

СЕМЬ МОДЕЛЕЙ КОНКУРЕНЦИИ И ПАРАЛЛЕЛИЗМА ЗА СЕМЬ НЕДЕЛЬ

РАСКРЫВАЕМ ТАЙНЫ ПОТОКОВ

автор серии: **Пол Батчер**

редактор-консультант: **Жаклин Картер**

Seven Concurrency Models in Seven Weeks

When Threads Unravel

PAUL BUTCHER



Pragmatic
Bookshelf

DALLAS, TEXAS • RALEIGH, NORTH CAROLINA

Семь моделей конкуренции и параллелизма за семь недель

Раскрываем тайны потоков

ПОЛ БАТЧЕР



Москва, 2015

УДК 004.42
ББК 32.973
Б28

Б28 Пол Батчер

Семь моделей конкуренции и параллелизма за семь недель. /
Пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 360 с.: ил.

ISBN 978-5-97060-244-7

С появлением микропроцессоров, обладающих большим числом ядер, понимание конкуренции и параллелизма при разработке программного обеспечения стало еще более важным, чем прежде. В книге вы познакомитесь с преимуществами функционального программирования с точки зрения конкуренции, узнаете, как применять акторы для разработки распределенного программного обеспечения, и исследуете приемы параллельной обработки огромных объемов информации на нескольких процессорах. Эта книга поможет вам приобрести новые навыки в разработке программ, благодаря чему вы будете готовы решать сложные задачи в ближайшие несколько лет.

УДК 004.42
ББК 32.973

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-937785-65-9 (англ.)
ISBN 978-5-97060-244-7 (рус.)

© 2014 The Pragmatic Programmers, LLC.
© Оформление, перевод на русский язык
ДМК Пресс, 2015



Положительные отзывы к книге «Семь моделей конкуренции и параллелизма за семь недель»

Уже не одно десятилетие профессиональные программисты реализуют параллельное выполнение кода с применением потоков и блокировок. Но эта модель — лишь одна из многих, что ярко демонстрирует книга «Семь моделей конкуренции и параллелизма за семь недель». Если вы хотите добиться успеха в мире, где основные языки программирования постепенно обретают поддержку акторов, CSP¹, параллелизма данных, функционального программирования и заимствуют унифицированную модель последовательностей из языка Clojure, обязательно прочитайте эту книгу.

➤ **Стюарт Хэллоуэй (Stuart Halloway)**

Сооснователь, Cognitect

С появлением микропроцессоров, обладающих большим числом ядер, понимание конкуренции стало еще более важным, чем прежде. Вы познакомитесь с преимуществами функционального программирования с точки зрения конкуренции, узнаете, как применять акторы для разработки распределенного программного обеспечения, и исследуете приемы параллельной обработки огромных объемов информации на нескольких процессорах. Эта книга поможет вам приобрести новые навыки в разработке программ, благодаря чему вы будете готовы решать сложные задачи в ближайшие несколько лет.

➤ **Хосе Валим (José Valim)**

Сооснователь Plataformatec

В книге «Семь моделей конкуренции и параллелизма за семь недель», которая является поучительным обзором различных приемов параллельного/конкурентного выполнения, автору удалось найти баланс между описанием скучной теории и живыми экспериментами.

➤ **Фредерик Чанг (Frederick Cheung)**

Технический директор, Dressipi

Мир постоянно изменяется, и каждый действующий программист должен изучать приемы конкурентного программирования. Теперь, ког-

¹ Communicating sequential processes — взаимодействующие последовательные процессы. — *Прим. перев.*

да меня будут спрашивать: «Как сделать это?», – я смогу предложить прочитать эту книгу. Я многое узнал из нее и с радостью могу порекомендовать ее.

➤ **Эндрю Хейли (Andrew Haley)**

Ведущий программист на Java, Red Hat

По мере того, как закон Адмала² начинает преобладать над законом Мура³, происходит переход от объектно-ориентированного к конкурентно-ориентированному программированию. Вследствие этого данная книга появилась как нельзя кстати. Пол проделал фантастическую работу, описав наиболее важные модели конкуренции и дав читателю обширный набор инструментов, из которого он сможет выбирать наиболее подходящие для его потребностей. Эту книгу должен прочитать каждый, кто разрабатывает программное обеспечение в нашу многоядерную эпоху.

➤ **Франческо Цезарини (Francesco Cesarini)**

Основатель и технический директор Erlang Solutions

В своей книге Пол дает превосходное введение в тернистую тему конкуренции и параллелизма, ясно и доходчиво описывая различные подходы.

➤ **Шон Эллис (Sean Ellis)**

Архитектор GPU, ARM

Просто о сложном. Я бы с удовольствием прослушал университетский курс по этой теме, взяв в качестве руководства «Семь моделей конкуренции и параллелизма за семь недель».

➤ **Карлос Сесса (Carlos Sessa)**

Разработчик для Android, Groupon

Пол Батчер (Paul Butcher) взял проблему, которая бросает в дрожь многих разработчиков, и наглядно продемонстрировал целую серию парадигм программирования, которые с успехом можно использовать для реализации конкуренции в программном обеспечении.

➤ **Пайди Крид (Páidí Creed)**

Программист, SwiftKey

Имея опыт сотрудничества с Полом, могу рекомендовать его как опытного эксперта в области архитектуры языков программирования. Эта книга является ясным описанием темы, часто недооцененной, но жизненно важной для разработки программного обеспечения.

➤ **Бен Медлок (Ben Medlock)**

Сооснователь и технический директор, SwiftKey

² https://ru.wikipedia.org/wiki/Закон_Адмала – Прим. перев.

³ https://ru.wikipedia.org/wiki/Закон_Мура – Прим. перев.



ОГЛАВЛЕНИЕ

Положительные отзывы к книге «Семь моделей конкуренции и параллелизма за семь недель»	5
Предисловие	13
Благодарности.....	15
Вступление	17
О книге	17
Чем не является эта книга.....	18
Примеры кода	18
Примечание для пользователей IDE.....	19
Примечание для пользователей Windows.....	19
Ресурсы в Сети.....	19
Глава 1. Введение.....	21
Конкуренция или параллелизм?	21
Похожие, но разные.....	21
За рамками последовательного программирования.....	23
Параллельная архитектура	23
Параллелизм на уровне битов	24
Параллелизм на уровне инструкций	24
Параллелизм данных.....	24
Параллелизм на уровне задач	25
Конкуренция: за рамками множества ядер	26
Конкурентные программы для конкурентного мира	26
Распределенные программы для распределенного мира	27
Надежные программы для непредсказуемого мира.....	27
Простые программы в сложном мире	28
Семь моделей	28
Глава 2. Потоки выполнения и блокировки	31
Самое простое из того, что может работать.....	31
День 1: взаимоисключение и модели памяти.....	32
Создание потока	33
Наша первая блокировка	34
Загадочная память	37

Видимость памяти	38
Несколько блокировок	39
Опасности сторонних методов	43
В завершение первого дня	44
День 2: помимо встроенных блокировок	46
Прерываемое блокирование	47
Тайм-ауты	49
Блокирование методом перебора	51
Условные переменные	54
Атомарные переменные	57
В завершение второго дня	58
День 3: на плечах гигантов	60
Еще раз о создании потоков	61
Копирование при записи	62
Законченная программа	64
В завершение третьего дня	74
В завершение	75
Сильные стороны	75
Слабые стороны	76
Другие языки	78
Напоследок	79

Глава 3. Функциональное программирование 80

Если какие-то действия вредят вам, перестаньте выполнять их	80
День 1: программирование без изменяемого состояния	81
Опасности изменяемого состояния	81
Краткий экскурс в язык Clojure	84
Первая функциональная программа	86
Параллелизм без усилий	87
Функциональный подсчет слов	89
Лень – это благо	93
В завершение первого дня	94
День 2: функциональный параллелизм	95
По одной странице за раз	95
Разделение данных на пакеты для увеличения производительности	98
Редуценты (reducers)	99
Внутреннее устройство редуцентов	100
Разделяй и властвуй	103
Поддержка функции fold	104
Подсчет слов с помощью fold	105
В завершение второго дня	107
День 3: функциональная конкуренция	108
Та же структура, разный порядок вычислений	108

Ссылочная прозрачность	109
Потоки данных	110
Механизм future	111
Механизм promise	112
Функциональная веб-служба	113
В завершение третьего дня	121
В завершение	122
Сильные стороны	124
Слабые стороны	124
Другие языки	124
Напоследок	125

Глава 4. Путь Clojure – разделение идентичности и состояния 126

Лучшее из двух миров	126
День 1: атомы и сохраненные структуры данных	127
Атомы	127
Многопоточная веб-служба с изменяемым состоянием	129
Сохраненные структуры данных	130
Идентичность или состояние	134
Повторения	134
Валидаторы	135
Функции-наблюдатели	135
Гибридная веб-служба	136
В завершение первого дня	140
День 2: агенты и программная транзакционная память	141
Агенты	141
Журнал в памяти	145
Программная транзакционная память	146
Изменяемое разделяемое состояние	151
В завершение второго дня	151
День 3: погружение в глубину	152
Решение задачи о философах на основе STM	153
Решение задачи о философах без применения STM	155
Атомы или STM?	157
Собственная реализация конкуренции	158
В завершение третьего дня	160
В завершение	161
Сильные стороны	161
Слабые стороны	162
Другие языки	162
Напоследок	162

