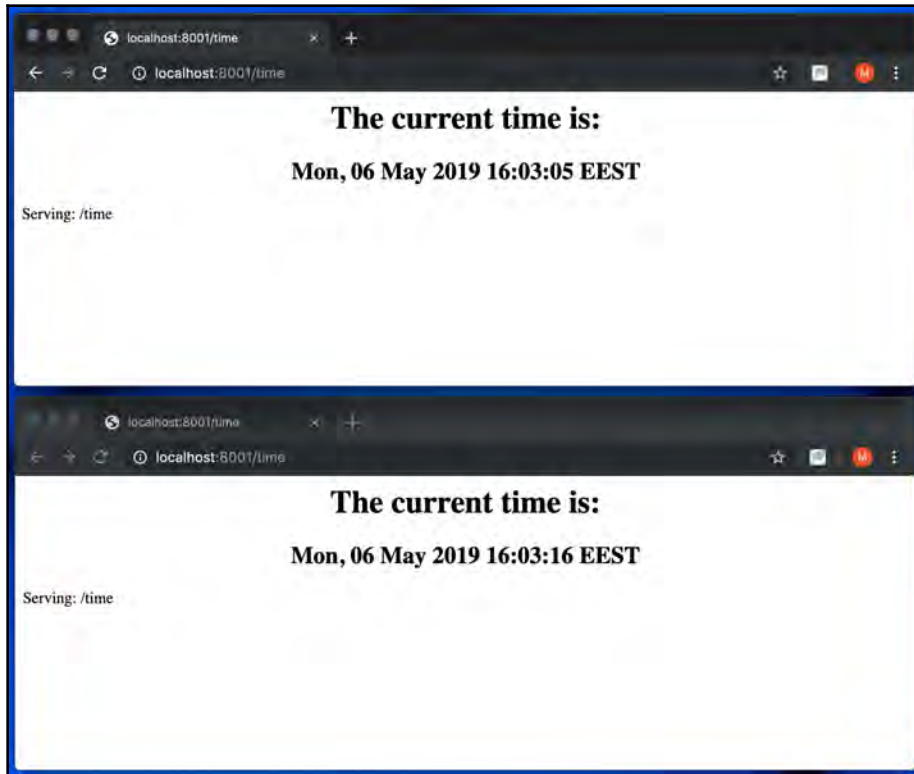


Figure 12.2: Getting the current date and time from the `www.go` web server

Using the `atomic` package

In this section, you will learn how to use the `atomic` package in an HTTP server environment. The name of the program is `atomWWW.go` and it will be presented in three parts.

The first part of `atomWWW.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
    "runtime"
    "sync/atomic"
)

var count int32
```

The variable that will be used by the `atomic` package is a global one, in order to be accessible from anywhere in the code.

The second part of `atomWWW.go` contains the following Go code:

```
func handleAll(w http.ResponseWriter, r *http.Request) {
    atomic.AddInt32(&count, 1)
}

func getCounter(w http.ResponseWriter, r *http.Request) {
    temp := atomic.LoadInt32(&count)
    fmt.Println("Count:", temp)
    fmt.Fprintf(w, "<h1 align=\"center\">%d</h1>", count)
}
```

The atomic counter used in this program is associated with the `count` global variable and helps us to count the total number of clients that the web server has served so far.

The last part of `atomWWW.go` is as follows:

```
func main() {
    runtime.GOMAXPROCS(runtime.NumCPU() - 1)
    http.HandleFunc("/getCounter", getCounter)
    http.HandleFunc("/", handleAll)
    http.ListenAndServe(":8080", nil)
}
```

Testing `atomWWW.go` using the `ab(1)` utility will verify how the `atomic` package helps, because it will reveal that the `count` variable can be accessed by all clients without any problems. First, you will need to execute `atomWWW.go`:

```
$ go run atomWWW.go
Count: 1500
```

Then, you will need to execute `ab(1)` as follows:

```
$ ab -n 1500 -c 100 http://localhost:8080/
This is ApacheBench, Version 2.3 <$Revision: 1826891 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking localhost (be patient)
Completed 150 requests
Completed 300 requests
Completed 450 requests
Completed 600 requests
Completed 750 requests
Completed 900 requests
Completed 1050 requests
Completed 1200 requests
Completed 1350 requests
Completed 1500 requests
Finished 1500 requests
Server Software:
Server Hostname:      localhost
Server Port:          8080
Document Path:        /
Document Length:      0 bytes
Concurrency Level:    100
Time taken for tests:  0.098 seconds
Complete requests:    1500
Failed requests:       0
Total transferred:    112500 bytes
HTML transferred:     0 bytes
Requests per second:  15238.64 [#/sec] (mean)
Time per request:     6.562 [ms] (mean)
Time per request:     0.066 [ms] (mean, across all concurrent requests)
Transfer rate:        1116.11 [Kbytes/sec] received

Connection Times (ms)
              min    mean[+/-sd] median    max
Connect:        0      3   0.5      3      5
Processing:      2      3   0.6      3      6
Waiting:         0      3   0.6      3      5
Total:           4      6   0.6      6      9
Percentage of the requests served within a certain time (ms)
50%             6
```

66%	6
75%	7
80%	7
90%	7
95%	7
98%	8
99%	8
100%	9 (longest request)

The previous `ab (1)` command sends 1,500 requests, with the number of concurrent requests being 100.

After the `ab (1)` part, you should visit the `/getCounter` address in order to get the current value of the `count` variable:

```
$ wget -qO- http://localhost:8080/getCounter
<h1 align="center">1500</h1>%
```

Profiling an HTTP server

As you learned in Chapter 11, *Code Testing, Optimization, and Profiling*, there is a standard Go package named `net/http/pprof`, which should be used when you want to profile a Go application with its own HTTP server. To that end, importing `net/http/pprof` will install various handlers under the `/debug/pprof/` URL. You will see more on this in a short while. For now, it is enough to remember that the `net/http/pprof` package should be used to profile web applications with an HTTP server, whereas the `runtime/pprof` standard Go package should be used to profile all other kinds of applications.

Note that if your profiler works using the `http://localhost:8080` address, you will automatically get support for the following web links:

- `http://localhost:8080/debug/pprof/goroutine`
- `http://localhost:8080/debug/pprof/heap`
- `http://localhost:8080/debug/pprof/threadcreate`
- `http://localhost:8080/debug/pprof/block`
- `http://localhost:8080/debug/pprof/mutex`
- `http://localhost:8080/debug/pprof/profile`
- `http://localhost:8080/debug/pprof/trace?seconds=5`

The next program to be presented uses `www.go` as its starting point and adds the necessary Go code to allow you to profile it.

The name of the new program is `wwwProfile.go`, and it will be presented in four parts.

Note that `wwwProfile.go` uses the `http.NewServeMux` variable to register the program's supported paths. The main reason for doing so is that the use of `http.NewServeMux` requires defining the HTTP endpoints manually. Also note that you are allowed to define a subset of the supported HTTP endpoints. If you decide not to use `http.NewServeMux`, then the HTTP endpoints will be registered automatically, which will also mean that you will have to import the `net/http/pprof` package using the `_` character in front of it.

The first part of `wwwProfile.go` contains the following Go code:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/pprof"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}

func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

The implementations of these two handler functions are exactly the same as before.

The second code segment of `wwwProfile.go` is as follows:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Println("Using port number: ", PORT)
    }
}
```

```
}

r := http.NewServeMux()
r.HandleFunc("/time", timeHandler)
r.HandleFunc("/", myHandler)
```

In the preceding Go code, you define the URLs that will be supported by your web server using `http.NewServeMux()` and `HandleFunc()`.

The third code portion for `wwwProfile.go` is shown in the following Go code:

```
r.HandleFunc("/debug/pprof/", pprof.Index)
r.HandleFunc("/debug/pprof/cmdline", pprof.Cmdline)
r.HandleFunc("/debug/pprof/profile", pprof.Profile)
r.HandleFunc("/debug/pprof/symbol", pprof.Symbol)
r.HandleFunc("/debug/pprof/trace", pprof.Trace)
```

The preceding Go code defines the HTTP endpoints related to profiling. Without them, you will not be able to profile your web application.

The remaining Go code is as follows:

```
err := http.ListenAndServe(PORT, r)
if err != nil {
    fmt.Println(err)
    return
}
}
```

This code begins the Go web server, and this allows it to serve connections from HTTP clients. You will notice that the second parameter to `http.ListenAndServe()` is no longer `nil`.

As you can see, `wwwProfile.go` does not define the `/debug/pprof/goroutine` HTTP endpoint, which makes perfect sense as `wwwProfile.go` does not use any goroutines.

Executing `wwwProfile.go` will generate the following type of output:

```
$ go run wwwProfile.go 1234
Using port number: 1234
Served time for: localhost:1234
```

Using the Go profiler to get data is a pretty simple task that requires the execution of the following command, which will take you to the shell of the Go profiler automatically:

```
$ go tool pprof http://localhost:1234/debug/pprof/profile
Fetching profile over HTTP from http://localhost:1234/debug/pprof/profile
```

```

Saved profile in /Users/mtsouk/pprof/pprof.samples.cpu.003.pb.gz
Type: cpu
Time: Mar 27, 2018 at 10:04pm (EEST)
Duration: 30s, Total samples = 21.04s (70.13%)
Entering interactive mode (type "help" for commands, "o" for options)
(pprof) top
Showing nodes accounting for 19.94s, 94.77% of 21.04s total
Dropped 159 nodes (cum <= 0.11s)
Showing top 10 nodes out of 75
      flat flat%   sum%        cum      cum%   syscall.Syscall
    13.73s  65.26%  65.26%    13.73s  65.26%   runtime.kevent
     1.58s   7.51%  72.77%     1.58s   7.51%   runtime.mach_semaphore_signal
     1.36s   6.46%  79.23%     1.36s   6.46%   runtime.usleep
     1.02s   4.85%  84.08%     1.02s   4.85%   runtime.mach_semaphore_wait
     0.80s   3.80%  87.88%     0.80s   3.80%   runtime.netpoll
     0.53s   2.52%  90.40%     2.11s  10.03%   internal/poll.convertErr
     0.44s   2.09%  92.49%     0.44s   2.09%   net.(*TCPConn).Read
     0.26s   1.24%  93.73%     0.26s   1.24%   runtime.freedefer
     0.18s   0.86%  94.58%     0.18s   0.86%   runtime.runqsteal
     0.04s   0.19%  94.77%     1.05s   4.99%
(pprof)

```

You can now use the profiling data and analyze it using `go tool pprof`, as you learned to do in Chapter 11, *Code Testing, Optimization, and Profiling*.



You can visit `http://HOSTNAME:PORTNUMBER/debug/pprof/` and see the profiling results there. When the `HOSTNAME` value is `localhost` and the `PORTNUMBER` value is `1234`, you should visit `http://localhost:1234/debug/pprof/`.

Should you wish to test the performance of your web server application, you can use the `ab(1)` utility, which is more widely known as the **Apache HTTP server benchmarking tool**, in order to create some traffic and benchmark `www.Profile.go`. This will also allow `go tool pprof` to collect more accurate data – you can execute `ab(1)` as follows:

```

$ ab -k -c 10 -n 100000 "http://127.0.0.1:1234/time"
This is ApacheBench, Version 2.3 <$Revision: 1807734 >
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/
Benchmarking 127.0.0.1 (be patient)
Completed 10000 requests
Completed 20000 requests
Completed 30000 requests
Completed 40000 requests
Completed 50000 requests
Completed 60000 requests
Completed 70000 requests

```



```

Completed 80000 requests
Completed 90000 requests
Completed 100000 requests
Finished 100000 requests

Server Software:
Server Hostname:      127.0.0.1
Server Port:          1234
Document Path:        /time
Document Length:      114 bytes
Concurrency Level:    10
Time taken for tests:  2.114 seconds
Complete requests:    100000
Failed requests:      0
Keep-Alive requests:  100000
Total transferred:    25500000 bytes
HTML transferred:     11400000 bytes
Requests per second:  47295.75 [#/sec] (mean)
Time per request:     0.211 [ms] (mean)
Time per request:     0.021 [ms] (mean, across all concurrent requests)
Transfer rate:        11777.75 [Kbytes/sec] received

Connection Times (ms)
      min    mean[+/-sd] median    max
Connect:    0      0   0.0      0      0
Processing:  0      0   0.7      0     13
Waiting:    0      0   0.7      0     13
Total:      0      0   0.7      0     13

Percentage of the requests served within a certain time (ms)
 50%      0
 66%      0
 75%      0
 80%      0
 90%      0
 95%      0
 98%      0
 99%      0
100%     13 (longest request)

```



Can you use the `net/http/pprof` package to profile command-line applications? Yes! However, the `net/http/pprof` package is particularly useful when you want to profile a running web application and capture live data, which is the main reason that it is presented in this chapter.

Creating a website in Go

Do you remember the `keyValue.go` application from Chapter 4, *The Uses of Composite Types*, and `kvSaveLoad.go` from Chapter 8, *Telling a UNIX System What to Do*? In this section, you will learn how to create a web interface for the `keyValue.go` application using the capabilities of the standard Go library. The name of the new Go source code file is `kvWeb.go`, and it is going to be presented in six parts.

The first difference between the Go code for `kvWeb.go` and `www.go`, developed earlier in this chapter, is that `kvWeb.go` uses the `http.NewServeMux` type to deal with HTTP requests because it is much more versatile for non-trivial web applications.

The first part of `kvWeb.go` is as follows:

```
package main

import (
    "encoding/gob"
    "fmt"
    "html/template"
    "net/http"
    "os"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

You have already seen this code in `kvSaveLoad.go` in Chapter 8, *Telling a UNIX System What to Do*.

The second part of `kvWeb.go` is shown in the following Go code:

```
func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
```

```
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
    if err != nil {
        fmt.Println("Cannot save to", DATAFILE)
        return err
    }
    return nil
}

func load() error {
    fmt.Println("Loading", DATAFILE)
    loadFrom, err := os.Open(DATAFILE)
    defer loadFrom.Close()
    if err != nil {
        fmt.Println("Empty key/value store!")
        return err
    }

    decoder := gob.NewDecoder(loadFrom)
    decoder.Decode(&DATA)
    return nil
}

func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}
```

```

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}

func PRINT() {
    for k, d := range DATA {
        fmt.Printf("key: %s value: %v\n", k, d)
    }
}

```

You should also be familiar with the preceding Go code, as it first appeared in `kvSaveLoad.go` in Chapter 8, *Telling a UNIX System What to Do*.

The third code portion of `kvWeb.go` is as follows:

```

func homePage(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Serving", r.Host, "for", r.URL.Path)
    myT := template.Must(template.ParseGlob("home.gohtml"))
    myT.ExecuteTemplate(w, "home.gohtml", nil)
}

func listAll(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Listing the contents of the KV store!")

    fmt.Fprintf(w, "<a href=\"/\" style=\"margin-right: 20px;\">Home sweet home!</a>")
    fmt.Fprintf(w, "<a href=\"/list\" style=\"margin-right: 20px;\">List all elements!</a>")
    fmt.Fprintf(w, "<a href=\"/change\" style=\"margin-right: 20px;\">Change an element!</a>")
    fmt.Fprintf(w, "<a href=\"/insert\" style=\"margin-right: 20px;\">Insert new element!</a>")

    fmt.Fprintf(w, "<h1>The contents of the KV store are:</h1>")
    fmt.Fprintf(w, "<ul>")
    for k, v := range DATA {
        fmt.Fprintf(w, "<li>")
    }
}

```

```

        fmt.Fprintf(w, "<strong>%s</strong> with value: %v\n", k, v)
        fmt.Fprintf(w, "</li>")
    }

    fmt.Fprintf(w, "</ul>")
}

```

The `listAll()` function does not use any Go templates to generate its dynamic output. Instead, its output is generated on the fly using Go. You may consider this an exception, as web applications usually work better with HTML templates and the `html/template` standard Go package.

The fourth part of `kvWeb.go` contains the following Go code:

```

func changeElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Changing an element of the KV store!")
    tmpl := template.Must(template.ParseFiles("update.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:    r.FormValue("name"),
        Surname: r.FormValue("surname"),
        Id:      r.FormValue("id"),
    }

    if !CHANGE(key, n) {
        fmt.Println("Update operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}

```

In the preceding Go code, you can see how to read the values from the fields of an HTML form with the help of the `FormValue()` function. `template.Must()` is a helper function that makes sure that the template file provided contains no errors.

The fifth code segment of `kvWeb.go` is contained in the following Go code:

```
func insertElement(w http.ResponseWriter, r *http.Request) {
    fmt.Println("Inserting an element to the KV store!")
    tmpl := template.Must(template.ParseFiles("insert.gohtml"))
    if r.Method != http.MethodPost {
        tmpl.Execute(w, nil)
        return
    }

    key := r.FormValue("key")
    n := myElement{
        Name:    r.FormValue("name"),
        Surname: r.FormValue("surname"),
        Id:       r.FormValue("id"),
    }

    if !ADD(key, n) {
        fmt.Println("Add operation failed!")
    } else {
        err := save()
        if err != nil {
            fmt.Println(err)
            return
        }
        tmpl.Execute(w, struct{ Success bool }{true})
    }
}
```

The remaining Go code is as follows:

```
func main() {
    err := load()
    if err != nil {
        fmt.Println(err)
    }

    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Using default port number: ", PORT)
    } else {
        PORT = ":" + arguments[1]
    }

    http.HandleFunc("/", homePage)
    http.HandleFunc("/change", changeElement)
    http.HandleFunc("/list", listAll)
```

```
    http.HandleFunc("/insert", insertElement)
    err = http.ListenAndServe(PORT, nil)
    if err != nil {
        fmt.Println(err)
    }
}
```

The `main()` function of `kvWeb.go` is much simpler than the `main()` function of `kvSaveLoad.go` from Chapter 8, *Telling a UNIX System What to Do*, because these two programs have totally different designs.

It is now time to look at the `gohtml` files used for this project, starting with `home.gohtml`, which is as follows:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

<h2>Welcome to the Go KV store!</h2>

</body>
</html>
```

The `home.gohtml` file is static, which means that its contents do not change. However, the remaining `gohtml` files display information dynamically.

The contents of `update.gohtml` are as follows:

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
```

```

<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element updated!</h1>
{{else}}
<h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br />
        <input type="text" name="key"><br />
        <label>Name:</label><br />
        <input type="text" name="name"><br />
        <label>Surname:</label><br />
        <input type="text" name="surname"><br />
        <label>Id:</label><br />
        <input type="text" name="id"><br />
        <input type="submit">
    </form>
{{end}}

</body>
</html>

```

The preceding code is mainly HTML code. Its most interesting part is the `if` statement, which specifies whether you should see the form or the `Element updated!` message.

Finally, the contents of `insert.gohtml` are as follows:

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>A Key Value Store!</title>
</head>
<body>

<a href="/" style="margin-right: 20px;">Home sweet home!</a>
<a href="/list" style="margin-right: 20px;">List all elements!</a>
<a href="/change" style="margin-right: 20px;">Change an element!</a>
<a href="/insert" style="margin-right: 20px;">Insert new element!</a>

{{if .Success}}
    <h1>Element inserted!</h1>
{{else}}
    <h1>Please fill in the fields:</h1>
    <form method="POST">
        <label>Key:</label><br />
        <input type="text" name="key"><br />

```



```
<label>Name:</label><br />
<input type="text" name="name"><br />
<label>Surname:</label><br />
<input type="text" name="surname"><br />
<label>Id:</label><br />
<input type="text" name="id"><br />
<input type="submit">
</form>
{{end}}

</body>
</html>
```

As you can see, `insert.gohtml` and `update.gohtml` are identical apart from the text in the `<title>` tag.

Executing `kvWeb.go` will generate the following output on a UNIX shell:

```
$ go run kvWeb.go
Loading /tmp/dataFile.gob
Using default port number: 8001
Serving localhost:8001 for /
Serving localhost:8001 for /favicon.ico
Listing the contents of the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Add operation failed!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Saving /tmp/dataFile.gob
Serving localhost:8001 for /favicon.ico
Inserting an element to the KV store!
Serving localhost:8001 for /favicon.ico
Changing an element of the KV store!
Serving localhost:8001 for /favicon.ico
```

Additionally, what is really interesting is how you can interact with `kvWeb.go` from a web browser.

The home page of the website, as defined in `home.gohtml`, can be seen in the following figure:

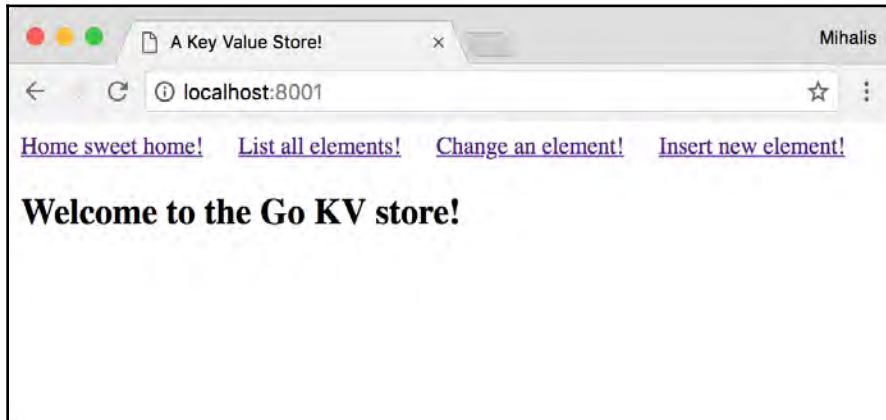


Figure 12.3: The static home page of our web application

The next figure presents the contents of the key-value store:

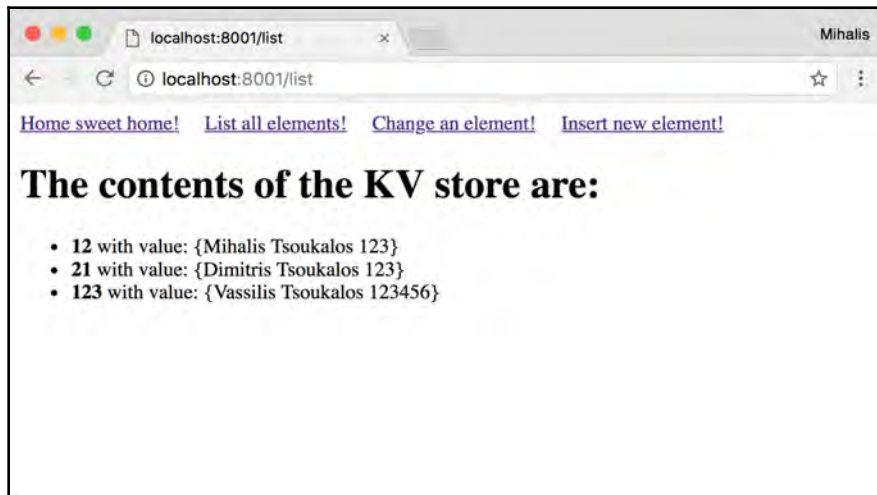


Figure 12.4: The contents of the key-value store

The next figure shows the appearance of the web page that allows you to add new data to the key-value store using the web interface of the `kvWeb.go` web application:

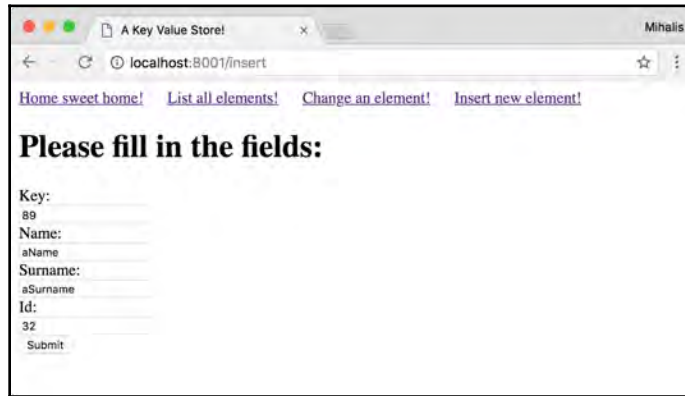


Figure 12.5: Adding a new entry to the key-value store

The next figure shows you how to update the value of an existing key using the web interface of the `kvWeb.go` web application:

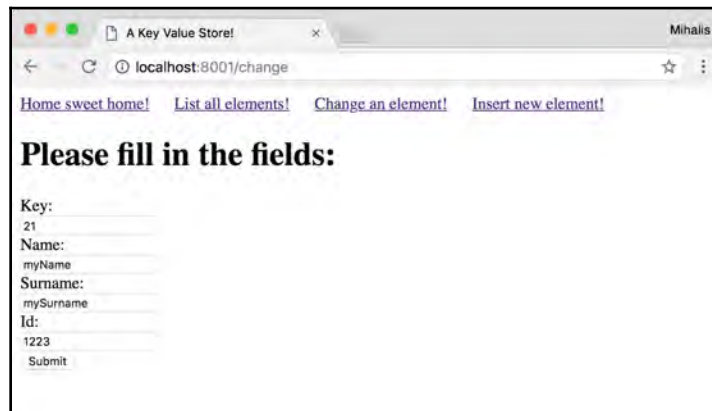


Figure 12.6: Updating the value of a key in the key-value store

The `kvWeb.go` web application is far from perfect, so feel free to improve it as an exercise!