Глава 5. Акторы 164

Не объекты, а скорее ориентированные на объекты	164
---	-----

День 1: сообщения и почтовые ящики.....	166
Наш первый актер	166
Почтовые ящики и очереди.....	167
Прием сообщений	168
Связывание процессов.....	169
Актеры с сохранением состояния	170
Соккрытие сообщений за фасадом API.....	171
Двунаправленное взаимодействие.....	172
Именованное взаимодействие	174
Отступление – функции первого порядка.....	176
Параллельная версия map().....	176
В завершение первого дня	177
День 2: обработка ошибок и отказоустойчивость.....	178
Актер кэширования	179
Определение момента отказа.....	182
Слежение за работой процессов	185
Тайм-ауты.....	186
Ядро ошибки	187
И пусть падает!	189
В завершение второго дня.....	190
День 3: распределенные приложения.....	191
ОТР	191
Узлы	196
Распределенный счетчик слов.....	200
В завершение третьего дня	206
В завершение.....	207
Сильные стороны	208
Слабые стороны	209
Другие языки	209
Напоследок	210

Глава 6. Взаимодействие последовательных процессов 211

Взаимодействия – это все.....	211
День 1: каналы и блоки go.....	213
Каналы	213
Блоки go	217
Операции с каналами	222
В завершение первого дня	225
День 2: множество каналов и ввод/вывод.....	227
Обслуживание множества каналов	227
Асинхронный опрос	230
Асинхронный ввод/вывод	233
В завершение второго дня.....	240
День 3: модель CSP на стороне клиента	241

Конкуренция – это образ жизни	242
Привет, ClojureScript	242
Обработка событий	245
Усмирение функций обратного вызова	247
Отправляемся в путь, чтобы увидеть Мастера	248
В завершение третьего дня	251
В завершение	251
Сильные стороны	252
Слабые стороны	252
Другие языки	253
Напоследок	253

Глава 7. Параллелизм данных..... 254

В недрах вашего ноутбука спрятан суперкомпьютер.....	254
День 1: программирование GPGPU.....	255
Обработка данных и параллелизм данных	255
Наша первая программа для OpenCL.....	258
Профилирование.....	264
Множество возвращаемых значений	265
Обработка ошибок.....	266
В завершение первого дня	268
День 2: многомерность и рабочие группы.....	270
Многомерные массивы рабочих элементов	270
Получение информации об устройстве	273
Модель платформы	275
Модель памяти	276
Параллельная свертка	277
Свертка с одной рабочей группой.....	277
В завершение второго дня.....	282
День 3: OpenCL и OpenGL – храните данные в GPU	283
Водная рябь	284
LWJGL.....	284
Отображение сетки в OpenGL	285
Доступ к буферу OpenGL из ядра OpenCL	287
Имитация ряби	288
В завершение третьего дня	291
В завершение	293
Сильные стороны	293
Слабые стороны	294
Другие языки	294
Напоследок	294

Глава 8. Лямбда-архитектура 295

Параллелизм позволяет обрабатывать гигантские объемы данных.....	295
---	-----

День 1: MapReduce	297
Практические аспекты	298
Основы Hadoop	299
Подсчет слов с помощью Hadoop	301
Опробование на Amazon EMR	305
Обработка XML	308
В завершение первого дня	310
День 2: пакетный уровень	313
Проблемы с традиционными системами данных	313
Вечные истины	315
Лучшие данные – исходные данные	315
Авторы правок в Википедии	317
Завершение картины	323
В завершение второго дня	325
День 3: уровень ускорения	327
Архитектура уровня ускорения	328
Подсчет правок с помощью Storm	335
В завершение третьего дня	341
В завершение	342
Сильные стороны	343
Слабые стороны	343
Альтернативы	343
Напоследок	343
Глава 9. В заключение	344
Куда мы идем?	344
Будущее за неизменяемостью	345
Будущее за распределенными вычислениями	346
Темы, оставшиеся за бортом	346
Fork/Join и захват задачи	347
Потоки данных	347
Реактивное программирование	347
Функциональное реактивное программирование	348
Grid-вычисления	348
Пространства кортежей	348
Выбор за вами	349
Библиография	350
Предметный указатель	352



ПРЕДИСЛОВИЕ

Эта книга рассказывает историю.

Кому-то может показаться странным, что книга начинается с этих слов, но для меня они наполнены смыслом. Видите ли, при создании книги «Семь моделей конкуренции и параллелизма за семь недель» мы отклонили десятки предложений от разных авторов, считавших, что можно собрать в кучу семь не связанных между собой статей и назвать это книгой. Мы думаем иначе.

В первой нашей истории («Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages» [Tat10]) рассказывалось, что объектно-ориентированные языки были хороши для своего времени, но под давлением всевозрастающей сложности программного обеспечения и необходимости управления конкуренцией в многоядерных архитектурах стали появляться языки функционального программирования, формирующие иные подходы к программированию. Пол Батчер (Paul Butcher) оказался самым полезным из научных редакторов этой книги. После четырехлетнего знакомства я понял, почему.

Пол многое повидал, находясь в первых рядах тех, кто внедрял по-настоящему масштабируемую поддержку конкуренции в действующие приложения. В книге «Seven Languages» он подметил, что решение многих сложных задач все чаще и чаще переносится на уровень языка. Пару лет спустя Пол пришел к нам с предложением написать свою книгу. Он говорил, что языки занимают важное место в общей истории, но они – лишь верхушка айсберга. Он хотел поведать нашим читателям намного более полную историю и простым языком рассказать о наиболее важных инструментах, которые используются в современном программном обеспечении для решения проблем, связанных с параллельным выполнением кода.

Сначала мы скептически отнеслись к его предложению. Такие книги пишутся тяжело – для них требуется больше времени, а процент неудач слишком высок – Пол решил сразиться со слишком большим драконом. Но, став настоящей командой, мы смогли превратить пер-

воначальный список глав в хорошую историю. По мере того как росла стопка страниц, мы все отчетливее понимали, что Пол не только имел обширные технические знания для раскрытия данной темы, но и азарт, чтобы взяться за нее. Мы также осознали всю особенность этой книги и ее своевременность. По мере углубления в нее вы поймете, о чем я говорю.

Вы будете ежиться вместе с нами, читая главу о потоках выполнения и блокировках – наиболее широко используемых ныне механизмах реализации конкуренции. Вы увидите, в каких ситуациях это решение дает плохие результаты, и затем приметесь за работу. Пол познакомит вас с самыми разными подходами, от лямбда-архитектуры, используемой в некоторых высоконагруженных социальных платформах, до модели акторов, лежащей в основе многих крупнейших в мире, высоконадежных телекоммуникационных систем. Вы увидите языки программирования, которыми пользуются профессионалы, от Java до Clojure и захватывающего языка Elixir, основанного на Erlang. И на каждом шаге этого сложного пути Пол будет поддерживать вас, делаясь своими глубокими знаниями посвященного.

Я рад представить вам книгу «Семь моделей конкуренции и параллелизма за семь недель» и надеюсь, что для вас она окажется такой же ценной, как и для меня.

Брюс Тейт (Bruce A. Tate)

Технический директор icanmakeitbetter.com

Редактор серии «Семь за семь»

Остин, Техас



БЛАГОДАРНОСТИ

Когда я объявил, что подписал контракт на создание этой книги, друг спросил меня: «Как, прошло столько времени, что ты забыл, как тебе далась первая книга?» Я был слишком наивен, полагая, что работать над второй книгой будет проще. Возможно, если бы я выбрал иной формат, отличный от формата серии «Семь за семь», я был бы не так уж и не прав.

Я едва ли смог бы написать эту книгу без мощной поддержки редактора серии Брюса Тейта и редактора-консультанта Джеки Картер (Jackie Carter). Спасибо вам обоим, что поддерживали меня, пока я вынашивал эту книгу, а также Дейву и Энди за возможность еще раз внести свой вклад в эту великолепную серию.

Многие делились со мной советами во время работы над рукописью, в том числе (без какого-то определенного порядка): Симон Харди-Френсис (Simon Hardy-Francis), Сэм Холлидей (Sam Halliday), Майк Смит (Mike Smith), Нил Эклс (Neil Eccles), Мэтью Руди Джекобс (Matthew Rudy Jacobs), Джо Осборн (Joe Osborne), Дейв Страусс (Dave Strauss), Дерек Ло (Derek Law), Фредерик Чанг (Frederick Cheung), Хьюго Тайсон (Hugo Tyson), Пол Грегори (Paul Gregory), Стефен Спенсер (Stephen Spencer), Алекс Никсон (Alex Nixon), Бен Коппин (Ben Coppin), Кит Смитерс (Kit Smithers), Эндрю Икотт (Andrew Eacott), Фриланд Эбботт (Freeland Abbott), Джеймс Алей (James Aley), Мэтью Уилсон (Matthew Wilson), Симон Добсон (Simon Dobson), Дуг Опп (Doug Orr), Джонас Бонер (Jonas Bonér), Стю Хэллоуэй (Stu Halloway), Рич Морин (Rich Morin), Дэвид Уайттакер (David Whittaker), Бо Ридберг (Bo Rydberg), Джейк Гулдинг (Jake Goulding), Ари Голд (Ari Gold), Хуан Мануэль Химено Илла (Juan Manuel Gimeno Illa), Стив Бассетт (Steve Bassett), Норберто Ортигоза (Norberto Ortigoza), Лучано Рамальо (Luciano Ramalho), Шива Джайараман (Siva Jayaraman), Шон Перри (Shaun Parry) и Джоэл Ван дер Верф (Joel Van der Werf).

Особое спасибо научным рецензентам книги (и снова без какого-то особого порядка): Карлос Сесса (Carlos Sessa), Денни Вудс (Danny

Woods), Венкат Субраманиам (Venkat Subramaniam), Симон Вуд (Simon Wood), Пайди Крид (Páidí Creed), Ян Ругли (Ian Roughley), Эндрю Томсон (Andrew Thomson), Эндрю Хейли (Andrew Haley), Шон Эллис (Sean Ellis), Джефффри Клементс (Geoffrey Clements), Лорен Сандс-Рамшау (Loren Sands-Ramshaw) и Пол Хадсон (Paul Hudson).

Наконец, я хочу принести слова благодарности и свои извинения моим друзьям, коллегам и родным. Спасибо за вашу поддержку и извините, что стремился к уединению последние восемь месяцев.



ВСТУПЛЕНИЕ

В 1989-м я приступил к работе над диссертацией, в которой занялся исследованием реализации параллельных и распределенных вычислений на разных языках программирования, – я был убежден, что со временем конкурентное программирование займет господствующее положение. Спустя два десятилетия моя правота была доказана – вокруг ведется масса разговоров о многоядерных архитектурах и о том, как использовать их преимущества.

Но поддержка конкуренции позволяет получить не только высокую производительность многоядерных систем. При правильном использовании конкуренция оказывается ключом к высокой отзывчивости, надежности, эффективности и простоте программного обеспечения.

О книге

Эта книга имеет ту же структуру, что и остальные книги серии «Семь за семь» издательства The Pragmatic Bookshelf: «Seven Languages in Seven Weeks» [Tat10], «Seven Databases in Seven Weeks» [RW12] и «Seven Web Frameworks in Seven Weeks» [MD14].¹ Для этой книги были отобраны семь подходов, чтобы на их примере продемонстрировать общее положение дел в области поддержки конкуренции. Мы охватим подходы, которые уже прочно заняли господствующее положение, которые вскоре могут занять такое положение, а также те, которые вряд ли получат достаточно широкое распространение, но являются фантастически мощными в своих нишах. Я надеюсь, что по прочтении этой книги вы будете точно знать, какой инструмент выбрать, столкнувшись с той или иной проблемой конкуренции.

Каждая глава спроектирована так, чтобы ее можно было прочитать за выходные, и разбита на три дня. Каждый день завершается упражнениями, выполнение которых поможет закрепить знания, получен-

¹ В издательстве «ДМК Пресс» выпущено две книги этой же серии:
<http://dmkpress.com/catalog/computer/databases/978-5-94074-866-3/>
<http://dmkpress.com/catalog/computer/programming/978-5-94074-539-6/>

ные в этот день, а каждая глава завершается кратким перечислением сильных и слабых сторон подхода, изучавшегося в этой главе.

Несмотря на небольшие философские отступления, основное внимание в книге будет уделяться практическим примерам. Я настоятельно рекомендую поработать с этими примерами – нет ничего более убедительного, чем действующий код.

Чем не является эта книга

Эта книга не является справочным руководством. Я буду использовать языки, с которыми вы, возможно, плохо знакомы, такие как Elixir и Clojure. Но, поскольку главной темой этой книги является конкуренция, а не языки, я не буду подробно описывать те или иные аспекты языков, используемые в примерах. Надеюсь, что все и так будет понятно из контекста, но также надеюсь, что вы самостоятельно сможете заняться исследованием того или иного языка, если это потребуется. Возможно, удобнее будет читать эту книгу с веб-браузером под рукой, чтобы можно было оперативно обратиться к описанию языка в Интернете.

Эта книга не содержит инструкций по установке. Чтобы опробовать примеры кода, вам придется установить и настроить различные инструменты – какие-то подсказки вы найдете в файлах README, сопровождающих примеры, но в общем и целом вам придется проявить свою самостоятельность. Я использовал широко распространенные инструменты для всех примеров, поэтому вы не будете испытывать недостатка в руководствах по установке, которые легко можно найти в Интернете.

Наконец, эта книга не является исчерпывающим источником информации – в книге не так много места, чтобы охватить каждую тему во всех подробностях. Какие-то аспекты я буду упоминать лишь мимоходом, а какие-то просто пропущу. По возможности я преднамеренно использовал неидиоматический код, чтобы тем, кто незнаком с используемыми языками программирования, было проще понять, как он действует. Если вы решите заняться более глубокими исследованиями какой-либо из представленных здесь технологий, обращайтесь к специализированным книгам, упоминаемым в тексте.

Примеры кода

Все примеры кода, демонстрируемые в книге, можно загрузить с сайта книги². Каждый пример включает не только исходный код, но и

² <http://pragprog.com/book/pb7con/>

систему сборки. Для каждого языка я выбрал наиболее популярную систему сборки (Maven – для Java, Leiningen – для Clojure, Mix – для Elixir, sbt – для Scala и GNU Make – для C).

В большинстве случаев эти системы не только создадут из примера выполняемую программу, но и автоматически загрузят дополнительные зависимости. А системы sbt и Leiningen загрузят даже соответствующие версии компиляторов Scala и Clojure, поэтому вам остается только установить соответствующий инструмент сборки, а описание процесса установки вы без труда найдете в Интернете.

Главным исключением является код на языке C, демонстрирующийся в главе 7 «Параллелизм данных», для опробования которого вам придется установить инструмент OpenCL для вашей операционной системы и графической карты (если только вы не пользуетесь Mac OS, среда разработки Xcode в которой уже включает все необходимое).

Примечание для пользователей IDE

Все системы сборки были проверены из командной строки. Если вы пользуетесь исключительно интегрированной средой разработки (Integrated Development Environment, IDE), вам придется импортировать систему сборки в свою IDE – большинство IDE уже поддерживают систему сборки Maven и имеют расширения для поддержки sbt и Leiningen. Но я не проверял этого, поэтому многим из вас проще будет научиться пользоваться командной строкой.

Примечание для пользователей Windows

Все примеры были протестированы в OS X и Linux. Они также должны работать и в Windows, но я не проверял их в этой операционной системе.

Исключением является код на языке C, демонстрирующийся в главе 7 «Параллелизм данных», для компиляции которого используется GNU Make и GCC. Должно быть, несложно перенести этот код на Visual C++, но, опять же, я не проверял этого.

Ресурсы в Сети

Приложения и примеры, демонстрирующиеся в этой книге, можно найти на веб-сайте Pragmatic Programmers.³ Там же вы найдете форум читателей и форму для отправки сообщений, с помощью которой вы

³ <http://pragprog.com/book/pb7con>

сможете сообщить нам о найденных ошибках и опечатках или внести предложения к будущим изданиям книги.

Пол Батчер (Paul Butcher)

Ten Tenths Consulting

paul@tententhsconsulting.com

Кембридж, Великобритания, июнь 2014



ГЛАВА 1.

Введение

Конкурентное программирование не является чем-то новым, но оно лишь недавно превратилось в горячую тему. Языки программирования, такие как Erlang, Haskell, Go, Scala и Clojure, получили известность отчасти благодаря превосходной поддержке конкуренции.

Основной причиной всплеска интереса стал так называемый «кризис многоядерных систем». В соответствии с законом Мура число транзисторов на одном кристалле постоянно увеличивается¹, но это приводит не к повышению быстродействия процессора, а к увеличению числа ядер.

Как заметил Герб Саттер (Herb Sutter): «Бесплатного супа больше не будет».² Вы больше не сможете получить более высокую скорость выполнения вашего кода, просто подождав, пока появится более быстрое «железо». В наши дни, чтобы добиться более высокой производительности, требуется использовать несколько ядер, а значит, и приемы параллельного программирования.

Конкуренция или параллелизм?

Эта книга рассказывает о конкуренции, тогда почему мы говорим о параллелизме? Хотя эти два термина часто используются взаимозаменяемо, конкуренция и параллелизм суть разные вещи.

Похожие, но разные

Конкурентная программа всегда имеет несколько потоков выполнения. Потоки могут выполняться параллельно, но это не обязательно.

¹ https://ru.wikipedia.org/wiki/Закон_Мура

² <http://www.gotw.ca/publications/concurrency-ddj.htm> (Перевод на русский язык: <http://habrahabr.ru/post/145432/> – Прим. перев.)

Параллельная программа обычно действует намного быстрее, чем последовательная, решая сразу (параллельно) несколько задач. При этом она необязательно имеет несколько потоков выполнения.

Конкуренцию можно считать аспектом предметной области, например когда программе требуется обрабатывать одновременно (или почти одновременно) возникающие события. Параллелизм, напротив, является аспектом решения, например с целью увеличить скорость работы за счет параллельной обработки разных блоков данных.

Вот как обозначает различия Роб Пайк (Rob Pike),³

- Конкуренция – это способ одновременного решения множества разных задач.
- Параллелизм – это способ одновременного выполнения разных частей одной задачи.

Так о чем эта книга, о конкуренции или о параллелизме?



Джо спрашивает: в чем разница между конкуренцией и параллелизмом?

Моя жена – учитель. Как и большинство учителей, она может заниматься сразу несколькими делами. В каждый конкретный момент она может делать только что-то одно, но способна делать несколько дел конкурентно. Слушая, как читает кто-то из ее учеников, она может прервать его, чтобы успокоить расшумевшийся класс или ответить на чей-то вопрос. Это – конкурентная работа, а не параллельная (она может делать только что-то одно).

Если ей в помощь дать помощника (один из них будет слушать ученика, читающего текст, а другой – отвечать на вопросы), мы получим вариант работы в паре, который можно назвать конкурентным и параллельным.

Представьте, что класс придумал поздравительные открытки и решает заняться их массовым выпуском. Решить эту проблему можно, поручив каждому ребенку нарисовать пять открыток. Это параллельная организация работы (если отбросить несущественные мелочи), но не конкурентная – выполняется только одна задача.

³ <http://concur.rspace.googlecode.com/hg/talk/concur.html>

За рамками последовательного программирования

Параллелизм и конкуренцию объединяет то, что обе эти парадигмы выходят за рамки модели последовательного программирования, в которой все процессы протекают последовательно, друг за другом. В этой книге мы будем рассматривать обе парадигмы, параллельного и конкурентного программирования (если бы я был педантом, я дал бы этой книге название «Семь моделей конкурентного и/или параллельного программирования за семь недель», но такое название едва ли уместилось бы по ширине обложки).