This section illustrates how you can develop entire websites and web applications in Go. Although your requirements will undoubtedly vary, the techniques are the same as the ones used for `kvWeb.go`. Notice that custom-made sites are considered more secure than sites created using some popular content management systems.

HTTP tracing

Go supports **HTTP tracing** with the help of the `net/http/httptrace` standard package. That package allows you to trace the phases of an HTTP request. So, the use of the `net/http/httptrace` standard Go package will be illustrated in `httpTrace.go`, which is going to be presented in five parts.

The first part of `httpTrace.go` is as follows:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "net/http/httptrace"
    "os"
)
```

As you might expect, you need to import the `net/http/httptrace` package in order to enable HTTP tracing.

The second part of `httpTrace.go` contains the following Go code:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: URL\n")
        return
    }

    URL := os.Args[1]
    client := http.Client{}
```

In this part, we read the command-line arguments and create a new `http.Client` variable.

We will talk a little bit more about the `http.Client` object. The `http.Client` object offers a way to send a request to a server and get a reply.

Its `Transport` field permits you to set various HTTP details instead of using the default values.

Notice that you should never use the default values of the `http.Client` object in production software because those values do not specify request timeouts, which can jeopardize the performance of your programs and your goroutines. Additionally, by design, `http.Client` objects can be safely used in concurrent programs.

The third code segment of `httpTrace.go` is contained in the following Go code:

```
req, _ := http.NewRequest("GET", URL, nil)
trace := &httptrace.ClientTrace{
    GotFirstResponseByte: func() {
        fmt.Println("First response byte!")
    },
    GotConn: func(connInfo httptrace.GotConnInfo) {
        fmt.Printf("Got Conn: %+v\n", connInfo)
    },
    DNSDone: func(dnsInfo httptrace.DNSDoneInfo) {
        fmt.Printf("DNS Info: %+v\n", dnsInfo)
    },
    ConnectStart: func(network, addr string) {
        fmt.Println("Dial start")
    },
    ConnectDone: func(network, addr string, err error) {
        fmt.Println("Dial done")
    },
    WroteHeaders: func() {
        fmt.Println("Wrote headers")
    },
}
```

The preceding code is all about tracing HTTP requests. The `httptrace.ClientTrace` object defines the events that interest us. When such an event occurs, the relevant code is executed. You can find more information about supported events and their purpose in the documentation of the `net/http/httptrace` package.

The fourth part of the `httpTrace.go` utility is as follows:

```
req = req.WithContext(httptrace.WithClientTrace(req.Context(), trace))
fmt.Println("Requesting data from server!")
_, err := http.DefaultTransport.RoundTrip(req)
if err != nil {
    fmt.Println(err)
    return
}
```

The `httptrace.WithClientTrace()` function returns a new context based on the given parent context; while the `http.DefaultTransport.RoundTrip()` method wraps `http.DefaultTransport.RoundTrip` in order to tell it to keep track of the current request.

Notice that Go HTTP tracing has been designed to trace the events of a single `http.Transport.RoundTrip`. However, as you may have multiple URL redirects when serving a single HTTP request, you need to be able to identify the current request.

The remaining Go code for `httpTrace.go` is as follows:

```
    response, err := client.Do(req)
    if err != nil {
        fmt.Println(err)
        return
    }

    io.Copy(os.Stdout, response.Body)
}
```

This last part is about performing the actual request to the web server using `Do()`, getting the HTTP data, and displaying it on the screen.

Executing `httpTrace.go` will generate the following very informative output:

```
$ go run httpTrace.go http://localhost:8001/
Requesting data from server!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil>
Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142000 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
DNS Info: {Addrs:[{IP:::1 Zone:} {IP:127.0.0.1 Zone:}] Err:<nil>
Coalesced:false}
Dial start
Dial done
Got Conn: {Conn:0xc420142008 Reused:false WasIdle:false IdleTime:0s}
Wrote headers
First response byte!
Serving: /
```

Be aware that since `httpTrace.go` prints the full HTML response from the HTTP server, you might get lots of output when you test it on a real web server, which is the main reason for using the web server developed in `www.go` here.



If you have the time to look at the source code for the `net/http/httptrace` package at <https://golang.org/src/net/http/httptrace/trace.go>, you will immediately realize that `net/http/httptrace` is a pretty low-level package that uses the `context` package, the `reflect` package, and the `internal/nettrace` package to implement its functionality. Remember that you can do this for any code in the standard library because Go is a completely open-source project.

Testing HTTP handlers

In this section, we are going to learn how to test HTTP handlers in Go. We will begin with the Go code for `www.go` and modify it where needed.

The new version of `www.go` is called `testWWW.go`, and it will be presented in three parts. The first code portion of `testWWW.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
    "os"
)

func CheckStatusOK(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusOK)
    fmt.Fprintf(w, `Fine!`)
}
```

The second part of `testWWW.go` is contained in the following Go code:

```
func StatusNotFound(w http.ResponseWriter, r *http.Request) {
    w.WriteHeader(http.StatusNotFound)
}

func MyHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

The remaining Go code for `testWWW.go` is as follows:

```
func main() {
    PORT := ":8001"
```

```
arguments := os.Args
if len(arguments) == 1 {
    fmt.Println("Using default port number: ", PORT)
} else {
    PORT = ":" + arguments[1]
}

http.HandleFunc("/CheckStatusOK", CheckStatusOK)
http.HandleFunc("/StatusNotFound", StatusNotFound)
http.HandleFunc("/", MyHandler)

err := http.ListenAndServe(PORT, nil)
if err != nil {
    fmt.Println(err)
    return
}
}
```

We now need to start testing `testWWW.go`, which means that we should create a `testWWW_test.go` file. The contents of that file will be presented in four parts.

The first part of `testWWW_test.go` is contained in the following Go code:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httptest"
    "testing"
)
```

Notice that you need to import the `net/http/httptest` standard Go package in order to test web applications in Go.

The second code portion of `testWWW_test.go` contains the following code:

```
func TestCheckStatusOK(t *testing.T) {
    req, err := http.NewRequest("GET", "/CheckStatusOK", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(CheckStatusOK)
    handler.ServeHTTP(rr, req)
```


The `httptest.NewRecorder()` function returns an `httptest.ResponseRecorder` object, and it is used to record the HTTP response.

The third part of `testWWW_test.go` is as follows:

```
    status := rr.Code
    if status != http.StatusOK {
        t.Errorf("handler returned %v", status)
    }

    expect := `Fine!`
    if rr.Body.String() != expect {
        t.Errorf("handler returned %v", rr.Body.String())
    }
}
```

You first check that the response code is as expected, and then you verify that the body of the response is also what you expected.

The remaining Go code for `testWWW_test.go` is as follows:

```
func TestStatusNotFound(t *testing.T) {
    req, err := http.NewRequest("GET", "/StatusNotFound", nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    rr := httptest.NewRecorder()
    handler := http.HandlerFunc(StatusNotFound)
    handler.ServeHTTP(rr, req)

    status := rr.Code
    if status != http.StatusNotFound {
        t.Errorf("handler returned %v", status)
    }
}
```

This test function verifies that the `StatusNotFound()` function of the main package works as expected.

Executing the two test functions of `testWWW_test.go` will generate the following output:

```
$ go test testWWW.go testWWW_test.go -v --count=1
=== RUN   TestCheckStatusOK
--- PASS: TestCheckStatusOK (0.00s)
=== RUN   TestStatusNotFound
--- PASS: TestStatusNotFound (0.00s)
```

PASS
ok **command-line-arguments** (cached)

Creating a web client in Go

In this section, you will learn more about developing web clients in Go. The name of the web client utility is `webClient.go`, and it is going to be presented in four parts.

The first part of `webClient.go` contains the following Go code:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "path/filepath"
)
```

The second part of `webClient.go` is where you read the desired URL as a command-line argument:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL := os.Args[1]
```

The third code portion of `webClient.go` is where the real action takes place:

```
data, err := http.Get(URL)

if err != nil {
    fmt.Println(err)
    return
```

All of the work is done by the `http.Get()` call, which is pretty convenient when you do not want to deal with parameters and options. However, this type of call gives you no flexibility about the process. Notice that `http.Get()` returns an `http.Response` variable.

The remaining Go code is as follows:

```

    } else {
        defer data.Body.Close()
        _, err := io.Copy(os.Stdout, data.Body)
        if err != nil {
            fmt.Println(err)
            return
        }
    }
}

```

What the previous code does is copy the contents of the `Body` field of the `http.Response` structure to standard output.

Executing `webClient.go` will generate the following type of output (note that only a small portion of the output is presented here):

```

$ go run webClient.go http://www.mtsoukalos.eu/ | head -20
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML+RdFa 1.0//EN"
"http://www.w3.org/MarkUp/DTD/xhtml-rdFa-1.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
version="XHTML+RdFa 1.0" dir="ltr"
xmlns:content="http://purl.org/rss/1.0/modules/content/"
xmlns:dc="http://purl.org/dc/terms/"
xmlns:foaf="http://xmlns.com/foaf/0.1/"
xmlns:og="http://ogp.me/ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:sioc="http://rdfs.org/sioc/ns#"
xmlns:sioc="http://rdfs.org/sioc/types#"
xmlns:skos="http://www.w3.org/2004/02/skos/core#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#">
<head profile="http://www.w3.org/1999/xhtml/vocab">
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <link rel="shortcut icon" href="http://www.mtsoukalos.eu/misc/favicon.ico"
type="image/vnd.microsoft.icon" />
  <meta name="HandheldFriendly" content="true" />
  <meta name="MobileOptimized" content="width" />
  <meta name="Generator" content="Drupal 7 (http://drupal.org)" />

```

The main problem with `webClient.go` is that it gives you almost no control over the process – you either get the entire HTML output or nothing at all!

Making your Go web client more advanced

As the web client of the previous section is relatively simplistic and does not give you any flexibility, in this section, you will learn how to read a URL more elegantly without using the `http.Get()` function, and with more options. The name of the utility is `advancedWebClient.go` and it is going to be presented in five parts.

The first code segment of `advancedWebClient.go` contains the following Go code:

```
package main

import (
    "fmt"
    "net/http"
    "net/http/httputil"
    "net/url"
    "os"
    "path/filepath"
    "strings"
    "time"
)
```

The second part of `advancedWebClient.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }

    URL, err := url.Parse(os.Args[1])
    if err != nil {
        fmt.Println("Error in parsing:", err)
        return
    }
}
```

The third code portion of `advancedWebClient.go` contains the following Go code:

```
c := &http.Client{
    Timeout: 15 * time.Second,
}
request, err := http.NewRequest("GET", URL.String(), nil)
if err != nil {
    fmt.Println("Get:", err)
    return
}
```

```

httpData, err := c.Do(request)
if err != nil {
    fmt.Println("Error in Do():", err)
    return
}

```

The `http.NewRequest()` function returns an `http.Request` object given a method, a URL, and an optional body. The `http.Do()` function sends an HTTP request (`http.Request`) using an `http.Client` and gets an HTTP response (`http.Response`). So, `http.Do()` does the job of `http.Get()` in a more comprehensive way.

The "GET" string used in `http.NewRequest()` can be replaced by `http.MethodGet`.

The fourth part of `advancedWebClient.go` is contained in the following Go code:

```

fmt.Println("Status code:", httpData.Status)
header, _ := httputil.DumpResponse(httpData, false)
fmt.Print(string(header))

contentType := httpData.Header.Get("Content-Type")
characterSet := strings.SplitAfter(contentType, "charset=")
if len(characterSet) > 1 {
    fmt.Println("Character Set:", characterSet[1])
}

if httpData.ContentLength == -1 {
    fmt.Println("ContentLength is unknown!")
} else {
    fmt.Println("ContentLength:", httpData.ContentLength)
}

```

In the preceding code, you can see how to start searching the server response in order to find what you want.

The last part of the `advancedWebClient.go` utility is as follows:

```

length := 0
var buffer [1024]byte
r := httpData.Body
for {
    n, err := r.Read(buffer[0:])
    if err != nil {
        fmt.Println(err)
        break
    }
    length = length + n
}

```

```
    fmt.Println("Calculated response data length:", length)
}
```

In the preceding code, you can see a technique for discovering the size of the server HTTP response. If you want to display the HTML output on your screen, you can print the contents of the `r` buffer variable.

Using `advancedWebClient.go` to visit a web page will generate the following type of output, which is much richer than before:

```
$ go run advancedWebClient.go http://www.mtsoukalos.eu
Status code: 200 OK
HTTP/1.1 200 OK
Accept-Ranges: bytes
Age: 0
Cache-Control: no-cache, must-revalidate
Connection: keep-alive
Content-Language: en
Content-Type: text/html; charset=utf-8
Date: Sat, 24 Mar 2018 18:52:17 GMT
Expires: Sun, 19 Nov 1978 05:00:00 GMT
Server: Apache/2.4.25 (Debian) PHP/5.6.33-0+deb8u1 mod_wsgi/4.5.11
Python/2.7
Vary: Accept-Encoding
Via: 1.1 varnish (Varnish/5.0)
X-Content-Type-Options: nosniff
X-Frame-Options: SAMEORIGIN
X-Generator: Drupal 7 (http://drupal.org)
X-Powered-By: PHP/5.6.33-0+deb8u1
X-Varnish: 886025
Character Set: utf-8
ContentLength is unknown!
EOF
Calculated response data length: 50176
```

Executing `advancedWebClient.go` to visit a different URL will return a slightly different output:

```
$ go run advancedWebClient.go http://www.google.com
Status code: 200 OK
HTTP/1.1 200 OK
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-7
Date: Sat, 24 Mar 2018 18:52:38 GMT
Expires: -1
P3p: CP="This is not a P3P policy! See g.co/p3phelp for more info."
Server: gws
Set-Cookie: 1P_JAR=2018-03-24-18; expires=Mon, 23-Apr-2018 18:52:38 GMT;
```

```

path=/; domain=.google.gr
Set-Cookie:
NID=126=csX1_koD30SJcC_1jAfcM2V8kTfRkppmAamLjINLfcIracMxuk6JGe4gIc0Pjs8uD00
bqGaxkSW-J-ZNDJexG2ZX9pNB9E_dRc2y1KZ05V7pk0bOczE2FtS1zb50Uof1; expires=Sun,
23-Sep-2018 18:52:38 GMT; path=/; domain=.google.gr; HttpOnly
X-Frame-Options: SAMEORIGIN
X-Xss-Protection: 1; mode=block
Character Set: ISO-8859-7
ContentLength in unknown!
EOF
Calculated response data length: 10240

```

If you try to fetch an erroneous URL with `advancedWebClient.go`, you will get the following type of output:

```

$ go run advancedWebClient.go http://www.google
Error in Do(): Get http://www.google: dial tcp: lookup www.google: no such
host
$ go run advancedWebClient.go www.google.com
Error in Do(): Get www.google.com: unsupported protocol scheme ""

```

Feel free to make any changes you want to `advancedWebClient.go` in order to make the output match your requirements!

Timing out HTTP connections

This section will present a technique for timing out network connections that take too long to finish. Remember that you already know such a technique from *Chapter 10, Concurrency in Go – Advanced Topics*, when we talked about the `context` standard Go package. This technique was presented in the `useContext.go` source code file.

The method that is going to be presented in this section is much easier to implement. The relevant code is saved in `clientTimeout.go` and it is going to be presented in four parts. The utility accepts two command-line arguments, which are the URL and the timeout period in seconds. Please notice that the second parameter is optional.

The first part of the code portion for `clientTimeout.go` is as follows:

```

package main

import (
    "fmt"
    "io"
    "net"

```

```

    "net/http"
    "os"
    "path/filepath"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)

```

The second code segment for `clientTimeout.go` contains the following Go code:

```

func Timeout(network, host string) (net.Conn, error) {
    conn, err := net.DialTimeout(network, host, timeout)
    if err != nil {
        return nil, err
    }
    conn.SetDeadline(time.Now().Add(timeout))
    return conn, nil
}

```

You will learn more about the functionality of `SetDeadline()` in the next subsection. The `Timeout()` function will be used by the `Dial` field of an `http.Transport` variable.

The third code portion for `clientTimeout.go` is contained in the following code:

```

func main() {
    if len(os.Args) == 1 {
        fmt.Printf("Usage: %s URL TIMEOUT\n", filepath.Base(os.Args[0]))
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(temp) * time.Second
        }
    }

    URL := os.Args[1]
    t := http.Transport{
        Dial: Timeout,
    }

```

The remaining Go code for the `clientTimeout.go` utility is as follows:

```

client := http.Client{
    Transport: &t,
}

```



```
data, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
} else {
    defer data.Body.Close()
    _, err := io.Copy(os.Stdout, data.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `clientTimeout.go` web client is going to be tested using the `slowWWW.go` web server developed in Chapter 10, *Concurrency in Go – Advanced Topics*.

Executing `clientTimeout.go` twice will generate the following type of output:

```
$ go run clientTimeout.go http://localhost:8001
Serving: /
Delay: 0
$ go run clientTimeout.go http://localhost:8001
Get http://localhost:8001: read tcp [::1]:57397->[::1]:8001: i/o timeout
```

As you can see from the generated output, the first request had no problem connecting to the desired web server. However, the second `http.Get()` request took longer than expected and therefore timed out.

More information about SetDeadline

The `SetDeadline()` function is used by the `net` package to set the read and write deadlines of a given network connection. Due to the way the `SetDeadline()` function works, you will need to call `SetDeadline()` before any read or write operation. Keep in mind that Go uses deadlines to implement timeouts, so you do not need to reset the timeout every time your application receives or sends any data.

Setting the timeout period on the server side

In this subsection, you will learn how to time out a connection on the server side. You need to do this because there are times when clients take much longer than expected to end an HTTP connection. This usually happens for two reasons. The first reason is bugs in the client software, and the second reason is when a server process is experiencing an attack!

This technique is going to be implemented in the `serverTimeout.go` source code file, which will be presented in four parts. The first part of `serverTimeout.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
    "os"
    "time"
)

func myHandler(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served: %s\n", r.Host)
}
```

The second code portion of `serverTimeout.go` contains the following Go code:

```
func timeHandler(w http.ResponseWriter, r *http.Request) {
    t := time.Now().Format(time.RFC1123)
    Body := "The current time is:"
    fmt.Fprintf(w, "<h1 align=\"center\">%s</h1>", Body)
    fmt.Fprintf(w, "<h2 align=\"center\">%s</h2>\n", t)
    fmt.Fprintf(w, "Serving: %s\n", r.URL.Path)
    fmt.Printf("Served time for: %s\n", r.Host)
}
```

The third code segment of `serverTimeout.go` is shown in the following Go code:

```
func main() {
    PORT := ":8001"
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    } else {
        PORT = ":" + arguments[1]
        fmt.Printf("Listening on http://0.0.0.0%s\n", PORT)
    }

    m := http.NewServeMux()
    srv := &http.Server{
        Addr:      PORT,
        Handler:   m,
        ReadTimeout: 3 * time.Second,
        WriteTimeout: 3 * time.Second,
    }
}
```

In this case, we are using an `http.Server` structure that supports two kinds of timeouts with its fields. The first one is called `ReadTimeout`, and the second one is called `WriteTimeout`. The value of the `ReadTimeout` field specifies the maximum duration allowed to read the entire request, including the body.

The value of the `WriteTimeout` field specifies the maximum time duration before timing out the writing of the response. Put simply, this is the time from the end of the request header read to the end of the response write.

The remaining Go code for `serverTimeOut.go` is as follows:

```
m.HandleFunc("/time", timeHandler)
m.HandleFunc("/", myHandler)

err := srv.ListenAndServe()
if err != nil {
    fmt.Println(err)
    return
}
}
```

We are now going to execute `serverTimeOut.go` in order to interact with it using `nc(1)`:

```
$ go run serverTimeOut.go
Listening on http://0.0.0.0:8001
```

For the `nc(1)` part, which in this case acts as a dummy HTTP client, you should issue the following command to connect to `serverTimeOut.go`:

```
$ time nc localhost 8001
real    0m3.012s
user    0m0.001s
sys     0m0.002s
```

As we did not issue any commands, the HTTP server ended the connection. The output of the `time(1)` utility verifies the time it took the server to close the connection.

Yet another way to time out

This subsection will present yet another way to timeout an HTTP connection from the client side. As you will see, this is the simplest form of timeout, because you just need to use an `http.Client` object and set its `Timeout` field to the desired timeout value.

The name of the utility that showcases this last type of timeout is `anotherTimeout.go`, and it is going to be presented in four parts.

The first part of `anotherTimeout.go` is as follows:

```
package main

import (
    "fmt"
    "io"
    "net/http"
    "os"
    "strconv"
    "time"
)

var timeout = time.Duration(time.Second)
```

The second part of `anotherTimeout.go` contains the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please provide a URL")
        return
    }

    if len(os.Args) == 3 {
        temp, err := strconv.Atoi(os.Args[2])
        if err != nil {
            fmt.Println("Using Default Timeout!")
        } else {
            timeout = time.Duration(time.Duration(temp) * time.Second)
        }
    }

    URL := os.Args[1]
```

The third code segment for `anotherTimeout.go` contains the following Go code:

```
client := http.Client{
    Timeout: timeout,
}
client.Get(URL)
```

This is the place where the timeout period is defined using the `Timeout` field of the `http.Client` variable.

The last code portion for `anotherTimeout.go` is as follows:

```
data, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
} else {
    defer data.Body.Close()
    _, err := io.Copy(os.Stdout, data.Body)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Executing `anotherTimeout.go` and interacting with the `slowWWW.go` web server developed in Chapter 10, *Concurrency in Go – Advanced Topics*, will generate the following type of output:

```
$ go run anotherTimeout.go http://localhost:8001
Get http://localhost:8001: net/http: request canceled (Client.Timeout
exceeded while awaiting headers)
$ go run anotherTimeout.go http://localhost:8001 15
Serving: /
Delay: 8
```

The Wireshark and tshark tools

This section will briefly mention the powerful **Wireshark** and **tshark** utilities. Wireshark, which is a graphical application, is the dominant tool for analyzing network traffic of almost any kind. Although Wireshark is very powerful, there may be times when you need something lighter that you can execute remotely without a graphical user interface. In such situations, you can use tshark, which is the command-line version of Wireshark.

Unfortunately, talking more about Wireshark and tshark is beyond the scope of this chapter.

gRPC and Go

Strictly speaking, **gRPC** is a protocol built on HTTP/2 that allows you to create services easily. gRPC can use protocol buffers to specify an **interface definition language**, as well as to specify the format of the interchanged messages. gRPC clients and servers can be written in any programming language without the need to have clients written in the same programming language as their servers.

The process that will be presented here has three steps. The first step is creating the interface definition file, the second step is the development of the gRPC client, and the third step is the development of the gRPC server that will work with the gRPC client.

Defining the interface definition file

As you learned in the previous section, before we begin developing the gRPC client and server for our service, we need to define some data structures and protocols that will be used by them.

Protocol Buffers (protobuf) is a method for serializing structured data. As protobuf uses the binary format, it takes up less space than plain text serialization formats such as JSON and XML. However, it needs to be encoded and decoded in order to be machine-usable and human-readable, respectively.

As a result, in order to be able to use protobuf in your applications, you will need to download the necessary tools that allow you to work with it – the funny thing is that most protobuf tools are written in Go because Go is great at creating command-line tools!