Конкуренцию и параллелизм часто путают между собой, потому что традиционные потоки выполнения и блокировки не имеют прямого отношения к параллелизму. Если вы пожелаете задействовать несколько ядер с помощью потоков и блокировок, вы вынуждены будете создать конкурентную программу и запускать ее на параллельной архитектуре.

Такое ограничение весьма досадно, потому что программы, реализующие конкурентную модель, часто недетерминированы — они дают разные результаты в зависимости от точности синхронизации событий. При решении по-настоящему конкурентной задачи недетерминированность является естественным и вполне ожидаемым следствием. Параллелизм, напротив, необязательно подразумевает недетерминированность — операция удвоения всех чисел в массиве не приведет (по крайней мере, не должна) к недетерминированности только потому, что удвоение половины чисел выполняется одним ядром, а другая половина — другим. Языки с явной поддержкой параллелизма позволяют писать параллельный код, не внося недетерминированности.

Параллельная архитектура

Большинство из нас привыкло связывать параллелизм с наличием множества ядер, однако в действительности современные компьютеры обеспечивают параллелизм на многих других уровнях. Причина, до недавнего времени объяснявшая увеличение скорости работы ядер с каждым годом, заключается в параллельном использовании всех тех дополнительных транзисторов, предсказанных законом Мура, — как на уровне отдельных битов, так и на уровне инструкций.

Параллелизм на уровне битов

Почему 32-разрядный компьютер быстрее 8-разрядного? Дело в параллелизме. Чтобы сложить два 32-разрядных числа, на 8-разрядном компьютере потребуется выполнить последовательность 8-разрядных операций. На 32-разрядном компьютере, напротив, сложение можно выполнить в одно действие, параллельно обрабатывая все 4 байта, составляющие каждое 32-разрядное число.

Именно поэтому развитие компьютерной техники, которое мы наблюдали, двигалось от 8- к 16-, 32- и теперь уже 64-разрядным архитектурам. Однако подобный параллелизм имеет свои ограничения, из-за которых мы едва ли увидим в ближайшем будущем 128-разрядные компьютеры.

Параллелизм на уровне инструкций

Современные микропроцессоры имеют высокую степень параллелизма за счет использования таких технологий, как *конвейерная обработка, неупорядоченное и спекулятивное выполнение команд*.

Как программисты мы часто могли просто игнорировать эти особенности, потому что процессоры тщательно маскировали свой параллелизм, создавая иллюзию последовательного выполнения операций.

Однако теперь эта иллюзия постепенно разрушается. Создатели процессоров не в состоянии больше увеличивать скорость работы ядер. А кроме того, так как мы входим в многоядерный мир, мы должны начинать учитывать тот факт, что инструкции выполняются не последовательно. Но подробнее об этом мы поговорим в разделе «Видимость памяти» главы 2.

Параллелизм данных

Архитектуры с параллельной обработкой данных (иногда их называют SIMD, «single instruction, multiple data» – «с одним потоком команд и многими потоками данных») способны параллельно выполнять одни и те же операции с большим количеством данных. Нельзя сказать, что такие архитектуры подходят для решения всех видов задач, но они могут быть весьма эффективны при правильном применении.

Одним из таких правильных применений является обработка изображений. Например, чтобы увеличить яркость изображения, требуется увеличить яркость каждого пикселя. По этой причине развитие

современных GPU (graphics processing units – графические микропроцессоры) пошло по пути параллельной обработки данных.

Параллелизм на уровне задач

Наконец мы достигли того, что многие из нас привыкли считать параллелизмом, – наличия множества процессоров. С точки зрения программиста, наиболее важной отличительной чертой многопроцессорных архитектур является модель представления памяти, а именно – является память распределенной или разделяемой.

В модели с *разделяемой памятью* все процессоры могут обращаться к любым участкам памяти, и все взаимодействия между процессорами осуществляются преимущественно через память, как показано на рис. 1.

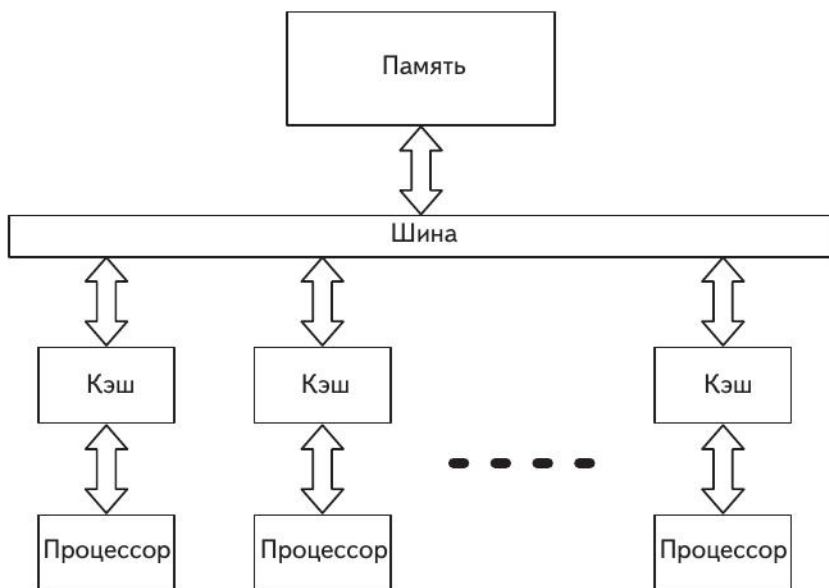


Рис. 1. Разделяемая память

На рис. 2 изображена модель системы с распределенной памятью, в которой каждый процессор имеет собственную локальную память, а взаимодействия между ними осуществляются через сеть.

Так как взаимодействия через память обычно осуществляются быстрее и проще, чем через сеть, разработка программ для архитектур с разделяемой памятью в общем случае дается гораздо легче. Но с уве-

личением числа процессоров разделяемая память становится узким местом — чтобы максимально использовать преимущества, которые дает наличие большого числа процессоров, необходим переход к использованию *распределенной памяти*. Использование распределенной памяти также является обязательным условием для тех, кто создает высоконадежные системы на основе множества компьютеров с целью избежать отказов в работе из-за выхода из строя аппаратных средств.

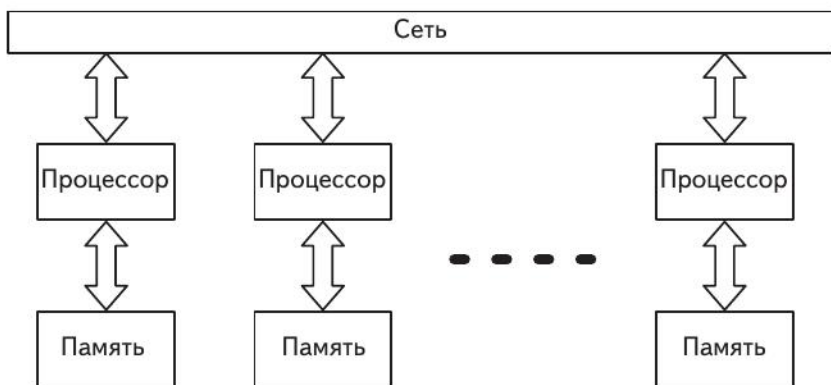


Рис. 2. Распределенная память

Конкуренция: за рамками множества ядер

Конкуренция — это нечто большее, чем простое использование параллелизма: при правильном подходе применение приемов конкурентного программирования позволяет писать более отзывчивые, надежные, эффективные и простые программы.

Конкурентные программы для конкурентного мира

Мир по своей природе является конкурентным, таковыми должны быть и программы, если мы хотим, чтобы они действовали эффективно.

Ваш мобильный телефон может проигрывать музыку, передавать данные по сети, следить за движениями ваших пальцев по экрану, и все это одновременно. Ваша IDE проверяет синтаксис исходного кода

в фоновом режиме, пока вы продолжаете ввод. Бортовой компьютер самолета одновременно опрашивает многочисленные датчики, отображает информацию для пилотов, повинуется командам и управляет рулями.

Конкуренция является ключом к созданию отзывчивых систем. Загружая файлы в фоновом режиме, вы не вынуждаете пользователей в расстроенных чувствах наблюдать песочные часы на экране. Обработывая сразу множество подключений, веб-сервер гарантирует, что один медленный запрос не затормозит обслуживание других.

Распределенные программы для распределенного мира

Иногда географическая разбросанность оказывается ключевой особенностью решаемой задачи. Любое программное обеспечение, распределенное среди множества компьютеров, которые не действуют в жесткой связке, по своей природе является конкурентным.

Кроме того, распределенное программное обеспечение помогает повысить надежность. Вы можете разместить одну половину своих серверов в Европе, другую половину – в США, и тогда отключение электроэнергии в одном месте не приведет к глобальной остановке. Это обстоятельство подводит нас к теме способности противостоять внешним факторам.

Надежные программы для непредсказуемого мира

Программы содержат ошибки, и иногда программы завершаются аварийно. Но даже если бы мы могли создавать программы, полностью лишенные ошибок, аппаратное обеспечение тоже не вечно и иногда выходит из строя.

Использование приемов конкуренции повышает надежность, или отказоустойчивость программ за счет *независимости* и механизма *обнаружения неисправностей*. Независимость играет важную роль, потому что ошибки в одной задаче не должны приводить к отказу от выполнения других задач. А механизм обнаружения ошибок помогает в случае ошибки в одной задаче (из-за аварийного завершения, «зависания» или выхода из строя аппаратных средств) известить другую задачу о необходимости проведения восстановительных действий.

Последовательные программы никогда не смогут быть столь же надежными, как конкурентные.

Простые программы в сложном мире

Если вам доводилось часами отыскивать сложные в диагностике ошибки, связанные с особенностями выполнения в многопоточной среде, вам трудно будет поверить, что конкурентные решения могут быть проще своих последовательных эквивалентов, если они написаны на правильно выбранном языке и с применением правильных инструментов.

Это особенно верно для программ, решающих проблемы, конкурентные по своей природе. Лишний труд, который необходим для преобразования конкурентной проблемы в последовательное решение, омрачает работу. Вы можете избежать этого лишнего труда, создав решение со структурой, соответствующей решаемой проблеме: вместо того чтобы пытаться воспроизвести сложный поток выполнения, решающий множество задач, можно запустить по одному простому потоку для каждой из них.

Семь моделей

Обзор семи моделей, отобранных для этой книги, дает достаточно полное представление о методах реализации конкуренции и параллелизма.

Потоки выполнения и блокировки. Программирование с применением потоков и блокировок сопряжено с множеством хорошо известных проблем, однако эта технология лежит в основе многих других моделей, которые мы также будем рассматривать, и она все еще часто выбирается для реализации конкурентных программ.

Функциональное программирование. Функциональное программирование завоевывает все большую и большую популярность по множеству причин, из которых не самой маловажной является превосходная поддержка конкуренции параллелизма. Не имея изменчивого состояния, функциональные программы по своей природе оказываются потокобезопасными и легко поддаются распараллеливанию.

Путь Clojure – разделение идентичности и состояния. Язык Clojure продвигает особенно эффективную смесь императивного и функционального стилей программирования, усиливающую положительные качества обоих подходов.

Акторы. Модель акторов (actors) является универсальной моделью конкурентного программирования с весьма широкой областью применения. Она может применяться в любой из двух архитектур памяти – разделяемой и распределенной, – упрощает географическое распределение программных компонентов и обеспечивает особенно сильную поддержку надежности и отказоустойчивости.

Взаимодействие последовательных процессов. На первый взгляд, модель взаимодействия последовательных процессов (Communicating Sequential Processes, CSP) имеет много общего с моделью акторов – обе они основаны на передаче сообщений. Однако данная модель опирается на использование каналов для взаимодействий вместо обмена объектами, в результате чего программы на основе модели CSP оказываются совершенно иной разновидностью.

Параллелизм данных. В недрах вашего ноутбука спрятан самый настоящий суперкомпьютер. Графический процессор (GPU) широко использует параллелизм данных для ускорения обработки графики, но он способен решать более широкий круг задач. Если написать код, выполняющий вычисления методом конечных элементов, решающий задачи гидродинамики или производящий обработку больших объемов информации на GPU, его производительность затмит любые другие реализации.

Лямбда-архитектура. Обработать гигантские объемы данных, не прибегая к параллелизму, просто невыносимо – только пустив в ход все ресурсы компьютера, можно рассчитывать на успех в обработке терабайтов информации. Лямбда-архитектура объединяет в себе мощь технологии MapReduce и приемов потоковой обработки для создания платформы, на основе которой можно решать задачи обработки гигантских объемов данных.

Каждая из перечисленных моделей имеет свои достоинства. Принимаясь за чтение каждой из последующих глав, обращайтесь особое внимание на ответы на следующие вопросы:

- для решения каких задач предназначена данная модель – конкуренции, распараллеливания или обеих;
- какие параллельные архитектуры поддерживает данная модель;

- содержит ли данная модель инструменты, помогающие писать высоконадежные или географически распределенные приложения.

В следующей главе мы приступим к знакомству с первой моделью: потоками выполнения и блокировками.



ГЛАВА 2.

Потоки выполнения и блокировки

Механизм потоков выполнения и блокировок (threads-and-locks) можно сравнить с автомобилем Ford Model T¹. Этот автомобиль сможет доставить вас из точки А в точку Б, но он очень прост в конструкции, сложен в управлении, ненадежен и опасен в сравнении с современными автомобилями.

Несмотря на все сопутствующие проблемы, потоки выполнения и блокировки, тем не менее по-прежнему широко используются для создания конкурентных программ и к тому же лежат в основе многих других технологий, которые будут рассматриваться в этой книге. Даже если вы не планируете использовать их непосредственно, вам все же следует знать, как они работают.

Самое простое из того, что может работать

Потоки и блокировки – это не более чем формализация действий аппаратной части компьютера. В этом их главное достоинство и главный недостаток.

Благодаря их простоте потоки и блокировки поддерживаются практически всеми языками программирования в той или иной форме, и они накладывают не слишком много ограничений. Но они весьма сложны для начинающих программистов. Программы, написанные с применением потоков и блокировок, трудно читать и еще труднее поддерживать.

Далее мы будем знакомиться с программированием потоков и блокировок на примере языка Java, но принципы, излагаемые в этой гла-

¹ https://ru.wikipedia.org/wiki/Ford_Model_T – Прим. перев.

ве, можно распространить на любой другой язык, поддерживающий потоки выполнения. В первый день мы рассмотрим основы разработки многопоточного кода на Java, основные ловушки, с которыми вы можете столкнуться, а также некоторые правила, которые помогут вам избежать их. Во второй день мы двинемся дальше и займемся исследованием инструментов из пакета `java.util.concurrent`. Наконец, в третий день мы посмотрим на некоторые из конкурентных структур данных, поддерживаемых стандартной библиотекой, и попробуем задействовать их для решения практической задачи.

О наиболее эффективных приемах

В первую очередь мы познакомимся с низкоуровневыми потоками выполнения и примитивами блокировок в языке Java. В современном коде редко приходится использовать эти примитивы непосредственно, благодаря наличию высокоуровневых инструментов, о которых мы поговорим во второй и третий дни. Однако, чтобы понять, от чего зависят эти инструменты, необходимо знать и понимать, как действуют базовые механизмы. Именно поэтому мы начнем с основ. Но вы должны знать, что в программах практически нет необходимости напрямую использовать класс `Thread`.

День 1: взаимное исключение и модели памяти

Если прежде вам приходилось заниматься созданием конкурентных программ, вы наверняка знакомы с понятием *взаимного исключения* (*mutual exclusion*) – применения блокировок, чтобы гарантировать доступность данных только для одного потока выполнения в каждый конкретный момент времени. И вы также наверняка знакомы с проблемами, которые возникают при неправильном использовании блокировок, включая состояние гонки и взаимоблокировки (не волнуйтесь, если эти термины вам незнакомы – они описываются чуть ниже).

Это действительно очень большие проблемы, и мы потратим немало времени на их обсуждение, но, как оказывается, существует еще кое-что, не менее, если не более, важное, что следует знать для работы с разделяемой памятью, – Модель Памяти. И если вы думаете, что

состояние гонки и взаимоблокировки способны породить фантастически странное поведение программы, ваше мнение наверняка изменится, когда мы увидим, насколько странной и причудливой может быть разделяемая память.

Однако мы, кажется, пытаемся бежать впереди паровоза – давайте притормозим немного и для начала посмотрим, как создать поток.

Создание потока

Базовым элементом конкуренции в Java является *поток выполнения* (thread), который, как можно заключить из названия, состоит из единственного *потока управления*. Потоки могут взаимодействовать друг с другом посредством разделяемой (или совместно используемой) памяти.

Ни одна книга по программированию не обходится без примера программы «Hello, World!», поэтому не будем отступать от традиций и рассмотрим многопоточную версию такой программы:

ThreadsLocks/HelloWorld/src/main/java/com/paulbutcher/HelloWorld.java²

```
public class HelloWorld {  
  
    public static void main(String[] args) throws InterruptedException  
    {  
        Thread myThread = new Thread() {  
            public void run() {  
                System.out.println("Hello from new thread");  
            }  
        };  
  
        myThread.start();  
        Thread.yield();  
        System.out.println("Hello from main thread");  
        myThread.join();  
    }  
}
```

Этот код создает экземпляр класса `Thread` и затем запускает его. С этого момента метод `run()` нового потока будет выполняться конкурентно с остальным кодом в методе `main()`. Наконец, метод `join()` ждет завершения работы потока (что произойдет с завершением метода `run()`).

Если запустить эту программу, она выведет следующее:

² <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/HelloWorld/src/main/java/com/paulbutcher/HelloWorld.java>

```
Hello from main thread  
Hello from new thread
```

Или:

```
Hello from new thread  
Hello from main thread
```

В какой именно последовательности появятся сообщения, зависит от того, какой из потоков успеет первым вызвать функцию `println()` (когда я тестировал эту программу, у меня получилось, что обе ситуации примерно равновероятны). Такая зависимость от времени является одной из самых больших сложностей при программировании многопоточных программ – только потому, что вы наблюдали какое-то определенное поведение, запустив код, не означает, что оно будет проявляться постоянно.



Вопрос Джо: зачем нужен метод `Thread.yield()`?

В нашей многопоточной версии «Hello, World!» имеется следующая строка:

```
Thread.yield();
```

Согласно документации, метод `yield()`:

подсказывает планировщику, что текущий поток добровольно желает отдать остаток выделенного ему кванта процессорного времени.

Без этого вызова, из-за необходимости выполнения процедур инициализации нового потока, главный поток почти наверняка добрался бы до вызова `println()` первым (впрочем, это не гарантируется – как мы увидим далее, если в конкурентной программе что-то *может* произойти, рано или поздно это произойдет, причем в самый неподходящий момент).

Попробуйте закомментировать вызов этого метода и посмотрите, что из этого получится. Посмотрите также, что получится, если заменить вызов `Thread.yield()` вызовом `Thread.sleep(1)`?

Наша первая блокировка

Когда к разделяемой памяти обращаются сразу несколько потоков выполнения, они могут, говоря иносказательно, «оттоптать ноги» друг другу. Избежать этой неприятности можно с помощью *взаимоисключающей блокировки*, удерживать которую может только какой-то один поток.