On a macOS machine, the necessary tools can be downloaded using **Homebrew**, as follows:

```
$ brew install protobuf
```

There is an extra step that needs to be completed in order to get protobuf support for Go, because Go is not supported by default. This step requires the execution of the following command:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Among other things, the previous command downloads the `protoc-gen-go` executable and puts it in `~/go/bin`, which is the value of `$GOPATH/bin` on my machine. However, for the `protoc` compiler to find it, you will need to include that directory in your `PATH` environment variable, which on `bash`(1) and `zsh`(1) can be done as follows:

```
$ export PATH=$PATH:~/go/bin
```

Once you have the necessary tools, you will need to define the structures and the functions that will be used between the gRPC client and the gRPC server. In our case, the interface definition file, which is saved in `api.proto`, will be the following:

```
syntax = "proto3";

package message_service;

message Request {
    string text    = 1;
    string subtext = 2;
}

message Response {
    string text    = 1;
    string subtext = 2;
}

service MessageService {
    rpc SayIt (Request) returns (Response);
}
```

The gRPC server and the gRPC client that are going to be developed will support that protocol, which more or less defines a simple, basic messaging service with two basic types named `Request` and `Response`, and a single function named `SayIt()`.

However, we are not done yet, as `api.proto` needs to be processed and compiled by the `protobuf` tool, which in this case is the `protobuf` compiler that can be found at `/usr/local/bin/protoc`. This should be done as follows:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
```

After executing these commands, you will have an extra file on your computer named `api.pb.go`:

```
$ ls -l api.pb.go
-rw-r--r--  1 mtsouk  staff  7320 May  4 18:31 api.pb.go
```

So, for Go, the `protobuf` compiler generates `.pb.go` files that, among other things, contain a type for each message type in your interface definition file and a Go interface that needs to be implemented in the gRPC server. The `.pb.go` file extension will be different for other programming languages.

The first lines of `api.pb.go` are the following:

```
// Code generated by protoc-gen-go. DO NOT EDIT.
```

```
// source: api.proto

package message_service

import (
    context "context"
    fmt "fmt"
    proto "github.com/golang/protobuf/proto"
    grpc "google.golang.org/grpc"
    codes "google.golang.org/grpc/codes"
    status "google.golang.org/grpc/status"
    math "math"
)
```

These lines say that you should not edit `api.pb.go` on your own and that the name of the package is `message_service`. For your Go programs to be able to find all protobuf-related files, it is advisable to put them in their own GitHub repository. For the purposes of this example, that repository will be <https://github.com/mactsouk/protobuf>. This also means that you will need to get that repository using the following command:

```
$ go get github.com/mactsouk/protobuf
```



If you ever update `api.proto` or any other similar file on your local machine, you should remember to do two things: first, update the GitHub repository, and second, execute `go get -u -v` followed by the address of the remote GitHub repository to get the updates on your local machine.

Now we are ready to continue with developing the Go code for the gRPC client and server.

Note that if the required Go package is not present on your computer, you will get the following error message when you try to compile the interface definition file:

```
$ protoc -I . --go_out=plugins=grpc:. api.proto
protoc-gen-go: program not found or is not executable
--go_out: protoc-gen-go: Plugin failed with status code 1.
```

The gRPC client

In this subsection, we are going to develop a gRPC client in Go, which is going to be saved in the `gClient.go` file, which will be presented in three parts.

The first part of `gClient.go` is as follows:

```
package main
```



```
import (  
    "fmt"  
    p "github.com/mactsouk/protobuf"  
    "golang.org/x/net/context"  
    "google.golang.org/grpc"  
)  
  
var port = ":8080"
```

Notice that you should not need to download the required external Go packages because you already downloaded them when you executed the following command in order to compile the interface definition language file and create the Go-related output files:

```
$ go get -u github.com/golang/protobuf/protoc-gen-go
```

Keep in mind that `-u` tells `go get` to update the named packages along with their dependencies. If you use `-v`, you will get a better understanding of what actually happens, as this makes `go get` generate extra debugging information.

The second part of `gClient.go` contains the following Go code:

```
func AboutToSayIt(ctx context.Context, m p.MessageServiceClient, text  
string) (*p.Response, error) {  
    request := &p.Request{  
        Text:    text,  
        Subtext: "New Message!",  
    }  
    r, err := m.SayIt(ctx, request)  
    if err != nil {  
        return nil, err  
    }  
    return r, nil  
}
```

The way you name the `AboutToSayIt()` function is totally up to you. However, the signature of the function must contain a `context.Context` parameter, as well as a `MessageServiceClient` parameter in order to be able to call the `SayIt()` function later on. Notice that, for the client, you don't need to implement any functions of the interface definition language – you just have to call them.

The last part of `gClient.go` is as follows:

```
func main() {  
    conn, err := grpc.Dial(port, grpc.WithInsecure())  
    if err != nil {  
        fmt.Println("Dial:", err)  
        return  
    }  
}
```

```

    }

    client := p.NewMessageServiceClient(conn)
    r, err := AboutToSayIt(context.Background(), client, "My Message!")
    if err != nil {
        fmt.Println(err)
    }

    fmt.Println("Response Text:", r.Text)
    fmt.Println("Response SubText:", r.Subtext)
}

```

You need to call `grpc.Dial()` in order to connect to the gRPC server and create a new client using `NewMessageServiceClient()`. The name of the latter function depends on the value of the package statement found in the `api.proto` file. The message that is sent to the gRPC server is hardcoded in the code for `gClient.go`.

There is no point in executing the gRPC client without having a gRPC server, so you should wait for the next section. However, if you really want to find out the generated error message, you can still try:

```

$ go run gClient.go
rpc error: code = Unavailable desc = all SubConns are in TransientFailure,
latest connection error: connection error: desc = "transport: Error while
dialing dial tcp :8080: connect: connection refused"
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x8 pc=0x13d8afe]
goroutine 1 [running]:
main.main()
    /Users/mtsouk/ch12/gRPC/gClient.go:41 +0x22e
exit status 2

```

The gRPC server

In this subsection, you will learn how to develop a gRPC server in Go. The name of the program will be `gServer.go` and it is going to be presented in four parts.

The first part of `gServer.go` is as follows:

```

package main

import (
    "fmt"
    p "github.com/mactsouk/protobuf"
    "golang.org/x/net/context"

```

```

    "google.golang.org/grpc"
    "net"
)

```

As the name of the Go package with the interface definition language is `message_service`, which is pretty long, I've introduced an alias for it here, which is pretty short and convenient (p).

The `golang.org/x/net/context` and `google.golang.org/grpc` packages were downloaded earlier along with the other dependencies of the `github.com/golang/protobuf/protoc-gen-go` package, which means that you will not need to download any of them.



As of Go 1.7, the `golang.org/x/net/context` package is available in the standard library under the name `context`. Therefore, you are free to replace it with `context` if you want.

The second part of `gServer.go` contains the following Go code:

```

type MessageServer struct {
}

var port = ":8080"

```

You need that empty structure in order to be able to create the gRPC server later on in your Go code.

The third part of `gServer.go` is where you implement the interface:

```

func (MessageServer) SayIt(ctx context.Context, r *p.Request) (*p.Response,
error) {
    fmt.Println("Request Text:", r.Text)
    fmt.Println("Request SubText:", r.Subtext)

    response := &p.Response{
        Text:    r.Text,
        Subtext: "Got it!",
    }

    return response, nil
}

```

The signature of the `SayIt()` function depends on the data in the interface definition language file and can be found in `api.pb.go`.

What the `SayIt()` function does is send the contents of the `Text` field back to the client while changing the contents of the `Subtext` field.

The last part of `gServer.go` is as follows:

```
func main() {
    server := grpc.NewServer()
    var messageServer MessageServer
    p.RegisterMessageServiceServer(server, messageServer)
    listen, err := net.Listen("tcp", port)
    if err != nil {
        fmt.Println(err)
        return
    }
    fmt.Println("Serving requests...")
    server.Serve(listen)
}
```

In order to test the connection, you will need to begin by executing `gServer.go`:

```
$ go run gServer.go
Serving requests...
```

Executing `gClient.go` while `gServer.go` is running will generate the following kind of output:

```
$ go run gClient.go
Response Text: My Message!
Response SubText: Got it!
```

Once you execute `gClient.go`, you will see the following output in `gServer.go`:

```
Request Text: My Message!
Request SubText: New Message!
```

Although `gClient.go` will automatically end, you will need to manually terminate `gServer.go`.

Additional resources

The following resources will definitely broaden your horizons, so please find some time to read them:

- The official web page for the Apache web server is located at <http://httpd.apache.org/>.

- The official web page for the Nginx web server is located at <http://nginx.org/>.
- Should you wish to learn more about the Internet or TCP/IP, and its various services, you should start by reading the **RFC** documents. One of the places that you can find such documents is at <http://www.rfc-archive.org/>.
- Visit the website of both Wireshark and tshark at <https://www.wireshark.org/> to learn more about them.
- Visit the documentation page for the `net` standard Go package, which can be found at <https://golang.org/pkg/net/>. This is one of the largest documentation pages in the official Go documentation.
- Visit the documentation page for the `net/http` Go package at <https://golang.org/pkg/net/http/>.
- If you want to create a website without writing any Go code, you can try the **Hugo utility**, which is written in Go. You can learn more about the Hugo framework at <https://gohugo.io/>. However, what would really be interesting and educational for every Go programmer is to look at its Go code, which can be found at <https://github.com/gohugoio/hugo>.
- You can visit the documentation page for the `net/http/httptrace` package at <https://golang.org/pkg/net/http/httptrace/>.
- You can find the documentation page for the `net/http/pprof` package at <https://golang.org/pkg/net/http/pprof/>.
- Visit the manual page for the `nc(1)` command-line utility to learn more about its capabilities and its various command-line options.
- The `httpstat` utility, which was developed by Dave Cheney, can be found at <https://github.com/davecheney/httpstat>. It is a good example of the use of the `net/http/httptrace` Go package for HTTP tracing.
- You can find more information about the Candy web server at <https://github.com/caddyserver/cadd>.
- You can learn more about protobuf by visiting <https://opensource.google.com/projects/protobuf> and <https://developers.google.com/protocol-buffers/>.
- You can find the Protocol Buffers Language Guide documentation at <https://developers.google.com/protocol-buffers/docs/proto3>.
- The documentation page for the protobuf Go package can be found at <https://github.com/golang/protobuf>.
- You can find more information about `ab(1)` by visiting its manual page at <https://httpd.apache.org/docs/2.4/programs/ab.html>.

Exercises

- Write a web client in Go on your own without looking at the code in this chapter.
- Merge the code for `MXrecords.go` and `NSrecords.go` in order to create a single utility that does both jobs based on its command-line arguments.
- Modify the code for `MXrecords.go` and `NSrecords.go` to also accept IP addresses as input.
- Create a version of `MXrecords.go` and `NSrecords.go` using the `cobra` and `viper` packages.
- Create your own gRPC application using your own interface definition language.
- Make the necessary code changes to `gServer.go` in order to use goroutines and keep the number of clients that it has served.
- Modify `advancedWebClient.go` in order to save the HTML output in an external file.
- Try to implement a simple version of `ab(1)` on your own in Go using goroutines and channels.
- Modify `kvWeb.go` in order to support the `DELETE` and `LOOKUP` operations found in the original version of the key-value store.
- Modify `httpTrace.go` in order to have a flag that disables the execution of the `io.Copy(os.Stdout, response.Body)` statement.

Summary

This chapter presented Go code for programming web clients and web servers, as well as creating a website in Go. You also learned about the `http.Response`, `http.Request`, and `http.Transport` structures, which allow you to define the parameters of an HTTP connection.

Additionally, you learned how to develop gRPC applications, how to get the network configuration of a UNIX machine using Go code, and how to perform DNS lookups in a Go program, including getting the NS and MX records of a domain.

Finally, I talked about Wireshark and `tshark`, which are two very popular utilities that allow you to capture and, most importantly, analyze network traffic. At the beginning of this chapter, I also mentioned the `nc(1)` utility.

In the next chapter of this book, we will continue our discussion of network programming in Go. However, this time, I will present lower-level Go code that will allow you to develop TCP clients and servers, as well as UDP client and server processes. Additionally, you will learn about creating RCP clients and servers.

13

Network Programming – Building Your Own Servers and Clients

The previous chapter discussed topics related to network programming, including developing HTTP clients, HTTP servers, and web applications; performing DNS lookups; and timing out HTTP connections.

This chapter will take you to the next level by showing you how to work with HTTPS and how to program your own TCP clients and servers, as well as your own UDP clients and servers.

Additionally, the chapter will demonstrate how you can program a concurrent TCP server using two examples. The first example will be relatively simple, as the concurrent TCP server will just calculate and return numbers in the Fibonacci sequence. However, the second example will use the code of the `keyValue.go` application from Chapter 4, *The Uses of Composite Types*, as its foundation and convert the **key-value store** into a concurrent TCP application that can operate without the need for a web browser. In this chapter, you will learn about the following topics:

- Working with HTTPS traffic
- The `net` standard Go package
- Developing TCP clients and TCP servers
- Developing a concurrent TCP server
- Developing UDP clients and UDP servers
- Modifying `kvSaveLoad.go` from Chapter 8, *Telling a UNIX System What to Do*, so that it can serve requests through TCP connections
- Executing a TCP/IP server on a Docker image
- Creating an RCP client and RCP server

Working with HTTPS traffic

Before you create TCP/IP servers with Go, this chapter will teach you how to work with the HTTPS protocol in Go, which is the secure version of the HTTP protocol. Notice that the default TCP port for HTTPS is 443, but you can use any port number you want as long as you put it in the URL.

Creating certificates

In order to be able to follow and execute the code examples of this section, you will first need to create certificates because HTTPS requires them. On a macOS Mojave machine, you will need to execute the following commands:

```
$ openssl genrsa -out server.key 2048
Generating RSA private key, 2048 bit long modulus
.....+++
.....+++
e is 65537 (0x10001)
$ openssl ecparam -genkey -name secp384r1 -out server.key
$ openssl req -new -x509 -sha256 -key server.key -out server.crt -days 3650
```

The last command will ask you some questions that are not displayed here. It does not really matter what information you put in, which means that you are free to leave most of the answers blank. What you will have after executing these commands are the following two files:

```
$ ls -l server.crt
-rw-r--r--  1 mtsouk  staff   501 May 16 09:42 server.crt
$ ls -l server.key
-rw-r--r--  1 mtsouk  staff   359 May 16 09:42 server.key
```

Notice that if a certificate is self-signed, as the one that we have just generated is, you will need to use the `InsecureSkipVerify: true` option in the `http.Transport` structure for the HTTPS client to work – this will be illustrated shortly.

Now, you should create a certificate for the client, which requires the execution of a single command:

```
$ openssl req -x509 -nodes -newkey rsa:2048 -keyout client.key -out
client.crt -days 3650 -subj "/"
Generating a 2048 bit RSA private key
.....+++
.....+++
```

```
writing new private key to 'client.key'
-----
```

The previous command will generate two new files:

```
$ ls -l client.*
-rw-r--r--  1 mtsouk  staff   924 May 16 22:17 client.crt
-rw-r--r--  1 mtsouk  staff  1704 May 16 22:17 client.key
```

We are now ready to continue and talk about creating an HTTPS client.

An HTTPS client

Nowadays, most web sites work using HTTPS instead of HTTP. Therefore, in this subsection, you will learn how to create an HTTPS client. The name of the program will be `httpsClient.go` and it is going to be presented in three parts.



Depending on the deployment architecture, Go programs might just speak HTTP and some other service (such as the Nginx web server or a cloud-supplied service) might provide the **Secure Sockets Layer (SSL)** part.

The first part of `httpsClient.go` contains the following code:

```
package main

import (
    "crypto/tls"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
    "strings"
)
```

The most important package is `crypto/tls`, which, according to its documentation, partially implements **Transport Layer Security (TLS)** 1.2, as specified in RFC 5246, and TLS 1.3, as specified in RFC 8446.

The second part of `httpsClient.go` is as follows:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
    }
}
```

```
        return
    }
    URL := os.Args[1]

    tr := &http.Transport{
        TLSClientConfig: &tls.Config{},
    }
    client := &http.Client{Transport: tr}
    response, err := client.Get(URL)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer response.Body.Close()
```

The `http.Transport` structure is configured for TLS using `TLSClientConfig`, which holds another structure named `tls.Config` that, at this point, uses its default values.

The last part of the `httpsClient.go` HTTPS client contains the following code for reading the HTTPS server response and printing it on the screen:

```
    content, _ := ioutil.ReadAll(response.Body)
    s := strings.TrimSpace(string(content))

    fmt.Println(s)
}
```

Executing `httpsClient.go` in order to read a secure web site will generate the following kind of output:

```
$ go run httpsClient.go https://www.google.com
<!doctype html><html itemscope="" itemtype="http://schema.org/WebPage"
lang="el"><head><meta content="text/html; charset=UTF-8" http-
equiv="Content-Type"><meta
content="/images/branding/googleg/1x/googleg_standard_color_128dp.png"
.
.
.
```

However, `httpsClient.go` might fail in some cases, depending on the server certificate:

```
$ go run httpsClient.go https://www.mtsoukalos.eu/
Get https://www.mtsoukalos.eu/: x509: certificate signed by unknown
authority
```

The solution to this problem is the use of the `InsecureSkipVerify: true` option in the initialization of `http.Transport`.

You can try that now on your own or wait a little bit for `TLSclient.go`.

A simple HTTPS server

In this subsection, you are going to see the Go code of a HTTPS server. The implementation of the simple HTTPS server is saved in `https.go` and it is going to be presented in three parts.

The first part of `https.go` is as follows:

```
package main

import (
    "fmt"
    "net/http"
)

var PORT = ":1443"
```

The second part of `https.go` contains the following code:

```
func Default(w http.ResponseWriter, req *http.Request) {
    fmt.Fprintf(w, "This is an example HTTPS server!\n")
}
```

This function will handle all incoming HTTPS connections.

The last part of `https.go` is as follows:

```
func main() {
    http.HandleFunc("/", Default)
    fmt.Println("Listening to port number", PORT)

    err := http.ListenAndServeTLS(PORT, "server.crt", "server.key", nil)
    if err != nil {
        fmt.Println("ListenAndServeTLS: ", err)
        return
    }
}
```

The `ListenAndServeTLS()` function is similar to the `ListenAndServe()` function used in the previous chapter – their main difference is that `ListenAndServeTLS()` expects HTTPS connections, whereas `ListenAndServe()` cannot serve HTTPS clients. Additionally, `ListenAndServeTLS()` requires more parameters than `ListenAndServe()` because it uses a certificate file as well as a key file.

Executing `https.go` and connecting to it using the `httpsClient.go` client will generate the following kind of output to the `httpsClient.go` client:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names,
but wanted to match localhost
```

Once again, the use of a self-signed certificate will not allow `httpsClient.go` to connect to our HTTPS server, which is an issue with the implementations of both the client and the server. In this case, the output of `https.go` will be as follows:

```
$ go run https.go
Listening to port number :1443
2019/05/17 10:11:21 http: TLS handshake error from [::1]:56716: remote
error: tls: bad certificate
```

The HTTPS server developed in this section uses HTTPS via SSL, which is not the most secure option. A better option is TLS – a Go implementation of an HTTPS server that uses TLS will be presented in the next section.

Developing a TLS server and client

In this section, we are going to implement a TLS server named `TLSserver.go`, which will be presented in four parts. This time, the HTTPS server will be better than the `https.go` server implemented in the previous section.

The first part of `TLSserver.go` is as follows:

```
package main

import (
    "crypto/tls"
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
)

var PORT = ":1443"

type handler struct {
}
```

The second part of `TLSServer.go` contains the following code:

```
func (h *handler) ServeHTTP(w http.ResponseWriter, req *http.Request) {
    w.Write([]byte("Hello world!\n"))
}
```

This is the handler function of the web server that handles all client connections.

The third part of `TLSServer.go` is as follows:

```
func main() {
    caCert, err := ioutil.ReadFile("client.crt")
    if err != nil {
        fmt.Println(err)
        return
    }

    caCertPool := x509.NewCertPool()
    caCertPool.AppendCertsFromPEM(caCert)
    cfg := &tls.Config{
        ClientAuth: tls.RequireAndVerifyClientCert,
        ClientCAs:  caCertPool,
    }

    srv := &http.Server{
        Addr:      PORT,
        Handler:   &handler{},
        TLSConfig: cfg,
    }
    fmt.Println("Listening to port number", PORT)
    fmt.Println(srv.ListenAndServeTLS("server.crt", "server.key"))
}
```

The `x509` Go package parses **X.509**-encoded keys and certificates. You can find more information about it at <https://golang.org/pkg/crypto/x509/>.

The last part of `TLSServer.go` contains the following code:

```
srv := &http.Server{
    Addr:      PORT,
    Handler:   &handler{},
    TLSConfig: cfg,
}
fmt.Println("Listening to port number", PORT)
fmt.Println(srv.ListenAndServeTLS("server.crt", "server.key"))
}
```

The `ListenAndServeTLS()` call starts the HTTPS server – its full configuration is held in the `http.Server` structure.

If you try to use `httpsClient.go` with `TLSServer.go`, you will get the following kind of output from the client:

```
$ go run httpsClient.go https://localhost:1443
Get https://localhost:1443: x509: certificate is not valid for any names,
but wanted to match localhost
```

In this case, the server will generate the following output:

```
$ go run TLSserver.go
Listening to port number :1443
2019/05/17 10:05:11 http: TLS handshake error from [::1]:56569: remote
error: tls: bad certificate
```

As mentioned earlier, in order for the `httpsClient.go` HTTPS client to communicate successfully with `TLSserver.go`, which uses a self-signed certificate, you will need to add the `InsecureSkipVerify: true` option to `http.Transport`. The version of `httpsClient.go` that works with `TLSserver.go` and contains the `InsecureSkipVerify: true` option is saved as `TLScient.go` and will be presented in four parts.

The first part of `TLScient.go` is as follows:

```
package main

import (
    "crypto/tls"
    "crypto/x509"
    "fmt"
    "io/ioutil"
    "net/http"
    "os"
    "path/filepath"
)
```

The second part of `TLScient.go` contains the following code:

```
func main() {
    if len(os.Args) != 2 {
        fmt.Printf("Usage: %s URL\n", filepath.Base(os.Args[0]))
        return
    }
    URL := os.Args[1]

    caCert, err := ioutil.ReadFile("server.crt")
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The third part of `TLScient.go` is the following:

```
caCertPool := x509.NewCertPool()
```

```
caCertPool.AppendCertsFromPEM(caCert)
cert, err := tls.LoadX509KeyPair("client.crt", "client.key")

if err != nil {
    fmt.Println(err)
    return
}

client := &http.Client{
    Transport: &http.Transport{
        TLSClientConfig: &tls.Config{
            RootCAs:      caCertPool,
            InsecureSkipVerify: true,
            Certificates:   []tls.Certificate{cert},
        },
    },
}

resp, err := client.Get(URL)
if err != nil {
    fmt.Println(err)
    return
}
```

In this part, you can see the use of `InsecureSkipVerify`.

The last part of `TLSclient.go` contains the following Go code:

```
htmlData, err := ioutil.ReadAll(resp.Body)
if err != nil {
    fmt.Println(err)
    return
}

defer resp.Body.Close()
fmt.Printf("%v\n", resp.Status)
fmt.Printf(string(htmlData))
}
```

If you try to connect `TLSclient.go` to `TLSServer.go`, you will get the expected output, which is the following:

```
$ go run TLSclient.go https://localhost:1443
200 OK
Hello world!
```


If you try to connect `TLScient.go` to `https.go`, you will get the expected output, which means that `TLScient.go` is a pretty good HTTPS client implementation:

```
$ go run TLScient.go https://localhost:1443
200 OK
This is an example HTTPS server!
```

The net standard Go package

That is enough with HTTPS. It is time to begin talking about the core protocols of TCP/IP, which are TCP, IP, and UDP.

You cannot create a TCP or UDP client or server in Go without using the functionality offered by the `net` package. The `net.Dial()` function is used to connect to a network as a client, whereas the `net.Listen()` function is used to tell a Go program to accept network connections and thus act as a server. The return value of both the `net.Dial()` function and the `net.Listen()` function is of the `net.Conn` type, which implements the `io.Reader` and `io.Writer` interfaces. The first parameter of both functions is the network type, but this is where their similarities end.

A TCP client

As you already know from the previous chapter, TCP's principal characteristic is that it is a reliable protocol. The TCP header of each packet includes the **source port** and **destination port** fields. These two fields, plus the source and destination IP addresses, are combined to uniquely identify every single TCP connection. The name of the TCP client that will be developed in this section is `TCPclient.go`, and it will be presented in four parts. The first part of `TCPclient.go` is shown in the following Go code:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second code segment of `TCPclient.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide host:port.")
        return
    }

    CONNECT := arguments[1]
    c, err := net.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `net.Dial()` function is used to connect to the remote server. The first parameter of the `net.Dial()` function defines the network that will be used, while the second parameter defines the server address, which must also include the port number. Valid values for the first parameter are `tcp`, `tcp4` (IPv4-only), `tcp6` (IPv6-only), `udp`, `udp4` (IPv4-only), `udp6` (IPv6-only), `ip`, `ip4` (IPv4-only), `ip6` (IPv6-only), `unix` (UNIX sockets), `unixgram`, and `unixpacket`.