Давайте создадим пару потоков, взаимодействующих друг с другом:

ThreadsLocks/Counting/src/main/java/com/paulbutcher/Counting.java³

```
public class Counting {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            private int count = 0;
            public void increment() { ++count; }
            public int getCount() { return count; }
        }
        final Counter counter = new Counter();
        class CountingThread extends Thread {
            public void run() {
                for(int x = 0; x < 10000; ++x)
                    counter.increment();
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.getCount());
    }
}
```

Здесь мы имеем очень простой класс `Counter` и два потока, каждый из которых вызывает метод `increment()` класса 10 000 раз. Очень просто и ... совершенно неправильно.

Попробуйте несколько раз запустить этот пример, и каждый раз вы будете получать разные результаты. Например, запустив пример три раза, я получил числа 13850, 11867 и 12616. Причина такого поведения – *состояние гонки* (race condition, поведение, зависящее от относительной продолжительности выполнения операций), возникающее из-за использования двумя потоками члена `count` класса `Counter`.

Если для вас это оказалось сюрпризом, давайте посмотрим, какой код получается в результате компиляции инструкции `++count`. Ниже приводится байт-код, сгенерированный компилятором:

```
getfield #2
iconst_1
iadd
putfield #2
```

³ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/Counting/src/main/java/com/paulbutcher/Counting.java>

Даже для тех, кто незнаком с байт-кодом JVM, совершенно очевидно, что здесь происходит: инструкция `getfield #2` извлекает значение `count`, инструкция `iconst_1` и следующая за ней инструкция `iadd` прибавляют 1 к этому значению, и инструкция `putfield #2` записывает результат обратно в `count`. Этот шаблон часто называют *прочитать–изменить–записать*.

Теперь представьте, что метод `increment()` был вызван двумя потоками одновременно. Поток 1 выполнит инструкцию `getfield #2` и получит значение 42. Прежде чем он сможет сделать что-то еще, поток 2 также выполнит инструкцию `getfield #2` и получит то же значение 42. Далее события начинают развиваться не по нашему сценарию, потому что оба потока увеличат на 1 одно и то же число 42, и оба запишут один и тот же результат, 43, обратно в член `count`. В конечном итоге получается, что были выполнены две операции увеличения на 1, а значение увеличилось только на 1.

Решить эту проблему можно путем *синхронизации* доступа к члену `count`. Одно из решений состоит в том, чтобы использовать *встроенную блокировку*, имеющуюся в каждом объекте Java (иногда ее называют *мьютекс*, *монитор* или *критическая секция*), объявив метод `increment()` синхронизированным:

```
ThreadsLocks/CountingFixed/src/main/java/com/paulbutcher/Counting.java4
```

```
class Counter {
private int count = 0;
> public synchronized void increment() { ++count; }
    public int getCount() { return count; }
}
```

Теперь метод `increment()` будет приобретать блокировку объекта `Counter` при вызове и освобождать ее в момент возврата управления. То есть в каждый конкретный момент времени тело метода будет выполняться только в каком-то одном потоке. Любые другие потоки, вызвавшие метод, будут заблокированы до момента освобождения блокировки (далее в этой главе мы увидим, что для таких простых случаев, когда в конкуренцию вовлечена единственная переменная, отличной альтернативой блокировке объекта может служить пакет `java.util.concurrent.atomic`).

Как и следовало ожидать, новая версия каждый раз будет возвращать верный результат, 20000.

⁴ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/CountingFixed/src/main/java/com/paulbutcher/Counting.java>

Однако не все так прекрасно – наш новый код все еще содержит одну малозаметную ошибку, причину которой мы обсудим далее.

Загадочная память

Давайте добавим немного загадочности. Как вы думаете, что выведет следующий код?

ThreadsLocks/Puzzle/src/main/java/com/paulbutcher/Puzzle.java⁵

```
1 public class Puzzle {
-   static boolean answerReady = false;
-   static int answer = 0;
-   static Thread t1 = new Thread() {
5       public void run() {
-           answer = 42;
-           answerReady = true;
-       }
-   };
10  static Thread t2 = new Thread() {
-       public void run() {
-           if (answerReady)
-               System.out.println("The meaning of life is: " + answer);
-           else
15               System.out.println("I don't know the answer");
-       }
-   };
-
-   public static void main(String[] args) throws InterruptedException {
20       t1.start(); t2.start();
-       t1.join(); t2.join();
-   }
- }
```

Если вы сразу подумали о состоянии гонки, то вы совершенно правы. Вы можете увидеть текст «The meaning of life is: 42» (число жизни: 42) или «I don't know the answer» (я не знаю ответ), в зависимости от того, в каком порядке будут выполняться потоки. Но это еще не все! Вы можете увидеть и такой вариант:

The meaning of life is: 0

Но как?! Как `answer` может иметь нулевое значение, когда `answerReady` уже имеет значение `true`, если такое возможно, только если поменять местами строки 6 и 7.

⁵ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/Puzzle/src/main/java/com/paulbutcher/Puzzle.java>

Не торопитесь с выводами. Как оказывается, такое действительно возможно, например по следующим причинам:

- в ходе статической оптимизации кода компилятор может изменять порядок выполнения операций;
- в ходе динамической оптимизации виртуальная машина JVM также может изменять порядок выполнения операций;
- микропроцессор, выполняющий код, тоже может выполнять операции в другом порядке.

Хуже того, иногда такие эффекты вообще невидимы для других потоков. Представьте, что мы переписали метод `run()` потока `t2`, как показано ниже:

```
public void run() {  
    while (!answerReady)  
        Thread.sleep(100);  
    System.out.println("The meaning of life is: " + answer);  
}
```

В этом случае наша программа может никогда не завершиться, потому что `answerReady` может никогда не получить значение `true`.

Если первой вашей реакцией было заставить компилятор, JVM и микропроцессор убрать свои «грязные руки» от вашего кода, вас можно понять. Но, к сожалению, это невозможно – значительная доля прироста производительности, который мы наблюдаем последние годы, как раз и обусловлена подобными оптимизациями. В частности, от них зависит быстродействие компьютеров с разделяемой памятью. Соответственно, нам остается только бороться со следствиями.

Очевидно, что такое положение вещей не может оставаться бесконтрольным – нам нужно что-то, что могло бы сообщить, на что мы можем положиться, а на что – нет. Для этого нам требуется познакомиться с моделью памяти в Java.

Видимость памяти

Модель памяти в Java определяет, когда изменения в памяти, выполненные одним потоком, должны быть *видимы* остальным потокам.⁶ Результатом является полное отсутствие каких-либо гарантий, если только оба потока – читающий и пишущий – не будут синхронизированы.

⁶ <http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4>

Мы уже видели один пример синхронизации, основанный на получении внутренней блокировки объекта. К числу других примеров синхронизации можно отнести запуск потока, ожидание завершения потока вызовом метода `join()` и применение множества классов из пакета `java.util.concurrent`.

Обратите особое внимание, что инструменты синхронизации должны использовать *оба* потока. Для решения проблемы недостаточно, если синхронизацию будет выполнять только поток, вносящий изменения. Это и есть причина малозаметной ошибки в коде. Чтобы избавиться от нее, одной только синхронизации метода `increment()` недостаточно – необходимо также синхронизировать метод `getCount()`. Если этого не сделать, поток, вызывающий `getCount()`, может увидеть устаревшее значение (так получилось, что в нашем примере метод `getCount()` не проявляет побочных эффектов, и только потому, что вызывается после вызова метода `join()`, но это – бомба замедленного действия, ждущая, когда кто-нибудь решит воспользоваться классом `Counter`).

Мы обсудили состояние гонки и видимость памяти – две основные проблемы, которые могут быть причиной неожиданного поведения многопоточной программы. Теперь перейдем к третьей проблеме: взаимоблокировке.

Несколько блокировок

Вас можно понять, если после прочтения предыдущих разделов вы решите, что единственный способ обезопасить себя в многопоточном мире – сделать все методы синхронизированными. К сожалению, все не так просто.

Во-первых, это было бы жутко неэффективно. Если синхронизировать каждый метод, большую часть своего времени потоки проводили бы в заблокированном состоянии, сводя на нет все преимущества конкурентного программирования. Но это одна из наименьших проблем – как только в программе появляется более одной блокировки (напомню, что в Java каждый объект имеет свою блокировку), открывается возможность взаимоблокировки потоков.

Продemonстрируем ситуацию взаимоблокировки на простом примере, часто используемом в академических статьях, посвященных проблемам реализации конкуренции – задаче «обедающих философов». Представьте пять философов, сидящих за круглым столом, и пять (не десять) палочек для еды, как показано ниже:

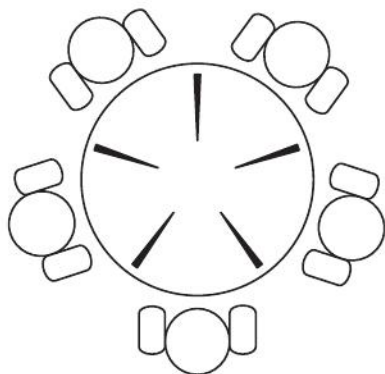


Рис. 3. Задача об обедающих философях

Каждый философ может или размышлять, или есть. Чтобы приступить к еде, он должен взять палочки с обеих сторон от себя, после чего он может поглощать пищу в течение некоторого времени (да, кстати, наши философы – мужчины; женщины ведут себя более разумно). Закончив, он кладет палочки на место.

Ниже показано, как можно было бы реализовать наших философов:

ThreadsLocks/DiningPhilosophers/src/main/java/com/paulbutcher/
Philosopher.java⁷

```
1 class Philosopher extends Thread {
-   private Chopstick left, right;
-   private Random random;
-
5   public Philosopher(Chopstick left, Chopstick right) {
-       this.left = left; this.right = right;
-       random = new Random();
-   }
-
10  public void run() {
-       try {
-           while(true) {
-               Thread.sleep(random.nextInt(1000)); // Размышлять
-               synchronized(left) {                // Взять палочку слева
15              synchronized(right) {                // Взять палочку справа
-                   Thread.sleep(random.nextInt(1000)); // Есть
-               }
-           }
-       }
-   }
```

⁷ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/DiningPhilosophers/src/main/java/com/paulbutcher/Philosopher.java>


```
-    }  
20    } catch (InterruptedException e) {}  
-    }  
- }
```

Строки 14 и 15 демонстрируют альтернативный способ приобретения внутренней блокировки объекта: `synchronized(object)`.

На моем компьютере эта программа благополучно выполнялась часами (я наблюдал за ее поведением в течение недели). А затем внезапно останавливалась.

Немного поразмыслив, легко можно понять причину – если все пять философов одновременно решат отвлечься от размышлений и приступить к еде, все они возьмут палочки слева и затем замрут – у каждого в левой руке будет одна палочка, и каждый будет ждать, пока освободится палочка, взятая философом справа. Возникает взаимоблокировка.

Опасность взаимоблокировки возникает всякий раз, когда поток пытается захватить более одной блокировки. К счастью, есть простое правило, гарантирующее избавление от взаимоблокировки: всегда приобретайте блокировки в глобальном фиксированном порядке.

Вот как это можно реализовать:

ThreadsLocks/DiningPhilosophersFixed/src/main/java/com/paulbutcher/
Philosopher.java⁸

```
class Philosopher extends Thread {  
> private Chopstick first, second;  
    private Random random;  
  
    public Philosopher(Chopstick left, Chopstick right) {  
>     if(left.getId() < right.getId()) {  
>         first = left; second = right;  
>     } else {  
>         first = right; second = left;  
>     }  
        random = new Random();  
    }  
    public void run() {  
        try {  
            while(true) {  
                Thread.sleep(random.nextInt(1000)); // Размышлять  
>                synchronized(first) {             // Взять первую палочку  
>                synchronized(second) {             // Взять вторую палочку  
                    Thread.sleep(random.nextInt(1000)); // Есть
```

⁸ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/DiningPhilosophersFixed/src/main/java/com/paulbutcher/Philosopher.java>

```

    }
    }
} catch (InterruptedException e) {}
}
}

```

Вместо того чтобы брать палочки слева, а затем справа, мы теперь берем первую и вторую палочки, используя член `id` класса `Chopstick`, чтобы гарантировать правильный порядок (нас не интересуют фактические идентификационные номера палочек – они нужны лишь для уникальности и упорядочения). Теперь можно не сомневаться, что взаимоблокировка больше не возникнет.



Вопрос Джо: можно ли использовать хэш-код объекта для упорядочения приобретения блокировок?

Вам часто будет встречаться совет использовать хэш-коды объектов для упорядочения приобретения блокировок, как показано ниже:

```

if (System.identityHashCode(left)
    < System.identityHashCode(right)) {
    first = left; second = right;
} else {
    first = right; second = left;
}

```

Преимущество такого подхода состоит в том, что он может применяться при работе с любыми объектами и позволяет избежать необходимости заводить дополнительное поле, упорядочивающее объекты. Но хэш-коды не гарантируют уникальности (весьма маловероятно, что два разных объекта будут иметь одинаковые хэш-коды, но такое может произойти). Поэтому, между нами говоря, я не стал бы использовать такое решение, разве только в случае, когда не остается ничего другого.

Как видите, совсем не сложно придерживаться правила глобального фиксированного порядка, когда приобретение блокировок выполняется в одном месте. Но в больших программах, где понятие глобальности слишком обширно, реализовать нечто подобное становится практически невозможно.

Опасности сторонних методов

Чтобы ослабить связь между модулями, в больших программах часто применяют прием обмена сообщениями, или событиями. Ниже, например, представлен класс, осуществляющий загрузку файла с указанным адресом URL и позволяющий регистрировать обработчики типа `ProgressListener`:

ThreadsLocks/HttpDownload/src/main/java/com/paulbutcher/Downloader.java⁹

```
class Downloader extends Thread {
    private InputStream in;
    private OutputStream out;
    private ArrayList<ProgressListener> listeners;

    public Downloader(URL url, String outputFilename) throws IOException {
        in = url.openConnection().getInputStream();
        out = new FileOutputStream(outputFilename);
        listeners = new ArrayList<ProgressListener>();
    }

    public synchronized void addListener(ProgressListener listener) {
        listeners.add(listener);
    }

    public synchronized void removeListener(ProgressListener listener) {
        listeners.remove(listener);
    }

    private synchronized void updateProgress(int n) {
        for (ProgressListener listener: listeners)
        >     listener.onProgress(n);
    }

    public void run() {
        int n = 0, total = 0;
        byte[] buffer = new byte[1024];

        try {
            while((n = in.read(buffer)) != -1) {
                out.write(buffer, 0, n);
                total += n;
                updateProgress(total);
            }
            out.flush();
        } catch (IOException e) { }
    }
}
```

Так как методы `addListener()`, `removeListener()` и `updateProgress()` синхронизированы, они могут вызываться множеством потоков без

⁹ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/HttpDownload/src/main/java/com/paulbutcher/Downloader.java>

риска «оттоптать друг другу ноги». Однако в этом коде кроется ловушка, которая может привести к взаимоблокировке, даже при том что используется всего одна блокировка.

Проблема в том, что `updateProgress()` вызывает *сторонний метод* – метод, о котором ничего не известно. Этот метод может делать что-то еще, в том числе пытаться приобрести другую блокировку. Если это так, тогда мы сталкиваемся с ситуацией приобретения двух блокировок, не зная, в правильном ли порядке они приобретаются. Как мы уже знаем, это может привести к взаимоблокировке.

Единственное решение данной проблемы – избегать вызовов сторонних методов при удержании блокировок. Одно из решений заключается в том, чтобы создать *защищенную копию* `listeners` перед выполнением итераций:

```
ThreadsLocks/HttpDownloadFixed/src/main/java/com/paulbutcher/  
Downloader.java10
```

```
private void updateProgress(int n) {  
    ArrayList<ProgressListener> listenersCopy;  
    synchronized(this) {  
        > listenersCopy = (ArrayList<ProgressListener>) listeners.clone();  
    }  
    for (ProgressListener listener: listenersCopy)  
        listener.onProgress(n);  
}
```

Этим изменением мы убиваем двух зайцев одним выстрелом: не только избегаем вызова стороннего метода при удержании блокировки, но еще и уменьшаем время, в течение которого удерживается блокировка. Удержание блокировки дольше, чем это необходимо, отрицательно сказывается на производительности (другие потоки будут дольше простаивать в ожидании снятия блокировки) и увеличивает вероятность взаимоблокировки. Кроме того, это изменение ликвидирует еще одну проблему, которая не связана с конкуренцией, – обработчик события может теперь вызвать `removeListener()` из своего метода `onProgress()`, что не повлияет на копию `listeners`, которая используется в цикле.

В завершение первого дня

Итак, мы подошли к концу первого дня. Мы охватили основные приемы создания многопоточного кода на языке Java, однако, как мы

¹⁰ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/HttpDownloadFixed/src/main/java/com/paulbutcher/Downloader.java>

увидим во второй день, стандартная библиотека включает альтернативы, которые часто оказываются более предпочтительными.

Что мы узнали в первый день

В первый день мы узнали, как создавать потоки и как использовать внутренние блокировки Java-объектов. Мы также познакомились с тремя основными проблемами потоков выполнения и блокировок – состояние гонки, взаимоблокировка и видимость памяти – и обсудили некоторые правила, помогающие их избежать:

- синхронизируйте доступ ко всем разделяемым переменным;
- синхронизируйте и читающие, и пишущие потоки выполнения;
- приобретайте несколько блокировок в фиксированном порядке;
- не вызывайте сторонних методов, удерживая блокировку;
- удерживайте блокировки не дольше, чем это необходимо.

Упражнения к первому дню

Найдите:

- и прочитайте статью «Java memory model» Уильяма Пью (William Pugh);
- ознакомьтесь с текстом «JSR 133 (Java memory model) FAQ»;¹¹
- какие гарантии дает модель памяти в Java относительно безопасности инициализации и всегда ли обязательно использовать блокировки для безопасной передачи объектов между потоками?
- в чем заключается суть антишаблона блокировки с двойной проверкой (double checked locking); почему этот прием назван антишаблоном?¹²

Задания:

- поэкспериментируйте с оригинальной версией решения задачи о пяти философях. Попробуйте изменить продолжительность периодов размышления и приема пищи, а также число философов. Как эти изменения отражаются на продолжительности периода до возникновения взаимоблокировки?

Представьте, что вы пытаетесь отладить эту программу и вам

¹¹ На русском языке: <http://habrahabr.ru/company/golovachcourses/blog/221133/> – Прим. перев.

¹² https://ru.wikipedia.org/wiki/Double_checked_locking – Прим. перев.

нужно повысить вероятность взаимоблокировки, – что бы вы предприняли?

- (сложное) напишите программу, которая демонстрировала бы кажущуюся перестановку операций записи в память в отсутствие синхронизации. Это непросто, потому что даже при том, что модель памяти в Java допускает такую перестановку, простые примеры не будут оптимизироваться до такой степени, чтобы действительно продемонстрировать проблему.

День 2: помимо встроенных блокировок

В первый день мы познакомились с классом `Thread` и встроенными блокировками, имеющимися во всех объектах в языке Java. В течение долгого времени эти два механизма оставались единственной поддержкой конкурентного программирования на Java. С выходом версии Java 5 положение дел в этой области существенно изменилось благодаря появлению пакета `java.util.concurrent`. Сегодня мы рассмотрим более совершенные механизмы блокировок, реализованные в этом пакете.