The third part of `TCPclient.go` contains the following code:

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(c, text+"\n")
}
```

The preceding code is for getting user input, which can be verified by the use of the `os.Stdin` file for reading. Ignoring the error value returned by `reader.ReadString()` is not good practice, but it saves some space here. Certainly, you should never do that in production software.

The last part of `TCPclient.go` follows:

```
message, _ := bufio.NewReader(c).ReadString('\n')
fmt.Print("->: " + message)
if strings.TrimSpace(string(text)) == "STOP" {
    fmt.Println("TCP client exiting...")
    return
}
}
```

For testing purposes, `TCPclient.go` will connect to a TCP server that is implemented using `netcat(1)`, and will create the following type of output:

```
$ go run TCPclient.go 8001
dial tcp: address 8001: missing port in address
$ go run TCPclient.go localhost:8001
>> Hello from TCPclient.go!
->: Hi from nc!
>> STOP
->:
TCP client exiting...
```

The output of the first command illustrates what will happen if you do not include a host name in the command-line arguments of `TCPclient.go`. The output of the `netcat(1)` TCP server, which should be executed first, is as follows:

```
$ nc -l 127.0.0.1 8001
Hello from TCPclient.go!
Hi from nc!
STOP
```



Notice that a client for a given protocol, such as TCP and UDP, can be reasonably generic in nature, which means that it can talk to many kinds of servers that support its protocol. As you will soon see, this is not the case with server applications, which must implement a prearranged functionality using a prearranged protocol.

A slightly different version of the TCP client

Go offers a different family of functions that also allows you to develop TCP clients and servers. In this section, you will learn how to program a TCP client using these functions.

The name of the TCP client will be `otherTCPclient.go`, and it is going to be presented in four parts. The first code segment from `otherTCPclient.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second code portion from `otherTCPclient.go` contains the following code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a server:port string!")
        return
    }

    CONNECT := arguments[1]

    tcpAddr, err := net.ResolveTCPAddr("tcp4", CONNECT)
    if err != nil {
        fmt.Println("ResolveTCPAddr:", err.Error())
        return
    }
}
```

The `net.ResolveTCPAddr()` function returns the address of a TCP endpoint (type `TCPAddr`) and can only be used for TCP networks.

The third part of `otherTCPclient.go` contains the following code:

```
conn, err := net.DialTCP("tcp4", nil, tcpAddr)
if err != nil {
    fmt.Println("DialTCP:", err.Error())
    return
}
```

The `net.DialTCP()` function is equivalent to `net.Dial()` but it is for TCP networks only.

The remaining code of `otherTCPclient.go` is as follows:

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    fmt.Fprintf(conn, text+"\n")

    message, _ := bufio.NewReader(conn).ReadString('\n')
    fmt.Print("->: " + message)
    if strings.TrimSpace(string(text)) == "STOP" {
        fmt.Println("TCP client exiting...")
        conn.Close()
        return
    }
}
```

Executing `otherTCPclient.go` and interacting with a TCP server will generate the following type of output:

```
$ go run otherTCPclient.go localhost:8001
>> Hello from otherTCPclient.go!
->: Hi from netcat!
>> STOP
->:
TCP client exiting...
```

For this example, the TCP server is supported by the `netcat (1)` utility, which is executed as follows:

```
$ nc -l 127.0.0.1 8001
Hello from otherTCPclient.go!
Hi from netcat!
STOP
```

A TCP server

The TCP server that is going to be developed in this section will return the current date and time to the client in a single network packet. In practice, this means that after accepting a client connection, the server will get the time and date from the UNIX system and send that data back to the client.

The name of the utility is `TCPserver.go`, and it will be presented in four parts.

The first part of `TCPserver.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
    "time"
)
```

The second code portion of `TCPserver.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
```

```

        fmt.Println("Please provide port number")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

```

The `net.Listen()` function listens for connections. If the second parameter does not contain an IP address but only a port number, `net.Listen()` will listen on all available IP addresses of the local system.

The third segment from `TCPserver.go` is as follows:

```

    c, err := l.Accept()
    if err != nil {
        fmt.Println(err)
        return
    }

```

The `Accept()` function waits for the next connection and returns a generic `Conn` variable. The only thing that is wrong with this particular TCP server is that it can only serve the first TCP client that is going to connect to it because the `Accept()` call is outside the `for` loop that is coming next. The remaining Go code of `TCPserver.go` is as follows:

```

    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }
        if strings.TrimSpace(string(netData)) == "STOP" {
            fmt.Println("Exiting TCP server!")
            return
        }

        fmt.Print("-> ", string(netData))
        t := time.Now()
        myTime := t.Format(time.RFC3339) + "\n"
        c.Write([]byte(myTime))
    }
}

```

Executing `TCPserver.go` and interacting with it using a TCP client application will generate the following type of output:

```
$ go run TCPserver.go 8001
-> HELLO
Exiting TCP server!
```

On the client side, you will see the following output:

```
$ nc 127.0.0.1 8001
HELLO
2019-05-18T22:50:31+03:00
STOP
```

If the `TCPserver.go` utility attempts to use a TCP port that is already in use by another UNIX process, you will get the following type of error message:

```
$ go run TCPserver.go 9000
listen tcp :9000: bind: address already in use
```

Finally, if the `TCPserver.go` utility attempts to use a TCP port in the 1-1024 range that requires root privileges on UNIX systems, you will get the following type of error message:

```
$ go run TCPserver.go 80
listen tcp :80: bind: permission denied
```

A slightly different version of the TCP server

In this section, you are going to see an alternative implementation of a TCP server in Go. This time, the TCP server implements the **Echo service**, which basically returns to the client, the data that the client sent. The program is called `otherTCPserver.go`, and it will be presented in four parts.

The first part of `otherTCPserver.go` is as follows:

```
package main

import (
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second portion of `otherTCPserver.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    SERVER := "localhost" + ":" + arguments[1]

    s, err := net.ResolveTCPAddr("tcp", SERVER)
    if err != nil {
        fmt.Println(err)
        return
    }

    l, err := net.ListenTCP("tcp", s)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `net.ListenTCP()` function is equivalent to `net.Listen()` for TCP networks.

The third segment of `otherTCPserver.go` is as follows:

```
buffer := make([]byte, 1024)
conn, err := l.Accept()
if err != nil {
    fmt.Println(err)
    return
}
```

The remaining code of `otherTCPserver.go` is as follows:

```
for {
    n, err := conn.Read(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }

    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
        fmt.Println("Exiting TCP server!")
        conn.Close()
        return
    }
}
```



```

        fmt.Print("> ", string(buffer[0:n-1]))
    }, err = conn.Write(buffer)
    if err != nil {
        fmt.Println(err)
        return
    }
}
}

```

Executing `otherTCPserver.go` and using a client for interacting with it will generate the following type of output:

```

$ go run otherTCPserver.go 8001
> 1
> 2
> Hello!
> Exiting TCP server!

```

On the client side, which in this case is going to be `otherTCPclient.go`, you will see the following type of output:

```

$ go run otherTCPclient.go localhost:8001
>> 1
->: 1
>> 2
->: 2
>> Hello!
->: Hello!
>> ->:
>> STOP
->: TCP client exiting...

```

Finally, I will present a way of finding out the name of the process that listens to a given TCP or UDP port on a UNIX machine. So, if you want to discover which process uses TCP port number 8001, you should execute the following command:

```

$ sudo lsof -n -i :8001
COMMAND    PID    USER   FD   TYPE    DEVICE  SIZE/OFF  NODE  NAME
TCPserver  86775  mtsouk   3u   IPv6  0x98d55014e6c9360f      0t0  TCP *:vcom-
tunnel (LISTEN)

```

A UDP client

If you know how to develop a TCP client, then you should find developing a UDP client much easier due to the simplicity of the UDP protocol.



The biggest difference between UDP and TCP is that UDP is not reliable by design. This also means that, in general, UDP is simpler than TCP because UDP does not need to keep the state of a UDP connection. Put simply, UDP is like "fire and forget," which in some cases is perfect.

The name of the utility presented for this topic is `UDPclient.go`, and it will be presented in four code segments. The first part of `UDPclient.go` is as follows:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strings"
)
```

The second segment of `UDPclient.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string")
        return
    }
    CONNECT := arguments[1]

    s, err := net.ResolveUDPAddr("udp4", CONNECT)
    c, err := net.DialUDP("udp4", nil, s)

    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Printf("The UDP server is %s\n", c.RemoteAddr().String())
    defer c.Close()
}
```

The `net.ResolveUDPAddr()` function returns an address of a UDP endpoint as defined by its second parameter. The first parameter (`udp4`) specifies that the program will support the IPv4 protocol only.

The `net.DialUDP()` function used is like `net.Dial()` for UDP networks.

The third segment of `UDPclient.go` contains the following code:

```
for {
    reader := bufio.NewReader(os.Stdin)
    fmt.Print(">> ")
    text, _ := reader.ReadString('\n')
    data := []byte(text + "\n")
    _, err = c.Write(data)
    if strings.TrimSpace(string(data)) == "STOP" {
        fmt.Println("Exiting UDP client!")
        return
    }
}
```

The preceding code requires that the user types some text, which is sent to the UDP server. The user text is read from UNIX standard input using `bufio.NewReader(os.Stdin)`. The `Write(data)` method sends the data over the UDP network connection.

The rest of the Go code is as follows:

```
if err != nil {
    fmt.Println(err)
    return
}

buffer := make([]byte, 1024)
n, _, err := c.ReadFromUDP(buffer)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply: %s\n", string(buffer[0:n]))
}
```

Once the client data is sent, you must wait for the data that the UDP server has to send, which is read using `ReadFromUDP()`. Executing `UDPclient.go` and interacting with the `netcat(1)` utility that acts as a UDP server will generate the following type of output:

```
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: Hi there!
>> Have to leave - bye!
Reply: OK.
>> STOP
Exiting UDP client!
```

On the UDP server side, the output will be as follows:

```
$ nc -v -u -l 127.0.0.1 8001
Hello!
Hi there!
Have to leave - bye!
OK.
STOP
^C
```

The reason for typing *Ctrl+C* in order to terminate `nc (1)` is that `nc (1)` does not have any code that tells it to terminate when it receives the `STOP` string as input.

Developing a UDP server

The purpose of the UDP server that is going to be developed in this section is returning random numbers between 1 and 1,000 to its UDP clients. The name of the program is `UDPserver.go`, and it is presented in four segments.

The first part of `UDPserver.go` is as follows:

```
package main

import (
    "fmt"
    "math/rand"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second segment of `UDPserver.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }
}
```

```
PORT := ":" + arguments[1]

s, err := net.ResolveUDPAddr("udp4", PORT)
if err != nil {
    fmt.Println(err)
    return
}
```

The third segment of `UDPserver.go` contains the following code:

```
connection, err := net.ListenUDP("udp4", s)
if err != nil {
    fmt.Println(err)
    return
}

defer connection.Close()
buffer := make([]byte, 1024)
rand.Seed(time.Now().Unix())
```

The `net.ListenUDP()` function acts like `net.ListenTCP()` for UDP networks.

The remaining Go code of `UDPserver.go` is as follows:

```
for {
    n, addr, err := connection.ReadFromUDP(buffer)
    fmt.Print("-> ", string(buffer[0:n-1]))

    if strings.TrimSpace(string(buffer[0:n])) == "STOP" {
        fmt.Println("Exiting UDP server!")
        return
    }

    data := []byte(strconv.Itoa(random(1, 1001)))
    fmt.Printf("data: %s\n", string(data))
    _, err = connection.WriteToUDP(data, addr)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

The `ReadFromUDP()` function allows you to read data from a UDP connection using a buffer, which, as expected, is a **byte slice**.

Executing `UDPserver.go` and connecting to it with `UDPclient.go` will generate the following type of output:

```
$ go run UDPserver.go 8001
-> Hello!
data: 156
-> Another random number please :)
data: 944
-> Leaving...
data: 491
-> STOP
Exiting UDP server!
On the client side, the output will be as follows:
$ go run UDPclient.go localhost:8001
The UDP server is 127.0.0.1:8001
>> Hello!
Reply: 156
>> Another random number please :)
Reply: 944
>> Leaving...
Reply: 491
>> STOP
Exiting UDP client!
```

A concurrent TCP server

In this section, you will learn how to develop a **concurrent TCP server** using goroutines. For each incoming connection to the TCP server, the program will start a new goroutine to handle that request. This allows it to accept more requests, which means that a concurrent TCP server can serve multiple clients simultaneously.

The job of the TCP concurrent server is to accept a positive integer and return a natural number from the Fibonacci sequence. If there is an error in the input, the return value will be `-1`. As the calculation of numbers in the Fibonacci sequence can be slow, we are going to use an algorithm that was first presented in *Chapter 11, Code Testing, Optimization, and Profiling*, and was included in `benchmarkMe.go`. Additionally, this time, the algorithm used will be explained a little more.

The name of the program is `fibotcp.go`, and its code is presented in five parts. As it is considered good practice to be able to define the port number of a web service as a command-line parameter, `fibotcp.go` will do exactly that.

The first part of `fibotcp.go` contains the following Go code:

```
package main

import (
    "bufio"
    "fmt"
    "net"
    "os"
    "strconv"
    "strings"
    "time"
)
```

The second code portion of `fibotcp.go` contains the following Go code:

```
func f(n int) int {
    fn := make(map[int]int)
    for i := 0; i <= n; i++ {
        var f int
        if i <= 2 {
            f = 1
        } else {
            f = fn[i-1] + fn[i-2]
        }
        fn[i] = f
    }
    return fn[n]
}
```

In the preceding code, you can see the implementation of the `f()` function, which generates natural numbers that belong to the Fibonacci sequence. The algorithm used is difficult to understand at first, but it is very efficient and therefore fast.

Firstly, the `f()` function uses a Go map named `fn`, which is pretty unusual when calculating numbers of the Fibonacci sequence. Secondly, the `f()` function uses a `for` loop, which is also fairly unusual. Finally, the `f()` function does not use recursion, which is the main reason for the speed of its operation.

The idea behind the algorithm used in `f()`, which uses a **dynamic programming** technique, is that whenever a Fibonacci number is computed, it is put into the `fn` map so that it will not be computed again. This simple idea saves a lot of time, especially when large Fibonacci numbers need to be calculated, because then you do not have to calculate the same Fibonacci number multiple times.

The third code segment of `fibotcp.go` is as follows:

```
func handleConnection(c net.Conn) {
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }

        temp := strings.TrimSpace(string(netData))
        if temp == "STOP" {
            break
        }

        fibo := "-1\n"
        n, err := strconv.Atoi(temp)
        if err == nil {
            fibo = strconv.Itoa(f(n)) + "\n"
        }
        c.Write([]byte(string(fibo)))
    }
    time.Sleep(5 * time.Second)
    c.Close()
}
```

The `handleConnection()` function deals with each client of the concurrent TCP server.

The fourth part of `fibotcp.go` is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp4", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()
```


The remaining Go code of `fibotcp.go` is as follows:

```
    for {
        c, err := l.Accept()
        if err != nil {
            fmt.Println(err)
            return
        }
        go handleConnection(c)
    }
}
```

The concurrency of the program is implemented by the `go handleConnection(c)` statement, which commences a new goroutine each time a new TCP client comes from the internet or a local network. The goroutine is executed concurrently, which gives the server the opportunity to serve even more clients.

Executing `fibotcp.go` and interacting with it using both `netcat (1)` and `TCPclient.go` on two different terminals will generate the following type of output:

```
$ go run fibotcp.go 9000
n: 10
fibotcp: 55
n: 0
fibotcp: 1
n: -1
fibotcp: 0
n: 100
fibotcp: 3736710778780434371
n: 12
fibotcp: 144
n: 12
fibotcp: 144
```

The output will be as follows on the `TCPclient.go` side:

```
$ go run TCPclient.go localhost:9000
>> 12
->: 144
>> a
->: -1
>> STOP
->: TCP client exiting...
```

On the `netcat (1)` side, the output will be as follows:

```
$ nc localhost 9000
10
```

```

55
0
1
-1
0
100
3736710778780434371
ads
-1
STOP

```

When you send the `STOP` string to the server process, the goroutine that serves that particular TCP client will terminate, which will cause the connection to end.

Finally, the impressive thing here is that both clients are served at the same time, which can be verified by the output of the following command:

```

$ netstat -anp TCP | grep 9000
tcp4      0 0 127.0.0.1.9000  127.0.0.1.57309  ESTABLISHED
tcp4      0 0 127.0.0.1.57309  127.0.0.1.9000  ESTABLISHED
tcp4      0 0 127.0.0.1.9000  127.0.0.1.57305  ESTABLISHED
tcp4      0 0 127.0.0.1.57305  127.0.0.1.9000  ESTABLISHED
tcp4      0 0 *.9000          *.*              LISTEN

```

The last line of the output of the preceding command tells us that there is a process that listens to port 9000, which means that you can still connect to port 9000. The first two lines of the output say that there is a client that uses port 57309 to talk to the server process. The third and fourth lines of the preceding output verify that there is another client that communicates with the server that listens to port 9000. That client uses TCP port 57305.

A handy concurrent TCP server

Although the concurrent TCP server from the previous section works fine, it does not serve a practical application. Therefore, in this section, you will learn how to convert the `keyValue.go` application from Chapter 4, *The Uses of Composite Types*, into a fully-featured concurrent TCP application.

What we are going to do is create our own kind of TCP protocol in order to be able to interact with the key-value store from a network. You will need a keyword for each one of the functions of the key-value store. For reasons of simplicity, each keyword will be followed by the relevant data. The result of most commands will be a success or failure message.



Designing your own TCP- or UDP-based protocols is not an easy job. This means that you should be particularly specific and careful when designing a new protocol.

The name of the utility shown in this topic is `kvTCP.go`, and it is presented in six parts.

The first part of `kvTCP.go` is as follows:

```
package main

import (
    "bufio"
    "encoding/gob"
    "fmt"
    "net"
    "os"
    "strings"
)

type myElement struct {
    Name      string
    Surname   string
    Id        string
}

const welcome = "Welcome to the Key Value store!\n"

var DATA = make(map[string]myElement)
var DATAFILE = "/tmp/dataFile.gob"
```

The second part of `kvTCP.go` contains the following Go code:

```
func handleConnection(c net.Conn) {
    c.Write([]byte(welcome))
    for {
        netData, err := bufio.NewReader(c).ReadString('\n')
        if err != nil {
            fmt.Println(err)
            return
        }

        command := strings.TrimSpace(string(netData))
        tokens := strings.Fields(command)
        switch len(tokens) {
        case 0:
            continue
        case 1:
```

```
        tokens = append(tokens, "")
        tokens = append(tokens, "")
        tokens = append(tokens, "")
        tokens = append(tokens, "")
    case 2:
        tokens = append(tokens, "")
        tokens = append(tokens, "")
        tokens = append(tokens, "")
    case 3:
        tokens = append(tokens, "")
        tokens = append(tokens, "")
    case 4:
        tokens = append(tokens, "")
    }

    switch tokens[0] {
    case "STOP":
        err = save()
        if err != nil {
            fmt.Println(err)
        }
        c.Close()
        return
    case "PRINT":
        PRINT(c)
    case "DELETE":
        if !DELETE(tokens[1]) {
            netData := "Delete operation failed!\n"
            c.Write([]byte(netData))
        } else {
            netData := "Delete operation successful!\n"
            c.Write([]byte(netData))
        }
    case "ADD":
        n := myElement{tokens[2], tokens[3], tokens[4]}
        if !ADD(tokens[1], n) {
            netData := "Add operation failed!\n"
            c.Write([]byte(netData))
        } else {
            netData := "Add operation successful!\n"
            c.Write([]byte(netData))
        }
        err = save()
        if err != nil {
            fmt.Println(err)
        }
    case "LOOKUP":
        n := LOOKUP(tokens[1])
```

```

        if n != nil {
            netData := fmt.Sprintf("%v\n", *n)
            c.Write([]byte(netData))
        } else {
            netData := "Did not find key!\n"
            c.Write([]byte(netData))
        }
    case "CHANGE":
        n := myElement{tokens[2], tokens[3], tokens[4]}
        if !CHANGE(tokens[1], n) {
            netData := "Update operation failed!\n"
            c.Write([]byte(netData))
        } else {
            netData := "Update operation successful!\n"
            c.Write([]byte(netData))
        }
        err = save()
        if err != nil {
            fmt.Println(err)
        }
    default:
        netData := "Unknown command - please try again!\n"
        c.Write([]byte(netData))
    }
}
}

```

The `handleConnection()` function communicates with each TCP client and interprets the client input.

The third segment of `kvTCP.go` contains the following Go code:

```

func save() error {
    fmt.Println("Saving", DATAFILE)
    err := os.Remove(DATAFILE)
    if err != nil {
        fmt.Println(err)
    }

    saveTo, err := os.Create(DATAFILE)
    if err != nil {
        fmt.Println("Cannot create", DATAFILE)
        return err
    }
    defer saveTo.Close()

    encoder := gob.NewEncoder(saveTo)
    err = encoder.Encode(DATA)
}

```

```
        if err != nil {
            fmt.Println("Cannot save to", DATAFILE)
            return err
        }
        return nil
    }

    func load() error {
        fmt.Println("Loading", DATAFILE)
        loadFrom, err := os.Open(DATAFILE)
        defer loadFrom.Close()
        if err != nil {
            fmt.Println("Empty key/value store!")
            return err
        }

        decoder := gob.NewDecoder(loadFrom)
        decoder.Decode(&DATA)
        return nil
    }
}
```

The fourth segment of `kvTCP.go` is as follows:

```
func ADD(k string, n myElement) bool {
    if k == "" {
        return false
    }

    if LOOKUP(k) == nil {
        DATA[k] = n
        return true
    }
    return false
}

func DELETE(k string) bool {
    if LOOKUP(k) != nil {
        delete(DATA, k)
        return true
    }
    return false
}

func LOOKUP(k string) *myElement {
    _, ok := DATA[k]
    if ok {
        n := DATA[k]
        return &n
    }
}
```

```
    } else {
        return nil
    }
}

func CHANGE(k string, n myElement) bool {
    DATA[k] = n
    return true
}
```

The implementation of the preceding functions is the same as in `keyValue.go`. None of them talk directly to a TCP client.

The fifth part of `kvTCP.go` contains the following Go code:

```
func PRINT(c net.Conn) {
    for k, d := range DATA {
        netData := fmt.Sprintf("key: %s value: %v\n", k, d)
        c.Write([]byte(netData))
    }
}
```

The `PRINT()` function sends data to a TCP client directly, one line at a time.

The remaining Go code of the program is as follows:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a port number!")
        return
    }

    PORT := ":" + arguments[1]
    l, err := net.Listen("tcp", PORT)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer l.Close()

    err = load()
    if err != nil {
        fmt.Println(err)
    }

    for {
        c, err := l.Accept()
```

```
        if err != nil {
            fmt.Println(err)
            os.Exit(100)
        }
        go handleConnection(c)
    }
}
```

Executing `kvTCP.go` will generate the following type of output:

```
$ go run kvTCP.go 9000
Loading /tmp/dataFile.gob
Empty key/value store!
open /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
remove /tmp/dataFile.gob: no such file or directory
Saving /tmp/dataFile.gob
Saving /tmp/dataFile.gob
```

For the purposes of this section, the `netcat (1)` utility will act as the TCP client of `kvTCP.go`:

```
$ nc localhost 9000
Welcome to the Key Value store!
PRINT
LOOKUP 1
Did not find key!
ADD 1 2 3 4
Add operation successful!
LOOKUP 1
{2 3 4}
ADD 4 -1 -2 -3
Add operation successful!
PRINT
key: 1 value: {2 3 4}
key: 4 value: {-1 -2 -3}
STOP
```

`kvTCP.go` is a concurrent application that uses goroutines and can serve multiple TCP clients simultaneously. However, all of these TCP clients will share the same data.

Creating a Docker image for a Go TCP/IP server

In this section, you will learn how to put `kvTCP.go` into a Docker image and use it from there, which is a pretty convenient way of using a TCP/IP application because a Docker image can be easily transferred to other machines or deployed in **Kubernetes**.

As you might expect, everything will begin with a `Dockerfile`, which will have the following contents:

```
FROM golang:latest

RUN mkdir /files
COPY kvTCP.go /files
WORKDIR /files

RUN go build -o /files/kvTCP kvTCP.go
ENTRYPOINT ["/files/kvTCP","80"]
```

After that, you will need to build the Docker image, as follows:

```
$ docker build -t kvtcp:latest .
Sending build context to Docker daemon 6.656kB
Step 1/6 : FROM golang:latest
----> 7ced090ee82e
Step 2/6 : RUN mkdir /files
----> Running in bbbbada6271f
Removing intermediate container bbbbada6271f
----> 5b0a621eee29
Step 3/6 : COPY kvTCP.go /files
----> 4aab441b14c2
Step 4/6 : WORKDIR /files
----> Running in 7185606bed2e
Removing intermediate container 7185606bed2e
----> 744e9800fdb
Step 5/6 : RUN go build -o /files/kvTCP kvTCP.go
----> Running in f44fcbc8951b
Removing intermediate container f44fcbc8951b
----> a8d00c7ead13
Step 6/6 : ENTRYPOINT ["/files/kvTCP","80"]
----> Running in ec3227170e09
Removing intermediate container ec3227170e09
----> b65ba728849a
Successfully built b65ba728849a
Successfully tagged kvtcp:latest
```

Executing `docker images` will verify that you have created the desired Docker image:

```
$ docker images
REPOSITORY          TAG                 IMAGE ID            CREATED
SIZE
kvtcp                latest             b65ba728849a       26
minutes ago         777MB
landoop/kafka-lenses-dev  latest             289093ceee7b       2 days
ago                 1.37GB
golang               latest             7ced090ee82e       9 days
ago                 774MB
```

Then, you will need to execute that Docker image. This time, you will need to specify some extra parameters to expose the port numbers that you want to make available to the outer world:

```
$ docker run -d -p 5801:80 kvtcp:latest
af9939992a25cb5ccf405b1b97b9c813fb4cb3f3a2e5b13942db637709c8cee2
```

The last command exposes port number 80 from the Docker image to the local machine using port number 5801 on the local machine.

From now on, you will be able to use the TCP server that is located in the Docker image, as follows:

```
$ nc 127.0.0.1 5801
Welcome to the Key Value store!
```

Notice that, until you save your data in a location outside of the Docker image, your data will be lost when you terminate that Docker image.

This capability of Docker gives you the opportunity to use any port number that you want to access the TCP server in the Docker image. However, if you try to map port number 80 again, you will get the following kind of error message:

```
$ docker run -d -p 5801:80 kvtcp:latest
709d44be8668284b101d7dfc253938d13e6797d812821838aa5ab18ea48527ec
docker: Error response from daemon: driver failed programming external
connectivity on endpoint eager_nobel
(fa9d43d3c129734576e824753703d8ac3ff51bcdcdc20e6937b30f3bcfefeff7): Bind
for 0.0.0.0:5801 failed: port is already allocated.
```

As the error message says, the port is already allocated and therefore cannot be used again. However, there is a workaround to this limitation, which can be illustrated by the following command:

```
$ docker run -d -p 5801:80 -p 2000:80 kvtcp:latest
```

```
5cbb17c5bbf720eaa5ce0f1e11cc73dbe5bef3cc925b7936ae1b97e245d54ae8
```

The previous command maps port number 80 from the Docker image to two external TCP ports (5801 and 2000) in a single `docker run` command. Notice that there is a single `kvTCP` server running and that there is a single copy of the data even if that data can be accessed using multiple ports.

This can also be verified by the output of the `docker ps` command:

```
$ docker ps
CONTAINER ID          IMAGE                COMMAND              CREATED
STATUS               PORTS              NAMES
5cbb17c5bbf7         kvtcp:latest        "/files/kvTCP 80"    3 seconds ago
Up 2 seconds         0.0.0.0:2000->80/tcp, 0.0.0.0:5801->80/tcp
compassionate_morse
```

Remote Procedure Call (RPC)

RPC is a client-server mechanism for interprocess communication that uses TCP/IP. Both the RPC client and the RPC server to be developed will use the following package, which is named `sharedRPC.go`:

```
package sharedRPC

type MyFloats struct {
    A1, A2 float64
}

type MyInterface interface {
    Multiply(arguments *MyFloats, reply *float64) error
    Power(arguments *MyFloats, reply *float64) error
}
```

The `sharedRPC` package defines one interface called `MyInterface` and one structure called `MyFloats`, which will be used by both the client and the server. However, only the RPC server will need to implement that interface.

After that, you will need to install the `sharedRPC.go` package by executing the following commands:

```
$ mkdir -p ~/go/src/sharedRPC
$ cp sharedRPC.go ~/go/src/sharedRPC/
$ go install sharedRPC
```

The RPC client

In this section, you are going to see the Go code of the RPC client, which will be saved as `RPCclient.go` and presented in four parts.

The first part of `RPCclient.go` is as follows:

```
package main

import (
    "fmt"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

The second segment of `RPCclient.go` contains the following Go code:

```
func main() {
    arguments := os.Args
    if len(arguments) == 1 {
        fmt.Println("Please provide a host:port string!")
        return
    }

    CONNECT := arguments[1]
    c, err := rpc.Dial("tcp", CONNECT)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

Observe the use of the `rpc.Dial()` function for connecting to the RPC server instead of the `net.Dial()` function, even though the RCP server uses TCP.

The third segment of `RPCclient.go` is as follows:

```
args := sharedRPC.MyFloats{16, -0.5}
var reply float64

err = c.Call("MyInterface.Multiply", args, &reply)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply (Multiply): %f\n", reply)
```

What is being exchanged between the RPC client and the RPC server with the help of the `Call()` function are function names, their arguments, and the results of the function calls, as the RPC client knows nothing about the actual implementation of the functions.

The remaining Go code of `RPCclient.go` is as follows:

```
err = c.Call("MyInterface.Power", args, &reply)
if err != nil {
    fmt.Println(err)
    return
}
fmt.Printf("Reply (Power): %f\n", reply)
}
```

If you try to execute `RPCclient.go` without having an RPC server running, you will get the following type of error message:

```
$ go run RPCclient.go localhost:1234
dial tcp [::1]:1234: connect: connection refused
```

The RPC server

The RPC server will be saved as `RPCserver.go`, and it is presented in five parts.

The first part of `RPCserver.go` is as follows:

```
package main

import (
    "fmt"
    "math"
    "net"
    "net/rpc"
    "os"
    "sharedRPC"
)
```

The second segment of `RPCserver.go` contains the following Go code:

```
type MyInterface struct{}

func Power(x, y float64) float64 {
    return math.Pow(x, y)
}

func (t *MyInterface) Multiply(arguments *sharedRPC.MyFloats, reply
```

```
*float64) error {
    *reply = arguments.A1 * arguments.A2
    return nil
}

func (t *MyInterface) Power(arguments *sharedRPC.MyFloats, reply *float64)
error {
    *reply = Power(arguments.A1, arguments.A2)
    return nil
}
```

In the preceding Go code, the RPC server implements the desired interface, as well as a helper function named `Power()`.

The third segment of `RPCserver.go` is as follows:

```
func main() {
    PORT := ":1234"
    arguments := os.Args
    if len(arguments) != 1 {
        PORT = ":" + arguments[1]
    }
}
```

The fourth part of `RPCserver.go` contains the following code:

```
myInterface := new(MyInterface)
rpc.Register(myInterface)
t, err := net.ResolveTCPAddr("tcp4", PORT)
if err != nil {
    fmt.Println(err)
    return
}
l, err := net.ListenTCP("tcp4", t)
if err != nil {
    fmt.Println(err)
    return
}
```

What makes this program an RPC server is the use of the `rpc.Register()` function. However, as the RPC server uses TCP, it still needs to make function calls to `net.ResolveTCPAddr()` and `net.ListenTCP()`.

The rest of the Go code of `RPCclient.go` is as follows:

```
for {
    c, err := l.Accept()
    if err != nil {
        continue
    }
}
```

```
    }  
    fmt.Printf("%s\n", c.RemoteAddr())  
    rpc.ServeConn(c)  
}  
}
```

The `RemoteAddr()` function returns the IP address and the port number used for communicating with the RPC client. The `rpc.ServeConn()` function serves the RPC client.

Executing `RPCserver.go` and waiting for `RPCclient.go` will create the following type of output:

```
$ go run RPCserver.go  
127.0.0.1:52289
```

Executing `RPCclient.go` will create the following type of output:

```
$ go run RPCclient.go localhost:1234  
Reply (Multiply): -8.000000  
Reply (Power): 0.250000
```

Doing low-level network programming

Although the `http.Transport` structure allows you to modify the various low-level parameters of a network connection, you can write Go code that permits you to read the raw data of network packets.

There are two tricky points here. Firstly, network packets come in binary format, which requires you to look for specific kinds of network packets and not just any type of network packet. Put simply, when reading network packets, you should specify the protocol or protocols that you are going to support in your applications in advance. Secondly, in order to send a network packet, you will have to construct it on your own.

The next utility to be shown is called `lowLevel.go`, and it will be presented in three parts. Notice that `lowLevel.go` captures **Internet Control Message Protocol (ICMP)** packets, which use the IPv4 protocol and print their contents. Also, note that working with raw network data requires root privileges for security reasons.

The first segment of `lowLevel.go` is as follows:

```
package main  
  
import (  
    "fmt"
```

```
    "net"  
)
```

The second part of `lowLevel.go` contains the following Go code:

```
func main() {  
    netaddr, err := net.ResolveIPAddr("ip4", "127.0.0.1")  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
    conn, err := net.ListenIP("ip4:icmp", netaddr)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
}
```

The ICMP protocol is specified in the second part of the first parameter (`ip4:icmp`) of `net.ListenIP()`. Moreover, the `ip4` part tells the utility to capture IPv4 traffic only.

The remaining part of `lowLevel.go` contains the following Go code:

```
    buffer := make([]byte, 1024)  
    n, _, err := conn.ReadFrom(buffer)  
    if err != nil {  
        fmt.Println(err)  
        return  
    }  
  
    fmt.Printf("% X\n", buffer[0:n])  
}
```

The preceding code tells `lowLevel.go` to read just a single network packet because there is no `for` loop.

The ICMP protocol is used by the `ping(1)` and `traceroute(1)` utilities, so one way to create ICMP traffic is to use one of these two tools. The network traffic will be generated using the following commands on both UNIX machines while `lowLevel.go` is already running:

```
$ ping -c 5 localhost  
PING localhost (127.0.0.1): 56 data bytes  
64 bytes from 127.0.0.1: icmp_seq=0 ttl=64 time=0.037 ms  
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms  
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.117 ms  
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.052 ms  
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.049 ms
```



```

--- localhost ping statistics ---
5 packets transmitted, 5 packets received, 0.0% packet loss
round-trip min/avg/max/stddev = 0.037/0.059/0.117/0.030 ms
$ traceroute localhost
traceroute to localhost (127.0.0.1), 64 hops max, 52 byte packets
 1  localhost (127.0.0.1)  0.255 ms  0.048 ms  0.067 ms

```

Executing `lowLevel.go` on a macOS Mojave machine with root privileges will produce the following type of output:

```

$ sudo go run lowLevel.go
03 03 CD DA 00 00 00 00 45 00 34 00 B4 0F 00 00 01 11 00 00 7F 00 00 01 7F
00 00 01 B4 0E 82 9B 00 20 00 00
$ sudo go run lowLevel.go
00 00 0B 3B 20 34 00 00 5A CB 5C 15 00 04 32 A9 08 09 0A 0B 0C 0D 0E 0F 10
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29
2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37

```

The first output example is generated by the `ping(1)` command, whereas the second output example is generated by the `traceroute(1)` command.

Running `lowLevel.go` on a Debian Linux machine will generate the following type of output:

```

$ uname -a
Linux mail 4.14.12-x86_64-linode92 #1 SMP Fri Jan 5 15:34:44 UTC 2018
x86_64 GNU/Linux
# go run lowLevel.go
08 00 61 DD 3F BA 00 01 9A 5D CB 5A 00 00 00 00 26 DC 0B 00 00 00 00 00 10
11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29
2A 2B 2C 2D 2E 2F 30 31 32 33 34 35 36 37
# go run lowLevel.go
03 03 BB B8 00 00 00 00 45 00 00 3C CD 8D 00 00 01 11 EE 21 7F 00 00 01 7F
00 00 01 CB 40 82 9A 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D
4E 4F 50 51 52 53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

```

The output of the `uname(1)` command prints useful information about the Linux system. Note that, on modern Linux machines, you should execute `ping(1)` with the `-4` flag in order to tell it to use the IPv4 protocol.

Grabbing raw ICMP network data

In this subsection, you will learn how to use the `syscall` package to capture raw ICMP network data and `syscall.SetsockoptInt()` in order to set the options of a socket.

Keep in mind that sending raw ICMP data is much more difficult, as you will have to construct the raw network packets on your own. The name of the utility is `syscallNet.go`, and it will be shown in four parts.

The first part of `syscallNet.go` is as follows:

```
package main

import (
    "fmt"
    "os"
    "syscall"
)
```

The second segment of `syscallNet.go` contains the following Go code:

```
func main() {
    fd, err := syscall.Socket(syscall.AF_INET, syscall.SOCK_RAW,
        syscall.IPPROTO_ICMP)
    if err != nil {
        fmt.Println("Error in syscall.Socket:", err)
        return
    }

    f := os.NewFile(uintptr(fd), "captureICMP")
    if f == nil {
        fmt.Println("Error in os.NewFile:", err)
        return
    }
}
```

The `syscall.AF_INET` parameter tells `syscall.Socket()` that you want to work with IPv4. The `syscall.SOCK_RAW` parameter is what makes the generated socket a raw socket. The last parameter, which is `syscall.IPPROTO_ICMP`, tells `syscall.Socket()` that you are interested in ICMP traffic only.

The third part of `syscallNet.go` is as follows:

```
err = syscall.SetsockoptInt(fd, syscall.SOL_SOCKET, syscall.SO_RCVBUF,
256)
if err != nil {
    fmt.Println("Error in syscall.Socket:", err)
    return
}
```

The call to `syscall.SetsockoptInt()` sets the size of the receive buffer of the socket to 256. The `syscall.SOL_SOCKET` parameter is for stating that you want to work on the socket layer level.

The remaining Go code of `syscallNet.go` is as follows:

```

    for {
        buf := make([]byte, 1024)
        numRead, err := f.Read(buf)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Printf("% X\n", buf[:numRead])
    }
}

```

Due to the `for` loop, `syscallNet.go` will keep capturing ICMP network packets until you terminate it manually.

Executing `syscallNet.go` on a macOS High Sierra machine will produce the following type of output:

```

$ sudo go run syscallNet.go
45 00 40 00 BC B6 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 3F 36 71
45 00 00 5A CB 6A 90 00 0B 9F 1A 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 62 FB 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 31 EF 71
45 00 01 5A CB 6A 91 00 0B AC 5F 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 9A 5F 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 1D D6 71
45 00 02 5A CB 6A 92 00 0B C0 76 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 6E 0D 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 09 CF 71
45 00 03 5A CB 6A 93 00 0B D4 7B 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 40 00 3A 07 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 00 00 FE 9C 71
45 00 04 5A CB 6A 94 00 0B DF AB 08 09 0A 0B 0C 0D 0E 0F 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 24 00 45 55 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 12 00
00 00 00 45 00 34 00 C5 73 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9B 00 20 00 00
45 00 24 00 E8 1E 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 10 00
00 00 00 45 00 34 00 C5 74 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9C 00 20 00 00
45 00 24 00 2A 4B 00 00 40 01 00 00 7F 00 00 01 7F 00 00 01 03 03 AB 0E 00
00 00 00 45 00 34 00 C5 75 00 00 01 11 00 00 7F 00 00 01 7F 00 00 01 C5 72
82 9D 00 20 00 00

```

Running `syscallNet.go` on a Debian Linux machine will generate the following type of output:

```
# go run syscallNet.go
45 00 00 54 7F E9 40 00 40 01 BC BD 7F 00 00 01 7F 00 00 01 08 00 6F 07 53
E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 7F EA 00 00 40 01 FC BC 7F 00 00 01 7F 00 00 01 00 00 77 07 53
E3 00 01 FA 6A CB 5A 00 00 00 00 AA 7B 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 54 00 00 34 01 8B 8E 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 40 4F 40 00 34 06 74 6A 6D 4A C1 FD 86 77 DC 57 B0 B8
DD 96 00 00 00 00 52 F1 AB DA 50 14 00 00 90 9E 00 00
45 00 00 54 80 4E 40 00 40 01 BC 58 7F 00 00 01 7F 00 00 01 08 00 7E 01 53
E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 4F 00 00 40 01 FC 57 7F 00 00 01 7F 00 00 01 00 00 86 01 53
E3 00 02 FB 6A CB 5A 00 00 00 00 9A 80 06 00 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 9B 40 00 40 01 BC 0B 7F 00 00 01 7F 00 00 01 08 00 93 EC 53
E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 9C 00 00 40 01 FC 0A 7F 00 00 01 7F 00 00 01 00 00 9B EC 53
E3 00 03 FC 6A CB 5A 00 00 00 00 83 94 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 55 00 00 34 01 8B 8D 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 40 D1 40 00 34 06 73 E8 6D 4A C1 FD 86 77 DC 57 8E 8E
DD 96 00 00 00 00 6C 6E D3 36 50 14 00 00 71 EF 00 00
45 00 00 54 80 F8 40 00 40 01 BB AE 7F 00 00 01 7F 00 00 01 08 00 F2 E7 53
E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 80 F9 00 00 40 01 FB AD 7F 00 00 01 7F 00 00 01 00 00 FA E7 53
E3 00 04 FD 6A CB 5A 00 00 00 00 23 98 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 82 0D 40 00 40 01 BA 99 7F 00 00 01 7F 00 00 01 08 00 4A 82 53
E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 10 11 12 13 14 15
16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 00 00 54 82 0E 00 00 40 01 FA 98 7F 00 00 01 7F 00 00 01 00 00 52 82 53
E3 00 05 FE 6A CB 5A 00 00 00 00 CA FC 06 00 00 00 00 10 11 12 13 14 15
```

```

16 17 18 19 1A 1B 1C 1D 1E 1F 20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E
2F 30 31 32 33 34 35 36 37
45 C0 00 44 68 56 00 00 34 01 8B 8C 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 41 74 40 00 34 06 73 45 6D 4A C1 FD 86 77 DC 57 2E 9B
DD 96 00 00 00 00 C3 D6 44 57 50 14 00 00 09 5A 00 00
45 C0 00 44 68 57 00 00 34 01 8B 8B 86 77 DC 57 6D 4A C1 FD 03 0A 8F 27 00
00 00 00 45 00 00 28 44 27 40 00 33 06 71 92 6D 4A C1 FD 86 77 DC 57 C5 C2
DD 96 00 00 00 00 CF DD DB BE 50 14 00 00 CE C3 00 00
45 C0 00 58 94 B4 00 00 40 01 E7 2E 7F 00 00 01 7F 00 00 01 03 03 F1 DA 00
00 00 00 45 00 00 3C 85 E1 00 00 01 11 35 CE 7F 00 00 01 7F 00 00 01 95 1E
82 9A 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B5 00 00 40 01 E7 2D 7F 00 00 01 7F 00 00 01 03 03 F9 EA 00
00 00 00 45 00 00 3C 85 E2 00 00 01 11 35 CD 7F 00 00 01 7F 00 00 01 8D 0D
82 9B 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B6 00 00 40 01 E7 2C 7F 00 00 01 7F 00 00 01 03 03 D2 EB 00
00 00 00 45 00 00 3C 85 E3 00 00 01 11 35 CC 7F 00 00 01 7F 00 00 01 B4 0B
82 9C 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B7 00 00 40 01 E7 2B 7F 00 00 01 7F 00 00 01 03 03 D6 AC 00
00 00 00 45 00 00 3C 85 E4 00 00 02 11 34 CB 7F 00 00 01 7F 00 00 01 B0 49
82 9D 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B8 00 00 40 01 E7 2A 7F 00 00 01 7F 00 00 01 03 03 F1 B4 00
00 00 00 45 00 00 3C 85 E5 00 00 02 11 34 CA 7F 00 00 01 7F 00 00 01 95 40
82 9E 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 B9 00 00 40 01 E7 29 7F 00 00 01 7F 00 00 01 03 03 CD 43 00
00 00 00 45 00 00 3C 85 E6 00 00 02 11 34 C9 7F 00 00 01 7F 00 00 01 B9 B0
82 9F 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BA 00 00 40 01 E7 28 7F 00 00 01 7F 00 00 01 03 03 9D 8F 00
00 00 00 45 00 00 3C 85 E7 00 00 03 11 33 C8 7F 00 00 01 7F 00 00 01 E9 63
82 A0 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BB 00 00 40 01 E7 27 7F 00 00 01 7F 00 00 01 03 03 A3 13 00
00 00 00 45 00 00 3C 85 E8 00 00 03 11 33 C7 7F 00 00 01 7F 00 00 01 E3 DE
82 A1 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BC 00 00 40 01 E7 26 7F 00 00 01 7F 00 00 01 03 03 D4 66 00
00 00 00 45 00 00 3C 85 E9 00 00 03 11 33 C6 7F 00 00 01 7F 00 00 01 B2 8A
82 A2 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BD 00 00 40 01 E7 25 7F 00 00 01 7F 00 00 01 03 03 A6 8D 00
00 00 00 45 00 00 3C 85 EA 00 00 04 11 32 C5 7F 00 00 01 7F 00 00 01 E0 62
82 A3 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

```

```

45 C0 00 58 94 BE 00 00 40 01 E7 24 7F 00 00 01 7F 00 00 01 03 03 F1 C6 00
00 00 00 45 00 00 3C 85 EB 00 00 04 11 32 C4 7F 00 00 01 7F 00 00 01 95 28
82 A4 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 BF 00 00 40 01 E7 23 7F 00 00 01 7F 00 00 01 03 03 A3 FE 00
00 00 00 45 00 00 3C 85 EC 00 00 04 11 32 C3 7F 00 00 01 7F 00 00 01 E2 EF
82 A5 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C0 00 00 40 01 E7 22 7F 00 00 01 7F 00 00 01 03 03 B9 AA 00
00 00 00 45 00 00 3C 85 ED 00 00 05 11 31 C2 7F 00 00 01 7F 00 00 01 CD 42
82 A6 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C1 00 00 40 01 E7 21 7F 00 00 01 7F 00 00 01 03 03 B3 B7 00
00 00 00 45 00 00 3C 85 EE 00 00 05 11 31 C1 7F 00 00 01 7F 00 00 01 D3 34
82 A7 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C2 00 00 40 01 E7 20 7F 00 00 01 7F 00 00 01 03 03 F2 62 00
00 00 00 45 00 00 3C 85 EF 00 00 05 11 31 C0 7F 00 00 01 7F 00 00 01 94 88
82 A8 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F
45 C0 00 58 94 C3 00 00 40 01 E7 1F 7F 00 00 01 7F 00 00 01 03 03 DD BE 00
00 00 00 45 00 00 3C 85 F0 00 00 06 11 30 BF 7F 00 00 01 7F 00 00 01 A9 2B
82 A9 00 28 FE 3B 40 41 42 43 44 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52
53 54 55 56 57 58 59 5A 5B 5C 5D 5E 5F

```

Additional resources

Take a look at the following resources:

- Visit the documentation of the `net` standard Go package, which can be found at <https://golang.org/pkg/net/>. This is one of the biggest documentation pages found in the Go documentation.
- You can learn more about the `crypto/tls` package of the standard Go library at <https://golang.org/pkg/crypto/tls/>.
- Visit <https://golang.org/pkg/crypto/x509/> to learn more about the `crypto/x509` package.
- The ICMP protocol for IPv4 is defined in RFC 792. It can be found in many places, including <https://tools.ietf.org/html/rfc792>.
- **WebSocket** is a protocol for two-way communication between a client and a remote host. There is a WebSocket implementation for Go at <https://github.com/gorilla/websocket>. You can learn more about WebSocket at <http://www.rfc-editor.org/rfc/rfc6455.txt>.

- If you are really into network programming and you want to be able to work with raw TCP packets, you might find interesting and helpful information and tools in the `gopacket` library, which can be found at <https://github.com/google/gopacket>.
- The `raw` package, which is located at <https://github.com/mdlayher/raw>, allows you to read and write data at the device driver level for a network device.

Exercises

- Develop a **File Transfer Protocol (FTP)** client in Go.
- Next, try to develop an FTP server in Go. Is it more difficult to implement the FTP client or the FTP server? Why?
- Try to implement a Go version of the `nc(1)` utility. The secret when programming such fairly complex utilities is to start with a version that implements the basic functionality of the desired utility, before trying to support every possible option.
- Modify `TCPserver.go` so that it returns the date in one network packet and the time in another.
- Modify `TCPserver.go` so that it can serve multiple clients in a sequential way. Notice that this is not the same as being able to serve multiple requests concurrently. Put simply, use a `for` loop so that the `Accept()` call can be executed multiple times.
- TCP servers, such as `fibotcp.go`, tend to terminate when they receive a given signal, so add signal handling code to `fibotcp.go`, as you learned in Chapter 8, *Telling a UNIX System What to Do*.
- Modify `kvTCP.go` so that the `save()` function is protected using a `sync.Mutex`. Is this required?
- Try to connect to `https.go` and `TLSserver.go` using your favorite web browser.
- Try to put `https.go` into a Docker image and use it from there.
- Develop your own small web server in Go using a plain TCP implementation instead of using the `http.ListenAndServe()` function.

Summary

This chapter talked about many interesting things including developing UDP and TCP clients and servers, which are applications that work over TCP/IP computer networks.

The next chapter, which will be the last, will talk about machine learning and Go. It will include topics such as regression, classification, anomaly detection, and neural networks.

14

Machine Learning in Go

The previous two chapters discussed topics related to network programming, TCP/IP, HTTPS, RPC, and the `net` package. This chapter will talk about **machine learning** in Go, including many interesting topics such as calculating statistical properties, classification, regression, clustering, anomaly detection, neural networks, outlier analysis, and working with Apache Kafka. However, as all these are huge topics that deserve a book on their own, this chapter will only scratch the surface and give you a quick introduction to them, as well as introduce you to some handy Go packages that can help you to do the job.

Notice that each machine learning technique has some theory behind it – knowing the theory, the parameters, and the limitations of the techniques you are trying to use is essential for the success of your work. Additionally, visualizing your data can be helpful from time to time, as it allows you to get a good sense of your data quickly.



If you are really into machine learning with Go, then I suggest that you begin by getting *Machine Learning with Go* by Daniel Whitenack (Packt Publishing, 2017). If you want to learn about the theory behind machine learning, then you can begin by reading *An Introduction to Statistical Learning*, by Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani (Springer, 2013), and *The Elements of Statistical Learning, 2nd Edition*, by Trevor Hastie, Robert Tibshirani, and Jerome Friedman (Springer, 2009).

Therefore, in this chapter, you will learn about the following:

- Calculating simple statistical properties
- Regression
- Classification
- Anomaly detection
- Clustering
- Neural networks
- Working with TensorFlow

- **Outlier analysis**
- Working with Apache Kafka

Calculating simple statistical properties

Statistics is an area of mathematics that deals with the collection, analysis, interpretation, organization, and presentation of data. The field of statistics is divided into two main areas: the area of descriptive statistics, which tries to describe an already existing group of values, and the area of inferential statistics, which tries to predict upcoming values based on the information found in the current set of values.

Statistical learning is a branch of applied statistics that is related to machine learning.

Machine learning, which is closely related to computational statistics, is an area of computer science that tries to learn from data and make predictions about it without being specifically programmed to do so.



Statistical models try to interpret data as accurately as possible. However, the accuracy of a model might depend on external factors that might affect the data. So, you might have a weather forecasting model that could become totally inaccurate when there is a hurricane nearby.

In this section, you are going to learn how to calculate basic statistical properties such as the mean value, the minimum and the maximum values of your sample, the **median** value, and the **variance** of the sample. These values give you a good overview of your sample without going into too much detail. However, generic values that try to describe your sample can easily trick you by making you believe that you know your sample well without this being true.

All these statistical properties will be computed in `stats.go`, which will be presented in five parts. Each line of the input file contains a single number, which means that the input file is read line by line. Invalid input will be ignored without any warning messages.

Notice that input will be stored in a slice in order to use a separate function for calculating each property. Also, as you will see shortly, the values of the slice will be sorted before processing them.

The first part of `stats.go` is as follows:

```
package main

import (
    "bufio"
```

```
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)

func min(x []float64) float64 {
    return x[0]
}

func max(x []float64) float64 {
    return x[len(x)-1]
}
```

As the slice is sorted, it is easy to find the minimum and the maximum values in it. However, if the elements of the slice are not numeric, you might need to calculate the minimum and the maximum values in a different way that is related to your data.

The second part of `stats.go` contains the following Go code:

```
func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

This function calculates the mean value of your numeric data.

The third part of `stats.go` is as follows:

```
func medianValue(x []float64) float64 {
    length := len(x)
    if length%2 == 1 {
        // Odd
        return x[(length-1)/2]
    } else {
        // Even
        return (x[length/2] + x[(length/2)-1]) / 2
    }
    return 0
}
```

```
func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}
```

In order to be able to calculate the median value of a set, the set needs to be sorted.

The fourth part of `stats.go` contains the following code:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: stats filename\n")
        return
    }

    data := make([]float64, 0)

    file := flag.Args()[0]
    f, err := os.Open(file)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer f.Close()
```

The final part of `stats.go` is the following:

```
r := bufio.NewReader(f)
for {
    line, err := r.ReadString('\n')
    if err == io.EOF {
        break
    } else if err != nil {
        fmt.Printf("error reading file %s", err)
        break
    }
    line = strings.TrimRight(line, "\r\n")
    value, err := strconv.ParseFloat(line, 64)
    if err == nil {
        data = append(data, value)
    }
}
```

```
sort.Float64s(data)

fmt.Println("Min:", min(data))
fmt.Println("Max:", max(data))
fmt.Println("Mean:", meanValue(data))
fmt.Println("Median:", medianValue(data))
fmt.Println("Variance:", variance(data))
fmt.Println("Standard Deviation:", math.Sqrt(variance(data)))
}
```

In this part, you will start reading the input file. There are many ways to get your input – feel free to refer to [Chapter 8, *Telling a UNIX System What to Do*](#), in order to learn more about working with files in Go.

Notice that although it is not obligatory to sort the slice with the data before processing it, it saves you time in some calculations, so it is performed inside `main()`.

The last block of `fmt.Println()` statements prints the calculated statistical properties.

Executing `stats.go` will generate the following kind of output:

```
$ go run stats.go data.txt
Min: -2
Max: 3
Mean: 1.04
Median: 1.2
Variance: 2.8064
Standard Deviation: 1.6752313273097539
```

Notice that the contents of `data.txt` are as follows:

```
$ cat data.txt
1.2
-2
1
2.0
not valid
3
```

As you can see, each line contains a single value, which means that we are working with the simplest form of data. Although the manipulation of more complex data might be slightly different, the general idea remains the same.

The program of this section calculated every statistical property without using any external Go packages related to statistics. Most of the Go code that follows will use existing Go packages and will not implement everything from scratch.

Regression

Regression is a statistical method for calculating relationships among variables. This section will implement **linear regression**, which is the most popular and simplest regression technique and a very good way to understand your data. Note that regression techniques are not 100% accurate, even if you use higher-order (nonlinear) polynomials. The key with regression, as with most machine learning techniques, is to find a good enough technique and not the perfect technique and model.

Linear regression

The idea behind linear regression is simple: you are trying to model your data using a first-degree equation. A first-degree equation can be represented as $y = a x + b$.

There are many methods that allow you to find out that first-degree equation that will model your data – all techniques calculate a and b .

Implementing linear regression

The Go code of this section will be saved in `regression.go`, which is going to be presented in three parts. The output of the program will be two floating-point numbers that define a and b in the first-degree equation.

The first part of `regression.go` contains the following code:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/gonum/stat"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}
```

The `xy` structure is used to hold the data and should change according to your data format and values.

The second part of `regression.go` is as follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: regression filename\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
    defer file.Close()

    r := csv.NewReader(file)

    records, err := r.ReadAll()
    if err != nil {
        fmt.Println(err)
        return
    }
    size := len(records)

    data := xy{
        x: make([]float64, size),
        y: make([]float64, size),
    }
```

The final part of `regression.go` is as follows:

```
for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements")
        continue
    }

    if s, err := strconv.ParseFloat(v[0], 64); err == nil {
        data.y[i] = s
    }

    if s, err := strconv.ParseFloat(v[1], 64); err == nil {
        data.x[i] = s
    }
}
```

```
    }  
}  
  
b, a := stat.LinearRegression(data.x, data.y, nil, false)  
fmt.Printf("%.4v x + %.4v\n", a, b)  
fmt.Printf("a = %.4v b = %.4v\n", a, b)  
}
```

The data from the data file is read into the `data` variable. The function that implements the linear regression is `stat.LinearRegression()` and it returns two numbers, which are `b` and `a`, in that particular order.

At this point, it would be a good time to download the `gonum` package:

```
$ go get -u gonum.org/v1/gonum/stat
```

Executing `regression.go` with the input data stored in `reg_data.txt` will generate the following output:

```
$ go run regression.go reg_data.txt  
0.9463 x + -0.3985  
a = 0.9463 b = -0.3985
```

The two numbers returned are `a` and `b` from the $y = a x + b$ formula.

The contents of `reg_data.txt` are as follows:

```
$ cat reg_data.txt  
1,2  
3,4.0  
2.1,3  
4,4.2  
5,5.1  
-5,-5.1
```

Plotting data

It is now time to plot the results and the dataset in order to test how accurate the results from the linear regression technique are. For that purpose, we are going to use the Go code of `plotLR.go`, which will be presented in four parts. `plotLR.go` requires three command-line arguments, which are `a` and `b` from the $y = a x + b$ formula, and the file that contains the data points. The fact that `plotLR.go` does not calculate `a` and `b` on its own gives you the opportunity to experiment with `a` and `b` using your own values or values that were calculated by another utility.

The first part of `plotLR.go` is as follows:

```
package main

import (
    "encoding/csv"
    "flag"
    "fmt"
    "gonum.org/v1/plot"
    "gonum.org/v1/plot/plotter"
    "gonum.org/v1/plot/vg"
    "image/color"
    "os"
    "strconv"
)

type xy struct {
    x []float64
    y []float64
}

func (d xy) Len() int {
    return len(d.x)
}

func (d xy) XY(i int) (x, y float64) {
    x = d.x[i]
    y = d.y[i]
    return
}
```

The `Len()` and `XY()` methods are needed for the plotting part, whereas the `image/color` package is needed to change the colors in the output.

The second part of `plotLR.go` contains the following code:

```
func main() {
    flag.Parse()
    if len(flag.Args()) < 3 {
        fmt.Printf("usage: plotLR filename a b\n")
        return
    }

    filename := flag.Args()[0]
    file, err := os.Open(filename)
    if err != nil {
        fmt.Println(err)
        return
    }
}
```

```
}
defer file.Close()

r := csv.NewReader(file)

a, err := strconv.ParseFloat(flag.Args()[1], 64)
if err != nil {
    fmt.Println(a, "not a valid float!")
    return
}

b, err := strconv.ParseFloat(flag.Args()[2], 64)
if err != nil {
    fmt.Println(b, "not a valid float!")
    return
}

records, err := r.ReadAll()
if err != nil {
    fmt.Println(err)
    return
}
```

This part of the program works with the command-line arguments and the reading of the data.

The third part of `plotLR.go` is as follows:

```
size := len(records)

data := xy{
    x: make([]float64, size),
    y: make([]float64, size),
}

for i, v := range records {
    if len(v) != 2 {
        fmt.Println("Expected two elements per line!")
        return
    }

    s, err := strconv.ParseFloat(v[0], 64)
    if err == nil {
        data.y[i] = s
    }

    s, err = strconv.ParseFloat(v[1], 64)
```

```
        if err == nil {
            data.x[i] = s
        }
    }
}
```

The final part of `plotLR.go` is as follows:

```
line := plotter.NewFunction(func(x float64) float64 { return a*x + b })
line.Color = color.RGBA{B: 255, A: 255}

p, err := plot.New()
if err != nil {
    fmt.Println(err)
    return
}

plotter.DefaultLineStyle.Width = vg.Points(1)
plotter.DefaultGlyphStyle.Radius = vg.Points(2)

scatter, err := plotter.NewScatter(data)
if err != nil {
    fmt.Println(err)
    return
}
scatter.GlyphStyle.Color = color.RGBA{R: 255, B: 128, A: 255}

p.Add(scatter, line)

w, err := p.WriterTo(300, 300, "svg")
if err != nil {
    fmt.Println(err)
    return
}

_, err = w.WriteTo(os.Stdout)
if err != nil {
    fmt.Println(err)
    return
}
}
```

The function that is going to be plotted is defined using the `plotter.NewFunction()` method.

At this point, you should download some external packages by executing the following commands:

```
$ go get -u gonum.org/v1/plot
```

```
$ go get -u gonum.org/v1/plot/plotter
$ go get -u gonum.org/v1/plot/vg
```

Executing `plotLR.go` will generate the following kind of output:

```
$ go run plotLR.go reg_data.txt
usage: plotLR filename a b
$ go run plotLR.go reg_data.txt 0.9463 -0.3985
<?xml version="1.0"?>
<!-- Generated by SVGo and Plotinum VG -->
<svg width="300pt" height="300pt" viewBox="0 0 300 300"
      xmlns="http://www.w3.org/2000/svg"
      xmlns:xlink="http://www.w3.org/1999/xlink">
  <g transform="scale(1, -1) translate(0, -300)">
    .
    .
    .
```

Therefore, you should save the generated output in a file before using it:

```
$ go run plotLR.go reg_data.txt 0.9463 -0.3985 > output.svg
```

As the output is in the **Scalable Vector Graphics (SVG)** format, you should load it into a web browser in order to see the results. The results from our data can be seen in the following figure.

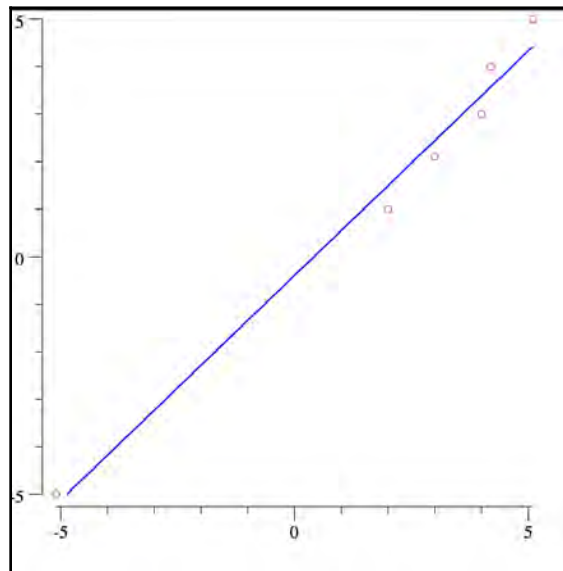


Figure 14.1: The output of the `plotLR.go` program

The output of the image will also show how accurately the data can be modeled using a linear equation.

Classification

In statistics and machine learning, **classification** is the process of putting elements into existing sets that are called categories. In machine learning, classification is considered a **supervised learning** technique, which is where a set that is considered to contain correctly identified observations is used for training before working with the actual data.

A very popular and easy-to-implement classification method is called **k-nearest neighbors (k-NN)**. The idea behind k-NN is that we can classify data items based on their similarity with other items. The *k* in k-NN denotes the number of neighbors that are going to be included in the decision, which means that *k* is a positive integer that is usually pretty small.

The input of the algorithm consists of the *k*-closest training examples in the feature space. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class that is the most common among its *k*-NN. If the value of *k* is 1, then the element is simply assigned to the class that is the nearest neighbor according to the **distance metric** used. The distance metric depends on the data you are dealing with. As an example, you will need a different distance metric when working with complex numbers and another when working with points in three-dimensional space.

The Go code that will illustrate classification in Go can be found in `classify.go`. The file is going to be presented in three parts.

The first part of `classify.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "strconv"

    "github.com/sjwhitworth/golearn/base"
    "github.com/sjwhitworth/golearn/evaluation"
    "github.com/sjwhitworth/golearn/knn"
)
```

The second part of `classify.go` contains the following Go code:

```
func main() {
    flag.Parse()
    if len(flag.Args()) < 2 {
        fmt.Printf("usage: classify filename k\n")
        return
    }

    dataset := flag.Args()[0]
    rawData, err := base.ParseCSVToInstances(dataset, false)
    if err != nil {
        fmt.Println(err)
        return
    }

    k, err := strconv.Atoi(flag.Args()[1])
    if err != nil {
        fmt.Println(err)
        return
    }

    cls := knn.NewKnnClassifier("euclidean", "linear", k)
```

The `knn.NewKnnClassifier()` method returns a new **classifier**. The last parameter of the function is the number of neighbors that the classifier will have.

The final part of `classify.go` is as follows:

```
    train, test := base.InstancesTrainTestSplit(rawData, 0.50)
    cls.Fit(train)

    p, err := cls.Predict(test)
    if err != nil {
        fmt.Println(err)
        return
    }

    confusionMat, err := evaluation.GetConfusionMatrix(test, p)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(evaluation.GetSummary(confusionMat))
}
```

The `Fit()` method stores the training data in case you want to use it later on, whereas the `Predict()` method returns a classification for the input based on the training data using the k-NN algorithm. Finally, the `evaluation.GetSummary()` method returns a table of precision, recall, true positive, false positive, and true negative values for each class for the given `ConfusionMatrix`, which was calculated using the call to `evaluation.GetConfusionMatrix()`.



As `base.InstancesTrainTestSplit()` will not return the same values all the time, the training and test data will be different each time you execute `classify.go`, which means that you will get different results.

At this point, you should install the `github.com/sjwhitworth/golearn` package as follows:

```
$ go get -t -u -v github.com/sjwhitworth/golearn
github.com/sjwhitworth/golearn (download)
github.com/sjwhitworth/golearn
$ cd ~/go/src/github.com/sjwhitworth/golearn
$ go get -t -u -v ./...
```

Now, you should return back to your previous directory in order to execute `classify.go` from there. Executing `classify.go` will create the following kind of output:

```
$ go run classify.go class_data.txt 2
Reference Class True Positives False Positives True Negatives Precision
Recall F1 Score
-----
Iris-versicolor 25 0 41 1.0000 0.9259
0.9615
Iris-virginica 5 2 61 0.7143 1.0000
0.8333
Iris-setosa 36 0 32 1.0000 1.0000
1.0000
Overall accuracy: 0.9706
$ go run classify.go class_data.txt 30
Reference Class True Positives False Positives True Negatives Precision
Recall F1 Score
-----
Iris-versicolor 27 5 36 0.8438 1.0000
0.9153
Iris-virginica 0 0 63 NaN 0.0000 NaN
Iris-setosa 36 0 32 1.0000 1.0000
1.0000
Overall accuracy: 0.9265
```

The contents of `class_data.txt`, which are pretty simple, have the following format:

```
$ head -4 class_data.txt
6.7,3.1,5.6,2.4,Iris-virginica
6.9,3.1,5.1,2.3,Iris-virginica
5.8,2.7,5.1,1.9,Iris-virginica
6.8,3.2,5.9,2.3,Iris-virginica
```

The data is taken from the **Iris dataset**.

The precision of the algorithm is high because the number of elements in `class_data.txt` is pretty small. As `golearn` includes some sample data, I can try using the entire Iris dataset on my personal macOS Mojave machine as follows:

```
$ go run classify.go
~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 2
Reference ClassnTrue Positives False Positives True Negatives Precision
Recall F1 Score
-----
Iris-setosa 30 0 58 1.0000 1.0000
1.0000
Iris-virginica 28 3 56 0.9032 0.9655
0.9333
Iris-versicolor 26 1 58 0.9630 0.8966
0.9286
Overall accuracy: 0.9545
$ go run classify.go
~/go/src/github.com/sjwhitworth/golearn/examples/datasets/iris.csv 50
Reference ClassnTrue Positives False Positives True Negatives Precision
Recall F1 Score
-----
Iris-setosa 0 0 58 NaN 0.0000 NaN
Iris-virginica 4 5 54 0.4444 0.1379
0.2105
Iris-versicolor 24 55 4 0.3038 0.8276
0.4444
Overall accuracy: 0.3182
```

If you work with the Iris dataset and try different values for the number of neighbors, you will find out that the larger the number of neighbors you want to use, the smaller the accuracy of the results. Unfortunately, further discussion of the k-NN algorithm is beyond the scope of this book.

Clustering

Clustering is the unsupervised version of classification where the grouping of data into categories is based on some metric of similarity or distance. This section will use **k-means clustering**, which is the most famous clustering technique and one that is also easy to implement. Once again, we are going to use an external library that can be found at <https://github.com/mash/gokmeans>.

The utility that showcases clustering in Go is called `cluster.go`, and it is going to be presented in three parts. The utility requires one command-line argument, which is the number of clusters that are going to be created.

The first part of `cluster.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "github.com/mash/gokmeans"
    "strconv"
)

var observations []gokmeans.Node = []gokmeans.Node{
    gokmeans.Node{4},
    gokmeans.Node{5},
    gokmeans.Node{6},
    gokmeans.Node{8},
    gokmeans.Node{10},
    gokmeans.Node{12},
    gokmeans.Node{15},
    gokmeans.Node{0},
    gokmeans.Node{-1},
}
```

This time, the data is included in the program for reasons of simplicity. However, nothing prohibits you from reading it from one or more external files.

The second part of `cluster.go` contains the following Go code:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: cluster k\n")
        return
    }
}
```

```

k, err := strconv.Atoi(flag.Args()[0])
if err != nil {
    fmt.Println(err)
    return
}

```

The final part of `cluster.go` is as follows:

```

if success, centroids := gokmeans.Train(observations, k, 50); success {
    fmt.Println("The centroids are the following:")
    for _, centroid := range centroids {
        fmt.Println(centroid)
    }

    fmt.Println("The clusters are the following:")
    for _, observation := range observations {
        index := gokmeans.Nearest(observation, centroids)
        fmt.Println(observation, "belongs in cluster", index+1, ".")
    }
}
}

```

At this point, you will need to get the external Go library by executing the following command:

```

$ go get -v -u github.com/mash/gokmeans
github.com/mash/gokmeans (download)
github.com/mash/gokmeans

```

Executing `cluster.go` when you want to have a single cluster will produce the following kind of output:

```

$ go run cluster.go 1
The centroids are the following:
[6.555555555555555]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 1 .
[10] belongs in cluster 1 .
[12] belongs in cluster 1 .
[15] belongs in cluster 1 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 1 .

```

As you have a single cluster, each item will belong to that cluster, hence the output returned by `cluster.go`.

If you change the value of k , you will get the following output:

```
$ go run cluster.go 5
The centroids are the following:
[5]
[-0.5]
[13.5]
[10]
[8]
The clusters are the following:
[4] belongs in cluster 1 .
[5] belongs in cluster 1 .
[6] belongs in cluster 1 .
[8] belongs in cluster 5 .
[10] belongs in cluster 4 .
[12] belongs in cluster 3 .
[15] belongs in cluster 3 .
[0] belongs in cluster 2 .
[-1] belongs in cluster 2 .
$ go run cluster.go 8
The centroids are the following:
[0]
[4.5]
[-1]
[9]
[-1]
[6]
[12]
[15]
The clusters are the following:
[4] belongs in cluster 2 .
[5] belongs in cluster 2 .
[6] belongs in cluster 6 .
[8] belongs in cluster 4 .
[10] belongs in cluster 4 .
[12] belongs in cluster 7 .
[15] belongs in cluster 8 .
[0] belongs in cluster 1 .
[-1] belongs in cluster 3 .
```

Anomaly detection

Anomaly detection techniques try to find the probability that a given set contains anomalous behavior, which can be unusual values or patterns.

The utility that is going to be developed in this section is called `anomaly.go`, and it is going to be presented in three parts. The utility uses probabilistic anomaly detection with the help of the **Anomalyzer** package and calculates the probability that the given set of numeric values contains anomalous behavior.

The first part of `anomaly.go` is as follows:

```
package main

import (
    "flag"
    "fmt"
    "math/rand"
    "strconv"
    "time"

    "github.com/lytics/anomalyzer"
)

func random(min, max int) int {
    return rand.Intn(max-min) + min
}
```

The second part of `anomaly.go` is as follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) == 0 {
        fmt.Printf("usage: anomaly MAX\n")
        return
    }

    MAX, err := strconv.Atoi(flag.Args()[0])
    if err != nil {
        fmt.Println(err)
        return
    }

    conf := &anomalyzer.AnomalyzerConf{
        Sensitivity: 0.1,
        UpperBound:  5,
        LowerBound:  anomalyzer.NA,
        ActiveSize:  1,
        NSeasons:    4,
        Methods:     []string{"diff", "fence", "magnitude", "ks"},
    }
```

The `anomalyzer` package supports the `cdf`, `diff`, `high rank`, `low rank`, `magnitude`, `fence`, and `bootstrap ks` algorithmic tests. If you decide to use some or all of them as the value of `Methods`, then `anomalyzer` will execute all of them. As each one of them returns a probability of anomalous behavior, the `anomalyzer` package will compute a weighted mean in order to establish whether the current dataset has anomalous behavior.

The `Sensitivity` field is for the magnitude test – its default value is `0.1`. The `UpperBound` and `LowerBound` fields are used by the fence test. On the other hand, the presence of the `ActiveSize` field is compulsory – its value should be at least `1`. Lastly, the `NSeasons` field has a default value of `4` if it is not defined.

The third part of `anomaly.go` contains the following Go code:

```
data := []float64{}
SEED := time.Now().Unix()
rand.Seed(SEED)

for i := 0; i < MAX; i++ {
    data = append(data, float64(random(0, MAX)))
}
fmt.Println("data:", data)
```

The `anomaly.go` code generates random data on its own. The number of elements is defined as a command-line argument to the program.

The final part of `anomaly.go` is as follows:

```
anom, _ := anomalyzer.NewAnomalyzer(conf, data)
prob := anom.Push(8.0)
fmt.Println("Anomalous Probability:", prob)
}
```

At this point, you should download the external Go package as follows:

```
$ go get -v -u github.com/lytics/anomalyzer
github.com/lytics/anomalyzer (download)
github.com/drewlanenga/govector (download)
github.com/drewlanenga/govector
github.com/lytics/anomalyzer
```

Executing `anomaly.go` will generate the following kind of output:

```
$ go run anomaly.go 20
data: [18 3 2 19 2 16 5 15 3 14 2 9 11 10 2 17 17 14 19 1]
Anomalous Probability: 0.8612730015082957
$ go run anomaly.go 20
```

```

data: [17 8 19 10 0 14 12 7 7 13 2 5 18 1 15 4 0 14 13 9]
Anomalous Probability: 0.7885470085470085
$ go run anomaly.go 100
data: [85 5 64 32 69 55 0 67 11 96 75 92 25 54 2 49 58 6 16 38 55 11 93 90
90 47 66 97 37 61 85 92 15 45 33 43 61 44 73 18 10 86 17 15 67 28 26 7 25
76 79 51 9 32 70 99 9 39 6 25 10 57 50 84 20 67 42 89 0 1 8 96 49 6 20 33
57 18 48 84 53 98 51 84 41 97 69 62 11 44 21 13 90 25 52 85 48 27 90 20]
Anomalous Probability: 0.8977395577395577

```

Neural networks

Neural networks, which try to work like the human brain, learn to perform tasks based on given examples. Neural network have layers, and the smallest neural network must have at least two layers: input and output. During the training phase, data flows through the layers of the neural network. The actual output values of the training data are used to correct the calculated output values of the training data so that the next iteration will be more precise.

The utility that will be developed in this section is named `neural.go`, and it will implement a really simple neural network. This is going to be presented in four parts.

The first part of `neural.go` is as follows:

```

package main

import (
    "fmt"
    "math/rand"
    "time"

    "github.com/goml/gobrain"
)

```

The new line in the `import` list tells the `gofmt` tool to sort the package names in blocks that are separated by new lines. The second part of `neural.go` contains the following Go code:

```

func main() {
    seed := time.Now().Unix()
    rand.Seed(seed)

    patterns := [][][]float64{
        {{0, 0, 0, 0}, {0}},
        {{0, 1, 0, 1}, {1}},
        {{1, 0, 1, 0}, {1}},
        {{1, 1, 1, 1}, {1}},
    }
}

```

The `patterns` slice holds the training data that will be used later. The `rand.Seed()` function initializes a new random-number generator that is automatically used by the `github.com/goml/gobrain` package.

The third part of `neural.go` is as follows:

```
ff := &gobrain.FeedForward{}
ff.Init(4, 2, 1)
ff.Train(patterns, 1000, 0.6, 0.4, false)
```

In this part, we initialize the neural network. The first parameter of the `Init()` method is the number of inputs, the second is the number of hidden nodes, and the third is the number of outputs. The dimension and the data of the `patterns` slice should be in compliance with the values in `Init()`, and vice versa.

The final part of `neural.go` is as follows:

```
in := []float64{1, 1, 0, 1}
out := ff.Update(in)
fmt.Println(out)

in = []float64{0, 0, 0, 0}
out = ff.Update(in)
fmt.Println(out)
}
```

There are two tests happening here. For the first test, the input is `{1, 1, 0, 1}`, whereas for the second test, the input is `{0, 0, 0, 0}`.

You should know by now which Go package you will need to download using `go get`, so please download it before trying to run `neural.go`.

Executing `neural.go` will generate the following kind of output:

```
$ go run neural.go
[0.9918648920317314]
[0.02826477691747802]
```

The first value is close to 1, whereas the second value is close to 0.

As there is a kind of randomness introduced, executing `neural.go` multiple times will generate slightly different results:

```
$ go run neural.go
[0.9920127780655835]
[0.028029429851140687]
```

```
go run neural.go
[0.9913803776914417]
[0.028875009295811015]
```

Outlier analysis

Outlier analysis is about finding the values that look like they do not belong with the rest of the values. Put simply, outliers are extreme values that greatly differ from the other observations. A good book on outlier analysis is *Outlier Analysis, 2nd Edition*, by Charu C. Aggarwal (Springer, 2017).

The outlier technique that is going to be implemented in `outlier.go`, which is going to be presented in four parts, is based on standard deviation. You will learn more about this technique later.

The first part of `outlier.go` is as follows:

```
package main

import (
    "bufio"
    "flag"
    "fmt"
    "io"
    "math"
    "os"
    "sort"
    "strconv"
    "strings"
)
```

The technique that is going to be implemented requires packages of the standard Go library only.

The second part of `outlier.go` contains the following Go code:

```
func variance(x []float64) float64 {
    mean := meanValue(x)
    sum := float64(0)
    for _, v := range x {
        sum = sum + (v-mean)*(v-mean)
    }
    return sum / float64(len(x))
}
```



```
func meanValue(x []float64) float64 {
    sum := float64(0)
    for _, v := range x {
        sum = sum + v
    }
    return sum / float64(len(x))
}
```

These two functions were used previously in `stats.go`. If you find yourself using the same functions all the time, it would be good to create one or more Go libraries and group your Go code.

The third part of `outlier.go` contains the following Go code:

```
func outliers(x []float64, limit float64) []float64 {
    deviation := math.Sqrt(variance(x))
    mean := meanValue(x)
    anomaly := deviation * limit
    lower_limit := mean - anomaly
    upper_limit := mean + anomaly
    fmt.Println(lower_limit, upper_limit)

    y := make([]float64, 0)
    for _, val := range x {
        if val < lower_limit || val > upper_limit {
            y = append(y, val)
        }
    }

    return y
}
```

This function holds the logic of the program. It calculates the standard deviation and the mean value of the sample in order to compute the upper and lower limits. Everything outside these two limits will be considered an outlier.

The final part of `outlier.go` is as follows:

```
func main() {
    flag.Parse()
    if len(flag.Args()) != 2 {
        fmt.Printf("usage: stats filename limit\n")
        return
    }

    file := flag.Args()[0]
    f, err := os.Open(file)
    if err != nil {
```

```

        fmt.Println(err)
        return
    }
    defer f.Close()

    limit, err := strconv.ParseFloat(flag.Args()[1], 64)
    if err != nil {
        fmt.Println(err)
        return
    }

    data := make([]float64, 0)
    r := bufio.NewReader(f)
    for {
        line, err := r.ReadString('\n')
        if err == io.EOF {
            break
        } else if err != nil {
            fmt.Printf("error reading file %s", err)
            break
        }
        line = strings.TrimRight(line, "\r\n")
        value, err := strconv.ParseFloat(line, 64)
        if err == nil {
            data = append(data, value)
        }
    }

    sort.Float64s(data)
    out := outliers(data, limit)
    fmt.Println(out)
}

```

The `main()` function deals with command-line arguments and the reading of the input file before calling the `outliers()` function.

Executing `outlier.go` will produce the following kind of output:

```

$ go run outlier.go data.txt 2
-94.21189713007178 95.36745268562734
[-100 100]
$ go run outlier.go data.txt 5
-236.3964094918461 237.55196504740167
[]
$ go run outlier.go data.txt 0.02
-0.3701189713007176 1.5256745268562737
[-100 -10 -2 2 3 10 100]

```

If you lower the value of the `limit` variable, you will find more outliers in your data. However, this depends on the problem you are trying to solve.

The contents of `data.txt` are the following:

```
$ cat data.txt
1.2
-2
1
2.0
not valid
3
10
100
-10
-100
```

Working with TensorFlow

TensorFlow is a rather famous open-source platform for machine learning. In order to use TensorFlow with Go, you will first need to download a Go package:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
```

However, for the aforementioned command to work, the C interface for TensorFlow should be already installed. On a macOS Mojave machine, this can be installed as follows:

```
$ brew install tensorflow
```

If the C interface is not installed, and you try to install the Go package for TensorFlow, you will get the following error message:

```
$ go get github.com/tensorflow/tensorflow/tensorflow/go
# github.com/tensorflow/tensorflow/tensorflow/go
ld: library not found for -ltensorflow
clang: error: linker command failed with exit code 1 (use -v to see invocation)
```

As TensorFlow is pretty complex, it would be good to execute the following command in order to validate your installation:

```
$ go test github.com/tensorflow/tensorflow/tensorflow/go
ok      github.com/tensorflow/tensorflow/tensorflow/go    0.109s
```

Apart from the Go tests, you can also execute the following Go program, which will print the version of the Go TensorFlow package that you are using:

```
package main

import (
    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
    "fmt"
)

func main() {
    s := op.NewScope()
    c := op.Const(s, "Using TensorFlow version: " + tf.Version())
    graph, err := s.Finalize()

    if err != nil {
        fmt.Println(err)
        return
    }

    sess, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    output, err := sess.Run(nil, []tf.Output{c}, nil)
    if err != nil {
        fmt.Println(err)
        return
    }

    fmt.Println(output[0].Value())
}
```

If you save that program as `tfVersion.go` and execute it, you will get the following output:

```
$ go run tfVersion.go
2019-06-10 22:30:12.880532: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Using TensorFlow version: 1.13.1
```

The first message is a warning generated by the TensorFlow Go package and you can ignore it at the moment.

We are now ready to continue and I will show you a real TensorFlow program that will illustrate how TensorFlow works. The related code is saved in `tFlow.go` and it is going to be presented in four parts. What the program does is add and multiply two numbers that are given as command-line arguments to the program.

The first part of `tFlow.go` is the following:

```
package main

import (
    "fmt"
    "os"
    "strconv"

    tf "github.com/tensorflow/tensorflow/tensorflow/go"
    "github.com/tensorflow/tensorflow/tensorflow/go/op"
)
```

The second part of `tFlow.go` contains the following code:

```
func Add(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("a1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("a2"), tf.Int8)
    sum_result_node := op.Add(sum_scope, input1, input2)

    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    a2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
}
```

```
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
            input1: a1,
            input2: a2,
        },
        []tf.Output{sum_result_node}, nil)

    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    return sum[0].Value(), nil
}
```

The `Add()` function adds two `int8` numbers. The code is pretty big for just adding two numbers, which illustrates that TensorFlow has a certain way of working. This mainly happens because TensorFlow is an advanced environment with many capabilities.

The third part of `tFlow.go` is as follows:

```
func Multiply(sum_arg1, sum_arg2 int8) (interface{}, error) {
    sum_scope := op.NewScope()
    input1 := op.Placeholder(sum_scope.SubScope("x1"), tf.Int8)
    input2 := op.Placeholder(sum_scope.SubScope("x2"), tf.Int8)

    sum_result_node := op.Mul(sum_scope, input1, input2)
    graph, err := sum_scope.Finalize()
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x1, err := tf.NewTensor(sum_arg1)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    x2, err := tf.NewTensor(sum_arg2)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
}
```

```
    session, err := tf.NewSession(graph, nil)
    if err != nil {
        fmt.Println(err)
        return 0, err
    }
    defer session.Close()

    sum, err := session.Run(
        map[tf.Output]*tf.Tensor{
            input1: x1,
            input2: x2,
        },
        []tf.Output{sum_result_node}, nil)

    if err != nil {
        fmt.Println(err)
        return 0, err
    }

    return sum[0].Value(), nil
}
```

The `Multiply()` function multiplies two `int8` values and returns the result.

The final part of `tFlow.go` is the following:

```
func main() {
    if len(os.Args) != 3 {
        fmt.Println("Need two integer parameters!")
        return
    }

    t1, err := strconv.Atoi(os.Args[1])
    if err != nil {
        fmt.Println(err)
        return
    }
    n1 := int8(t1)

    t2, err := strconv.Atoi(os.Args[2])
    if err != nil {
        fmt.Println(err)
        return
    }
    n2 := int8(t2)

    res, err := Add(n1, n2)
    if err != nil {
```

```
    fmt.Println(err)
  } else {
    fmt.Println("Add:", res)
  }

  res, err = Multiply(n1, n2)
  if err != nil {
    fmt.Println(err)
  } else {
    fmt.Println("Multiply:", res)
  }
}
```

Executing `tFlow.go` will generate the following kind of output:

```
$ go run tFlow.go 1 20
2019-06-14 18:46:52.115676: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Add: 21
Multiply: 20
$ go run tFlow.go -2 20
2019-06-14 18:47:23.104918: I
tensorflow/core/platform/cpu_feature_guard.cc:141] Your CPU supports
instructions that this TensorFlow binary was not compiled to use: AVX2 FMA
Add: 18
Multiply: -40
```

The warning message that appears on the output tells you that TensorFlow is not as fast as it can be on your machine. Basically, it informs you that you will need to compile TensorFlow from scratch in order to resolve that warning message.

Talking to Kafka

In this section, you will learn how to write and read JSON records using **Kafka**. Its name was inspired by the author Franz Kafka, because Kafka software is optimized for writing. Kafka is written in **Scala** and **Java**. It was originally developed by LinkedIn and donated to the Apache Software Foundation back in 2011. Kafka's design was influenced by transaction logs.

The main advantage of Kafka is that it can be used to store lots of data fast, which might interest you when you have to work with huge amounts of real-time data. However, its disadvantage is that in order to maintain that speed, the data is read-only and stored in a naive way.

The following program, which is named `writeKafka.go`, will illustrate how to write data to Kafka. In Kafka terminology, `writeKafka.go` is a **producer**. The utility will be presented in three parts.

The first part of `writeKafka.go` is as follows:

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/segmentio/kafka-go"
    "math/rand"
    "os"
    "strconv"
    "time"
)
```

The Kafka Go driver requires the use of an external package that you will need to download on your own.

The second part of `writeKafka.go` contains the following Go code:

```
type Record struct {
    Name    string `json:"name"`
    Random  int    `json:"random"`
}

func random(min, max int) int {
    return rand.Intn(max-min) + min
}

func main() {
    MIN := 0
    MAX := 0
    TOTAL := 0
    topic := ""
    if len(os.Args) > 4 {
        MIN, _ = strconv.Atoi(os.Args[1])
        MAX, _ = strconv.Atoi(os.Args[2])
        TOTAL, _ = strconv.Atoi(os.Args[3])
        topic = os.Args[4]
    } else {
        fmt.Println("Usage:", os.Args[0], "MIN MAX TOTAL TOPIC")
        return
    }
}
```

The `Record` structure is used to store the data that will be written to the desired Kafka topic.

The third part of `writeKafka.go` contains the following Go code:

```
partition := 0
conn, err := kafka.DialLeader(context.Background(), "tcp",
"localhost:9092", topic, partition)
if err != nil {
    fmt.Printf("%s\n", err)
    return
}

rand.Seed(time.Now().Unix())
```

In this part of the program, you define the address (`localhost:9092`) of the Kafka server.

The final part of `writeKafka.go` is as follows:

```
for i := 0; i < TOTAL; i++ {
    myrand := random(MIN, MAX)
    temp := Record{strconv.Itoa(i), myrand}
    recordJSON, _ := json.Marshal(temp)

    conn.SetWriteDeadline(time.Now().Add(1 * time.Second))
    conn.WriteMessages(
        kafka.Message{Value: []byte(recordJSON)},
    )

    if i%50 == 0 {
        fmt.Print(".")
    }
    time.Sleep(10 * time.Millisecond)
}

fmt.Println()
conn.Close()
}
```

The following Go program, which is named `readKafka.go`, will illustrate how to write data to Kafka. Programs that read data from Kafka are called **consumers** in Kafka terminology. The Go code of `readKafka.go` will be presented in four parts.

The first part of `readKafka.go` is as follows:

```
package main
```

```
import (  
    "context"  
    "encoding/json"  
    "fmt"  
    "github.com/segmentio/kafka-go"  
    "os"  
)
```

The `readKafka.go` utility also requires an external Go package.

The second part of `readKafka.go` is as follows:

```
type Record struct {  
    Name    string `json:"name"`  
    Random int    `json:"random"`  
}  
  
func main() {  
    if len(os.Args) < 2 {  
        fmt.Println("Need a Kafka topic name.")  
        return  
    }  
  
    partition := 0  
    topic := os.Args[1]  
    fmt.Println("Kafka topic:", topic)
```

This part of the program defines the Go structure that will hold the Kafka records and deals with the command-line arguments of the program.

The third part of `readKafka.go` contains the following Go code:

```
r := kafka.NewReader(kafka.ReaderConfig{  
    Brokers:    []string{"localhost:9092"},  
    Topic:      topic,  
    Partition:  partition,  
    MinBytes:   10e3,  
    MaxBytes:   10e6,  
})  
r.SetOffset(0)
```

The `kafka.NewReader()` structure holds the data required to connect to the Kafka server process and Kafka topic.

The final code segment from `readKafka.go` is the following:

```
for {  
    m, err := r.ReadMessage(context.Background())
```

```

        if err != nil {
            break
        }
        fmt.Printf("message at offset %d: %s = %s\n", m.Offset,
            string(m.Key), string(m.Value))

        temp := Record{}
        err = json.Unmarshal(m.Value, &temp)
        if err != nil {
            fmt.Println(err)
        }
        fmt.Printf("%T\n", temp)
    }

    r.Close()
}

```

This `for` loop reads the data from the desired Kafka topic and prints it on the screen. Note that the program will keep waiting for data and will never end.

For the purposes of this section, we are going to use a Docker image with Kafka in order to execute both `readKafka.go` and `writeKafka.go`. So, first, we will need to execute the following command to download the desired Kafka image, if it is not already on our local machine:

```
$ docker pull landoop/fast-data-dev:latest
```

The output of the `docker images` command will be able to verify that the desired Kafka Docker image is there. Then, you should execute that image and run it as a **container**, as follows:

```

$ docker run --rm --name=kafka-box -it -p 2181:2181 -p 3030:3030 -p
8081:8081 -p 8082:8082 -p 8083:8083 -p 9092:9092 -p 9581:9581 -p 9582:9582
-p 9583:9583 -p 9584:9584 -e ADV_HOST=127.0.0.1 landoop/fast-data-
dev:latest
Setting advertised host to 127.0.0.1.
Starting services.
This is Landoop's fast-data-dev. Kafka 2.0.1-L0 (Landoop's Kafka
Distribution).
You may visit http://127.0.0.1:3030 in about a minute.
.
.
.

```

For the presented utilities to work, you will need to download the Go package that allows you to communicate with Kafka.

This can be done as follows:

```
$ go get -u github.com/segmentio/kafka-go
```

After that, you are ready to use the Docker image of Kafka and execute the two Go utilities. Executing `writeKafka.go` will generate the following kind of output:

```
$ go run writeKafka.go 1 1000 500 my_topic
.....
$ go run writeKafka.go 1 1000 500 my_topic
.....
```

Executing `readKafka.go` will produce the following output:

```
$ go run readKafka.go my_topic | head
Kafka topic: my_topic
message at offset 0: = {"name":"0","random":134}
main.Record
message at offset 1: = {"name":"1","random":27}
main.Record
message at offset 2: = {"name":"2","random":168}
main.Record
message at offset 3: = {"name":"3","random":317}
main.Record
message at offset 4: = {"name":"4","random":455}
signal: broken pipe
```

Additional resources

Have a look at the following resources:

- You can learn more about Kafka by visiting <https://kafka.apache.org/>.
- You can learn more about TensorFlow at <https://www.tensorflow.org/>.
- **Lenses** is a great product for working with Kafka and Kafka records. You can learn more about Lenses at <https://lenses.io/>.
- You can find the documentation page of the Go TensorFlow package at <https://godoc.org/github.com/tensorflow/tensorflow/tensorflow/go>.
- You can learn more about the Iris dataset at <https://archive.ics.uci.edu/ml/datasets/iris>.
- Have a look at <https://machinebox.io/>, which is machine learning software for people without a science degree, is free for developers, and works using Docker images.

- The documentation page for the Anomalyzer Go package can be found at <https://github.com/lytics/anomalyzer>.

Exercises

- Develop your own Kafka producer that will write JSON records with three fields to a Kafka topic.
- A very interesting statistical property is **covariance**. Find its formula and implement it in Go.
- Change the code of `stats.go` in order to work with integer values only.
- Modify `cluster.go` in order to get the data from an external file that will be given as a command-line argument to the program.
- Change the code of `outlier.go` in order to divide the input into two slices and work with each one of these slices.
- Change the code of `outlier.go` in order to accept the upper and lower limits from the user without calculating them.
- If you find TensorFlow difficult to use, you can try `tfgo`, which can be found at <https://github.com/galeone/tfgo>.

Summary

This chapter talked about many interesting things related to machine learning, including regression, anomaly detection, classification, clustering, and outliers. Go is a great and robust programming language for use in machine learning areas, so feel free to include it in your machine learning projects.

Where to go next?

Philosophically speaking, no programming book can ever be perfect, and neither is this book! Did I leave some Go topics out? Absolutely, yes! Why? Because there are always more topics that could be covered in a book, so if I tried to cover everything, the book would never be ready for publication. This situation is somehow analogous to the specifications of a program: you can always add new and exciting features, but if you do not freeze its specifications, the program will always be under development and will never be ready.

The good thing is that after reading this book, you will be ready to learn on your own, which is the biggest benefit you can get from any good computer book on programming. The main purpose of this book is to help you to learn how to program in Go and gain some experience. However, there is no substitute for trying things on your own and failing often because the only way to learn a programming language is to keep developing non-trivial things. You are now ready to start writing your own software in Go and learning new things.

I would like to congratulate you and thank you for choosing this book, with the hope that you found it useful and will continue to use it as a reference. Go is a great programming language that I believe you will not regret learning. This is the end of another Go book for me, but just the beginning of the journey for you!

Soli Deo gloria

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:



Mastering Go

Mihalis Tsoukalos

ISBN: 978-1-78862-654-5

- Understand the design choices of Golang syntax
- Know enough Go internals to be able to optimize Golang code
- Appreciate concurrency models available in Golang
- Understand the interplay of systems and networking code
- Write server-level code that plays well in all environments
- Understand the context and appropriate use of Go data types and data structures



Go Systems Programming

Mihalis Tsoukalos

ISBN: 978-1-78712-564-3

- Explore the Go language from the standpoint of a developer conversant with Unix, Linux, and so on
- Understand Goroutines, the lightweight threads used for systems and concurrent applications
- Learn how to translate Unix and Linux systems code in C to Golang code
- How to write fast and lightweight server code
- Dive into concurrency with Go
- Write low-level networking code

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!

Index

▪

.gitignore file 360

A

abstract syntax tree (AST) 307

abstract types 335

Alef language 9

algorithm complexity 211

anomaly detection 733, 735

Anomalyzer package 734

anonymous function 78, 273, 419, 442

Apache 147, 616

Apache HTTP server benchmarking tool 626

array dimensions

finding 264

arrays 115

assembly

about 588

used, with Go code 588, 589, 590

atomic operation 495

atomic package

about 495, 496, 497

using 620, 621, 622

automated testing 549

B

balanced tree 212

benchmark function

defining 576

benchmarking

about 66

buffered writing 577, 578, 579, 580

example 570, 571, 572, 573, 574, 575, 576

Go code 569, 570

big endian byte 400

binary formats

advantages 402, 403

binary tree

advantages 215

implementing, in Go 213, 214, 215

in Go 212

black set 58

black-box testing 564

blank identifier 26

branches

working with 358, 359

buffered channel 476, 477, 478

buffered writing

benchmarking 577, 578, 579, 580

bufio package 392

byte slice 123, 163, 171, 400, 414

bytes package

about 414, 415, 416

reference link 591

C

C code

about 72, 76, 77

calling, from Go 70

calling, from Go with same file 70, 71

calling, from Go with separate files 71

mixing, with Go 74

C language 8

C shared library 75

C strings 171

C++ 353, 542

cancellation 508

Candy 616

cardinality 113

cat(1) utility

implementing, in Go 423, 424

channel of channels 480, 481, 482

channels

- about 60, 439, 449
- as function parameters 454, 455
- reading from 452, 453
- writing 450, 451
- circular list 237
- classification 727, 728, 729, 730
- closed channel
 - receiving from 453, 454
- closed variable 501
- closures 273
- clustering 731, 733
- cobra package
 - about 383
 - command aliases, creating 389, 390, 391
 - example 384, 385, 386, 387, 388
- code coverage
 - testing 554, 555, 556
- code optimization 527, 528
- command line arguments
 - working with 26, 27
- comparable 131
- complex number 108, 109, 110
- concurrency 441
- concurrent TCP server 688, 689, 690, 691, 692, 693, 695, 697, 698
- constant generator *iota*
 - about 136, 137, 138
 - defining 137
- consumers 748
- container package
 - about 237, 238, 244
 - container/heap, using 238, 239, 240
 - container/list, using 240, 241, 242
 - container/ring, using 243, 244
- container
 - using 238, 239, 240, 241, 242, 243, 244
- context package
 - about 508, 509, 510, 512
 - advanced example 512, 513, 516, 517
 - another example 517, 518, 519
 - worker pool 519, 520, 521, 522, 524
- continuation 465
- continuation stealing 466
- copy() function 123, 124, 125
- CPU profiling 529, 532

- critical section 487
- cross-compilation 583, 584
- crypto/rand package
 - reference link 252
- CSV files
 - about 403
 - reading 403, 404, 405, 406

D

- data amount
 - reading 400, 401, 402
- data race condition
 - about 503
 - catching 503, 506, 507
- data
 - loading, on disk 409, 410, 412
 - saving, on disk 409, 410, 412
- database backend
 - used, for testing HTTP server 556, 557, 558, 560, 563
- date
 - about 142, 143
 - formats, modifying 147, 148
 - parsing 146, 147
 - working with 146
- Dave Cheney 538
- deadlocks 469
- debug parse tree 95
- debugger 362
- debugging
 - example 363, 365, 366
 - with Delve 363
- defer keyword
 - about 77, 78, 79, 80
 - used, for logging 80, 81
- deferred functions 78
- degenerate 258
- Delve
 - debugging with 362
- dereferencing the pointer 139
- deserialization 409
- directed acyclic graph 211
- directed graph 211
- directives 11
- disassemble 588

- distance metric 727
- DNS lookups
 - performing 611, 612, 613
- Docker API 433, 434, 435
- Docker image
 - about 293
 - creating, for Go TCP/IP server 699, 700, 701
 - using 596, 597
- Docker
 - using 46, 48, 49
- documentation
 - generating 590, 591, 592, 594, 595
- Domain Name System (DNS) 611
- DOT language 542
- doubly linked list
 - advantages 230
 - implementing, in Go 227, 228, 229, 230
 - in Go 226, 227
 - versus singly linked list 227
- dtrace tool
 - about 86, 87, 88
 - URL 88
- dynamic programming 689

E

- Echo service 681
- element type 450
- Elixir 462
- encapsulation 286
- Erlang concurrency model
 - versus Go concurrency model 461, 462
- error data type 41, 42
- error handling
 - about 43, 44, 46
 - error data type 41, 42
 - in Go 41
- error messages 41
- error output 28, 29
- example functions
 - creating 585, 586
 - reference link 585
- execution time
 - measuring 149, 150

F

- fair scheduling strategy 465
- Fibonacci sequence 570
- fields of structure 154
- file descriptors 19
- file permissions 416, 417
- file
 - writing to 406, 408
- files
 - working with 359
- finite automaton 161
- flag package 370, 371, 373, 374
- floating-point numbers 108
- fmt.Println() function
 - reference link 305
- for loop 111, 112
- fork-join concurrency model 465
- format specifier 19

G

- G1 62
- garbage collection 10, 55, 56, 57, 58
- garbage collection safe-point 61
- garbage collector, Go code
 - reference link 61
- git cherry-pick command 362
- git commit command 357
- git diff command
 - using 360
- git pull command 357
- git push command 357
- git status command 356, 357
- git
 - .gitignore file 360
 - about 356
 - branches, working with 358, 359
 - files, working with 359
 - git cherry-pick command 362
 - git commit command 357
 - git diff command, using 360
 - git pull command 357
 - git push command 357
 - git status command 356, 357
 - tag, working with 360, 361, 362

- using 356
- GitHub 356
- GitLab 356
- global variables 134, 314
- Glut 403
- Gnuplot 403
- Go 2
 - number literals 111
 - reference link 111
- Go arrays
 - about 115
 - multi-dimensional arrays 116, 117, 118
 - shortcomings 118
- Go assembler 91, 92
- go build 98, 99
- Go channels 113
- Go channels revisited
 - about 475, 476
 - buffered channel 476, 477, 478
 - channel of channels 480, 481, 482
 - goroutines, avoiding 485, 486
 - nil channel 478, 479, 480
 - order of execution, specifying for goroutines 483, 484, 485
 - signal channel 476
- Go code
 - about 72, 73
 - advice 104
 - assembly, using 588, 589, 590
 - benchmarking 569, 570
 - code coverage, testing 554, 555, 556
 - compiling 12
 - convenient external package, for profiling 537, 538, 539
 - executing 13, 14
 - logical error, finding 581, 582, 583
 - net/http/pprof standard Go package 529
 - optimizing 528
 - profiling 529
 - profiling, example 529, 530, 531, 532, 533, 535, 536, 537
 - testing 549
 - tests, writing 549, 550, 551, 552, 554
 - to machine code 587, 588
 - web interface, of Go profiler 539
- Go compiler 54, 55
- Go concurrency model
 - versus Erlang concurrency model 461, 462
 - versus Rust concurrency model 461
- Go constants
 - about 134, 135, 136
 - constant generator iota 136, 137, 138
- go env command 90, 91
- Go environment 88, 89, 90
- Go execution tracer 544
- Go function
 - about 272, 273
 - anonymous functions 273
 - calling, from C code 75
 - multiple values, returning 273, 274, 275
 - other functions, accepted as parameters 281, 282
 - other functions, returning 279, 280
 - pointers, returning 278, 279
 - return values, naming 275, 276, 277
 - variadic functions 282, 283
 - with pointer parameters 277, 278
- Go garbage collector
 - about 62, 544
 - operation 61, 62
 - operation, measuring 151
- Go interface
 - about 104, 335, 336
 - switch statement, used with data types 341, 342, 343
 - type assertion 336, 337, 338
 - using 339, 340, 341
 - writing 338, 339
- Go loop
 - about 111
 - example 113, 114, 115
 - for loop 111, 112
 - range keyword 113
 - while loop 112, 113
- Go map
 - about 131, 132, 133, 571
 - need for 134
 - nil map, storing 133, 134
- Go module
 - about 17, 289

- code and information, storing 299
- creating 289
- different versions, using 298, 299
- go mod vendor command 300
- reference link 289
- using 289
- version v1.0.0, creating 290
- version v1.0.0, using 290, 291, 292
- version v1.1.0, creating 292
- version v1.1.0, using 293, 294
- version v2.0.0, creating 294, 295
- version v2.0.0, using 296
- version v2.1.0, creating 296
- version v2.1.0, using 297
- Go package
 - about 75, 76, 272
 - compiling 286
 - creating 300, 301, 302
 - developing 284, 285
 - downloading 17, 18
 - init() function 286, 287, 288
 - private functions 286
 - private variables 286
 - usage 14, 15, 16
- Go pointers
 - about 139, 141
 - need for 142
- Go profiler
 - web interface 539
- Go program
 - example 312, 313, 314
 - variable names, finding with string length 314, 315, 316, 318, 319
- Go race detector 503
- Go reflection
 - about 343, 344
 - advanced reflection example 346, 347, 348
 - disadvantages 349, 350
 - reflectwalk library 350, 351
 - simple reflection example 344, 346
- Go rules
 - about 14
 - curly braces, formatting 16
 - Go package, usage 14, 15, 16
- Go scheduler 441
- Go scheduler revisited 465, 466, 467
- Go scheduler, entities
 - goroutines (G) 466
 - logical processors (P) 466
 - OS threads (M) 466
- Go slices
 - about 56, 119
 - array, appending to 130, 131
 - byte slice 123
 - copy() function 123, 124, 125
 - example 126, 127, 128
 - expanding, automatically 121, 122, 123
 - multi-dimensional slice 125, 126
 - operations, performing 119, 120, 121
 - sorting, with sort.Slice() function 128, 129
- go statement
 - revisiting 500, 501, 502
- Go structures 92
- Go TCP/IP server
 - Docker image, creating for 699, 700, 701
- go tool trace utility 543, 544, 545, 547, 548
- Go type
 - about 137
 - method 333, 334, 335
- Go web client
 - creating 644, 645
 - upgrading 646, 647, 648, 649
- go/ast package
 - about 307
 - reference link 319
- go/parser package 307, 310, 311, 312
- go/scanner package 307, 308, 310
- go/token package 307
- Go
 - about 9, 433, 434, 435, 656
 - advantages 9
 - binary tree in 212
 - binary tree, implementing 213, 214, 215
 - C code, calling from 70
 - C code, calling with same file 70, 71
 - C code, calling with separate files 71
 - cat(1) utility, implementing 423, 424
 - cross-compilation capabilities, used for creating
 - WebAssembly code 100, 101
 - disadvantages 10

- doubly linked list in 226, 227
- doubly linked list, implementing 227, 228, 229, 230
- error handling in 41
- features 9, 10
- godoc utility 11, 12
- hash table in 216
- hash table, implementing 217, 218, 219
- history 8
- key-value store, developing in 187, 189, 190, 192
- linked list in 220, 221
- linked list, implementing 222, 223, 224, 225
- mixing, with C code 74
- object-oriented programming in 352, 353, 354, 355
- perfect programming language 10, 11
- queue in 231
- queue, implementing 231, 232, 233, 234
- stack in 234
- stack, implementing 234, 235, 236, 237
- UNIX pipe, programming in 422, 423
- web server, creating 616, 617, 618, 620
- website, creating 628, 630, 632, 636, 637
- glob format 409
- godoc package
 - reference link 590
- godoc utility 11, 12
- GOMAXPROCS environment variable 466, 467, 468
- Google Chrome
 - using 102
- goroutines
 - about 58, 412, 439, 440, 442
 - avoiding 485, 486
 - creating 442, 443, 444, 445
 - error messages 448, 449
 - finishing 445, 446, 448
 - timing out 471, 472, 473, 474, 475
 - used, for sharing memory 497, 498, 499
- grammar 160, 161
- graph structures 542
- graphical user interface (GUI) 10
- graphs
 - about 211
 - acyclic graphs 211

- cyclic graphs 211
- Graphviz 537, 542, 543
- gray set 59
- group ID 432, 433
- gRPC 656
- gRPC client
 - developing, in Go 658, 659, 660
- gRPC server
 - developing, in Go 660, 661, 662

H

- handler function 617
- hash function 216
- hash table
 - about 131
 - advantages 220
 - implementing, in Go 217, 218, 219
 - in Go 216
 - lookup functionality, implementing 219, 220
- Haskell programming language 564
- head 221
- heap
 - about 237
 - using 238, 239, 240
- Homebrew
 - URL 68
 - using 656
- HTML output
 - constructing 322, 323, 324, 325, 327, 328, 329
- HTML template 319
- HTTP connection
 - timeout, ways 653, 654, 655
 - timing out 649, 650, 651
- HTTP handlers
 - testing 641, 643
- HTTP server
 - profiling 623, 624, 625, 626
 - testing, with database backend 556, 557, 558, 560, 563
 - testing/quick package 564, 565, 566
- HTTP tracing 638, 639, 640
- http.Request type
 - about 603
 - reference link 603
- http.Response type 602

- `http.RoundTripper` 602
- `http.Transport` type
 - about 604
 - reference link 604
- HTTPS client
 - creating 668, 669
- HTTPS server 670, 671
- HTTPS traffic
 - certificates, creating 667, 668
 - TLS client, developing 671, 672, 674
 - TLS server, developing 671, 672, 674
 - working with 667

I

- interface 21, 238, 602
- interface definition file
 - defining 656, 657, 658
- interface definition language 656
- Internet Protocol (IP) 605
- `io.Reader` interface
 - about 391
 - buffered file input 392
 - buffered file output 392
 - unbuffered file input 392
 - unbuffered file output 392
- `io.Writer` interface
 - about 391
 - buffered file input 392
 - buffered file output 392
 - unbuffered file input 392
 - unbuffered file output 392
- IP address 167
- IPv4 605
- IPv4 address 167
- IPv6 605
- Iris dataset 730

J

- Java 353, 746
- Java code 185
- Java programming language 62
- JavaScript 100, 104
- JSON data
 - parsing 198, 199, 200, 201
 - reading 193, 194, 195

- saving 195, 196
- JSON format 409, 410
- JSON format, for Go
 - about 193
 - JSON data, parsing 198, 199, 200, 201
 - JSON data, reading 193, 194, 195
 - JSON data, saving 195, 196
 - `Marshal()` method, using 197, 198
 - `Marshal()`, using 198
 - `Unmarshal()` method, using 197
 - XML file, reading 204, 205
 - XML output, customizing 206, 207
 - XML, supporting from Go 201, 202, 203

K

- k-means clustering 731
- k-nearest neighbors (k-NN) 727
- Kafka 746, 747, 748, 751
- key-value store
 - about 187, 409
 - developing, in Go 187, 189, 190, 192
 - fundamental tasks, implementing 188

L

- last in first out (LIFO) 78
- lexical analysis 307
- lightweight tag 360
- linear regression
 - about 720
 - implementing 720, 721, 722
- linked list
 - advantages 225
 - implementing, in Go 222, 223, 224, 225
 - in Go 220, 221
- list
 - about 237
 - using 240, 241, 242
- little endian byte 400
- local variables 314
- log files
 - Go program, used for sending information 32, 33, 34
 - line numbers, printing in log entries 39, 40
 - log servers 31
 - `log.Fatal()` function 35

- log.Panic() function 36
- logging facility 30, 31
- logging level 30
- writing 30, 37, 39
- log package
 - reference link 38
- log servers 31
- log.Fatal() function 35
- log.Panic() function 36
- logging facility 30, 31
- logging level 30
- low-level network programming 708
 - performing 705, 706, 707
 - raw ICMP network data, grabbing 707, 709, 710

M

- machine code
 - from Go code 587, 588
- machine learning (ML) 716
- map key 131
- map literal 131
- map
 - about 62
 - splitting 65
 - using, with pointers 63
 - using, without pointers 64
- Marshal() method
 - using 197, 198
- matrices
 - adding 252, 253, 254
 - array dimensions, finding 264
 - dividing 258, 260, 261, 262
 - multiplying 255, 258
 - subtracting 252, 253, 254
- matrix 252
- matrix calculation
 - matrices, adding 252, 253, 254
 - matrices, dividing 258, 260, 261, 262
 - matrices, multiplying 255, 258
 - matrices, subtracting 252, 253, 254
 - performing 252
- memory profiling 529, 532
- memory
 - sharing, goroutines used 497, 498, 499
- message-passing 461

- method overriding 355
- modulo operator 217
- monitor goroutine 497
- multi-dimensional arrays 116, 117, 118
- multi-dimensional slice 125
- multi-dimensional slices 126
- mutator 59
- mutex variable 487
- mutual exclusion variable 487
- MX records
 - obtaining, of domain 615, 616

N

- name servers 613
- named type 137
- nc(1) command-line utility 606
- net 602
- net standard Go package 675
- net/http 602
- net/http/pprof standard Go package 529
- network interfaces
 - configuration, reading 606, 607, 608, 609, 611
- neural networks 736, 737
- Nginx 147, 616
- nil channel 478
- node trees 92, 95
 - about 96, 98
- Node.js 103
- nodes 211
- NS records
 - obtaining, of domain 613, 614, 615
- null 226
- number literals
 - about 111
 - in Go 2 111
 - reference link 111
- numeric data types
 - about 108
 - complex number 108, 109
 - complex numbers 110
 - floating-point numbers 108
 - integers 108
 - number literals, in Go 2 111

O

- O notation 211
- Oberon language 9
- object 333
- object-oriented programming
 - about 10
 - in Go 352, 353, 354, 355
- order of execution
 - specifying, for goroutines 483, 484, 485
- outlier analysis 738, 739, 741
- output
 - printing 19, 20, 21

P

- pack operator 282
- package variables 314
- panic 44
- panic function
 - about 82, 84
 - using 84, 85
- parallelism 441
- Pascal language 8
- pattern matching
 - about 160, 161
 - advanced example 164, 166, 167
 - example 161, 162, 163, 164
 - IPv4 addresses, matching 167, 168, 169, 170
- Perl 14, 542
- Pi
 - about 185
 - calculating, with high accuracy 185, 187
- pipeline 455, 456, 457, 458, 459
- pointers
 - about 139
 - map, using with 63
 - map, using without 64
- Pop() function 234
- PostgreSQL server 556
- pragmas 11
- preprocessor 10, 11
- presented techniques
 - performance, comparing 65, 66
- process 370, 440
- process ID 370

- profiling 529
- program 370, 440
- Protocol Buffers (protobuf) 656
- pseudo-random number
 - generating 244, 245, 246, 247
- pseudo-random strings
 - generating 248, 249
- Push() function 234
- Python 14, 353, 542

Q

- queue
 - implementing, in Go 231, 232, 233, 234
 - in Go 231
- quotient 110

R

- race conditions 55, 447, 459, 460
- range keyword 113
- raw ICMP network data
 - grabbing 707, 708, 709, 710
- re-slicing 120
- receive-only channel 455
- receiver method 333
- recognizer 161
- recover function 82, 83, 84
- reflection 343, 564
- regression
 - about 720
 - data, plotting 722, 723, 725, 726
- regular expressions
 - about 144, 160, 161
 - advanced example 164, 166, 167
 - example 161, 162, 163, 164
 - IPv4 addresses, matching 167, 168, 169, 170
 - theory 161
- Remote Procedure Call (RPC)
 - about 701
 - RPC client 702, 703
 - RPC server 703, 705
- RFC850 format 149
- ring
 - about 237
 - using 243, 244
- roots 59

- RPC client 702, 703
- RPC server 703, 705
- Ruby 14, 542
- rune 174, 176
- rune constant 174
- rune literal 174
- Rust concurrency model
 - versus Go concurrency model 461

S

- Scala 746
- Scalable Vector Graphics (SVG) 726
- scanner 307
- scheduler 465
- secure pseudo-random numbers
 - generating 250, 252
- Secure Sockets Layer (SSL) 668
- seed 245
- select keyword 468, 469, 470
- semantic analysis 307
- semantic versioning 289
- semaphores 477
- serialization 409
- server side
 - timeout period, setting 651, 653
- SetDeadline() function
 - using 651
- sharding 65
- shared memory 486, 487
- shared variables 486, 487
- shared-state 461
- short assignment statement 23, 24
- SIGINT signal 417
- signal channel 476
- signed integers 108
- singular 258
- slice literal 119
- slice of bytes 22
- slice of structures 321
- slice
 - about 62, 63
 - using 63
- sort.Slice() function
 - used, for sorting Go slices 128, 129
- sorted linked list 225
- sqlite3 package
 - reference link 323
- square matrix 252, 258
- stack
 - implementing, in Go 234, 235, 236, 237
 - in Go 234
- stalling join 465
- standard error 19
- standard input
 - about 19
 - reading from 24, 25
- standard output
 - about 19
 - using 21, 22
- static linking 10
- Static Single Assignment (SSA) 588
- statically linked executable file 13
- statistical learning 716
- statistical properties
 - calculating 716, 719
- stop-the-world garbage collector 61
- strace tool 85, 86
- stream encoding 409
- string literal 171
- string slice 179
- string
 - about 171, 172, 173
 - package 177, 178, 179, 180
 - rune 174, 176
 - unicode package 176
- strings 414
- strings package
 - revisiting 413, 414
- structure literal 154
- structures
 - about 154, 155
 - keyword, using 158, 159
 - pointers to 156, 157
- Sudoku puzzles
 - solving 264, 266, 267
- switch statement 181, 182, 184
- sync.Mutex type
 - about 487, 488, 489, 490
 - unlocking 490, 491
- sync.RWMutex type 492, 493, 495

- syntax error 16
- syscall package
 - about 302, 303, 304
 - fmt.Println() function, working 305, 306
- syscall.PtraceRegs 425, 427
- system call
 - about 302
 - tracing 427, 430, 431

T

- tag
 - about 292, 360
 - working with 360, 361, 362
- tail 221
- TCP client
 - about 675, 676, 677
 - version 677, 678, 679
- TCP server
 - about 679, 680, 681
 - version 681, 682, 683
- TCP/IP 605
- templates 319
- TensorFlow
 - about 741
 - working with 741, 742, 744
- testing/quick package 564, 565, 566
- text file
 - reading 392
 - reading, character by character 396, 397, 398
 - reading, from /dev/random system device 398, 399, 400
 - reading, line by line 393, 394
 - reading, word by word 394, 395, 396
- text output
 - generating 319, 320, 321, 322
- text template 319
- thread 440
- time
 - about 142, 143
 - formats, modifying 147, 148
 - parsing 144, 145, 146
 - working with 144
- timeout parameter
 - using 567, 568, 569
- timeout period

- setting, on sever side 651, 653
- TLS client
 - developing 671, 672, 674
- TLS server
 - developing 671, 672, 674
- Transmission Control Protocol (TCP) 605
- Transmission Control Protocol/Internet Protocol (TCP/IP) networks 192
- Transport Layer Security (TLS) 668
- tricolor algorithm 58, 59, 60, 61
- tricolor mark-and-sweep algorithm 58
- trim function 180
- tshark tools 655
- tuple assignment 160
- tuples 154, 159, 160
- type assertion 244, 336, 337, 338
- type identity 156

U

- UDP client 683, 684, 685, 686
- UDP server
 - developing 686, 688
- unbalanced tree 212
- underscore character (`_`) 160
- Unicode 10
- Unicode code point 174
- unicode package 176
- UNIX epoch time 143
- UNIX pipe
 - about 422
 - programming, in Go 422, 423
- UNIX processes
 - about 370
 - daemon processes 370
 - kernel processes 370
 - user processes 370
- UNIX signal
 - about 417
 - all signals, handling 420, 421, 422
 - handling 417, 418, 476
 - two signals, handling 418, 419, 420
- UNIX standard error 28
- UNIX stderr 18, 19
- UNIX stdin 18, 19
- UNIX stdout 18, 19

UNIX utilities

- about 85
- dtrace tool 86, 87, 88
- strace tool 85, 86

Unmarshal() method

- using 197, 198

unpack operator 282

unsafe code 66, 67, 68

unsafe package

- about 68
- example 69, 70

unsigned integers 108

unstructured JSON data 199

User Datagram Protocol (UDP) 605

user ID 432, 433

user input

- command line arguments, working with 25, 26, 27
- obtaining 23
- short assignment statement 23, 24
- standard input, reading from 24, 25

useUpdatedV2.go 297

V

variadic functions 282, 283

verbs

- about 19
- reference link 19

viper package

- about 375
- converting, from flag package 377, 378
- example 376, 377
- JSON configuration files, reading 379, 380
- YAML configuration files, reading 380, 381, 382, 383

W

web interface

Graphviz 542, 543

of Go profiler 539

profiling, example 539, 541

web server

creating, in Go 616, 617, 618, 620

WebAssembly code

- about 100
- creating, with cross-compilation capabilities of Go 100, 101
- example 101
- generated, using 102, 103, 104
- requisites 100

website

creating, in Go 628, 630, 632, 636, 637

while loop 112, 113

white set 58

Wireshark tools 655

work-stealing strategy 465

worker pool 519

write barrier

- about 59
- using 58

X

XML file

reading 204, 205

XML format 409

XML output

customizing 206, 207

XML

- about 201
- supporting, from Go 201, 202, 203

Y

YAML Ain't Markup Language (YAML) 207

YAML format, for Go

- about 207
- reference link 207