Встроенные блокировки удобны, но они имеют ряд ограничений:

- нет никакой возможности прервать работу потока, заблокированного при попытке приобрести встроенную блокировку;
- нет никакой возможности задать предельное время ожидания, в течение которого поток может оставаться заблокированным;
- существует только один способ приобрести встроенную блокировку: блок `synchronized`:

```
synchronized(object) {  
    использование разделяемых ресурсов  
}
```

Это означает, что приобретение и освобождение блокировки могут производиться только в пределах одного метода, и должны строго соблюдаться правила вложенности. Имейте в виду, что объявление метода как `synchronized` – это всего лишь синтаксический сахар, помогающий избавиться в исходном коде от заключения тела метода в инструкцию `synchronized`:

```
synchronized(this) {  
    тело метода  
}
```

Класс `ReentrantLock` позволяет преодолеть все эти ограничения путем явного использования методов приобретения и освобождения блокировки вместо применения инструкции `synchronized`.

Но, прежде чем переходить к обсуждению преимуществ класса `ReentrantLock` перед встроенными блокировками, посмотрим, как его использовать в качестве прямой замены инструкции `synchronized`:

```
Lock lock = new ReentrantLock();
lock.lock();
try {
    использование разделяемых ресурсов
} finally {
    lock.unlock();
}
```

Применение конструкции `try ... finally` – хорошая практика, гарантирующая освобождение блокировки при любом исходе, независимо от того, что произойдет в коде, выполняющемся под защитой блокировки.

А теперь посмотрим, как этот класс помогает преодолевать ограничения, свойственные встроенным блокировкам.

Прерываемое блокирование

Из-за того, что выполнение потока, заблокированного в попытке приобрести встроенную блокировку, нельзя прервать, мы лишены возможности защититься от ситуации взаимоблокировки. Увидеть это можно на простом примере, который воссоздает ситуацию взаимоблокировки и затем пытается прервать выполнение заблокированных потоков:

`ThreadsLocks/Uninterruptible/src/main/java/com/paulbutcher/Uninterruptible.java`¹³

```
public class Uninterruptible {

    public static void main(String[] args) throws InterruptedException {

        final Object o1 = new Object(); final Object o2 = new Object();

        Thread t1 = new Thread() {
            public void run() {
                try {
                    synchronized(o1) {
                        Thread.sleep(1000);
                    }
                }
            }
        };
        Thread t2 = new Thread() {
            public void run() {
                try {
                    synchronized(o2) {
                        Thread.sleep(1000);
                    }
                }
            }
        };
        t1.start();
        t2.start();
        t1.join();
        t2.join();
    }
}
```

¹³ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/Uninterruptible/src/main/java/com/paulbutcher/Uninterruptible.java>

```
        synchronized(o2) {}
    }
} catch (InterruptedException e) { System.out.println("t1
interrupted"); }
}
};

Thread t2 = new Thread() {
    public void run() {
        try {
            synchronized(o2) {
                Thread.sleep(1000);
                synchronized(o1) {}
            }
        } catch (InterruptedException e) { System.out.println("t2
interrupted"); }
    }
};

t1.start(); t2.start();
Thread.sleep(2000);
t1.interrupt(); t2.interrupt();
t1.join(); t2.join();
}
}
```

Эта программа заблокируется «намертво». Единственный способ завершить ее – принудительно остановить виртуальную машину JVM, под управлением которой она выполняется.



Вопрос Джо: неужели нет никакого способа остановить заблокированный поток?

Вы могли бы подумать, что должен быть хоть какой-то механизм, позволяющий остановить заблокированный поток. Но, как это ни печально, если вы так подумали, то оказались неправы. Все механизмы, которые я опробовал, не помогли достигнуть желаемого и ныне отмечены в документации как не рекомендуемые к использованию.*

Как результат единственный способ завершить поток выполнения в Java – вернуть управление из метода `run()` (возможно, в результате исключения `InterruptedException`). То есть если потоки оказались взаимозаблокированными на встроенной блокировке, значит, вам просто не повезло. Вы не сможете прервать их ожидание, и единственный способ завершить потоки – принудительно остановить виртуальную машину JVM.

* <http://docs.oracle.com/javase/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>

Впрочем, у нас есть выход. Мы можем реализовать наши потоки, задействовав в них `ReentrantLock` вместо встроенной блокировки, и прервать ожидание вызовом метода `lockInterruptibly()` этого класса:

ThreadsLocks/Interruptible/src/main/java/com/paulbutcher/Interruptible.java¹⁴

```
final ReentrantLock l1 = new ReentrantLock();
final ReentrantLock l2 = new ReentrantLock();

Thread t1 = new Thread() {
    public void run() {
        try {
            > l1.lockInterruptibly();
            Thread.sleep(1000);
            > l2.lockInterruptibly();
        } catch (InterruptedException e) { System.out.println("t1
interrupted"); }
    }
};
```

Эта версия просто завершает работу при вызове метода `Thread.interrupt()`. Применение дополнительных синтаксических конструкций в этом примере не кажется слишком дорогой ценой за возможность остановить взаимозаблокированные потоки.

Тайм-ауты

Класс `ReentrantLock` позволяет преодолеть еще одно ограничение встроенных блокировок: он дает возможность ограничить ожидание в попытке приобрести блокировку. В результате мы получаем альтернативный способ решить задачу о пяти философях.

Ниже демонстрируется класс `Philosopher`, который ожидает освождения обеих пар палочек не дольше заданного времени:

ThreadsLocks/DiningPhilosophersTimeout/src/main/java/com/paulbutcher/Philosopher.java¹⁵

```
class Philosopher extends Thread {
    private ReentrantLock leftChopstick, rightChopstick;
    private Random random;
```

¹⁴ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/Interruptible/src/main/java/com/paulbutcher/Interruptible.java>

¹⁵ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/DiningPhilosophersTimeout/src/main/java/com/paulbutcher/Philosopher.java>

```
public Philosopher(ReentrantLock leftChopstick, ReentrantLock
rightChopstick) {
    this.leftChopstick = leftChopstick; this.rightChopstick =
rightChopstick;
    random = new Random();
}

public void run() {
    try {
        while(true) {
            Thread.sleep(random.nextInt(1000)); // Размышлять
            leftChopstick.lock();
            try {
                if (rightChopstick.tryLock(1000, TimeUnit.MILLISECONDS)) {
                    // Взять палочку справа
                    try {
                        Thread.sleep(random.nextInt(1000)); // Есть
                    } finally { rightChopstick.unlock(); }
                } else {
                    // Палочку справа взять не удалось - вернуться к размышлениям
                }
            } finally { leftChopstick.unlock(); }
        }
    } catch (InterruptedException e) {}
}
```

Вместо метода `lock()` в этой версии используется метод `tryLock()`, который возвращает управление через заданный интервал времени, если приобрести блокировку не удалось. Это означает, что даже если мы не будем следовать правилу, требующему приобретать блокировки в фиксированном порядке, эта версия не будет попадать в ситуацию взаимоблокировки (по крайней мере, взаимоблокировка будет продолжаться не дольше установленного времени).

Зацикливание

Несмотря на то что метод `tryLock()` помогает избежать взаимоблокировки «навечно», это решение нельзя назвать хорошим. Во-первых, этот метод не избавляет от ситуации взаимоблокировки – он просто дает возможность выйти из нее. Во-вторых, он подвержен такому явлению, как *зацикливание* (livelock) – если все потоки заблокируются в один и тот же момент времени, есть вероятность, что по истечении тайм-аута они немедленно будут заблокированы вновь. Да, взаимоблокировка не будет продолжаться вечно, но тем не менее работа остановится.

Эту проблему можно облегчить, например если назначать всем потокам разное время тайм-аута, чтобы уменьшить вероятность одновременного выхода из ожидания. Но суть не в этом, а в том, что использование тайм-аутов само по себе редко является оптимальным решением, – предпочтительнее вообще избегать ситуации взаимоблокировки.

Блокирование методом перебора

Представьте, что нам нужно вставить запись в связанный список. Как вариант можно было бы защитить блокировкой весь список целиком, но это означало бы, что никакой другой поток не смог бы получить доступ к списку, пока блокировка не освободится. Альтернативным решением является *блокирование методом перебора*, когда блокируется доступ только к малой части списка, что позволяет другим потокам беспрепятственно обращаться к незаблокированным узлам. На рис. 4 показано, как действует блокирование методом перебора.

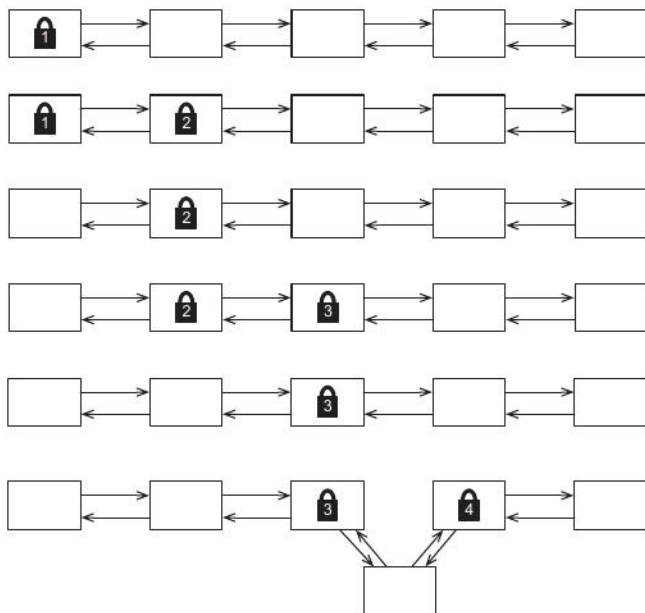


Рис. 4. Блокирование методом перебора

Чтобы вставить новый узел в список, требуется заблокировать доступ к двум узлам, находящимся с обеих сторон от точки вставки. Программа сначала блокирует первые два узла. Если это не та позиция, куда нужно вставить новый узел, программа разблокирует первый узел и блокирует третий. Если это опять не та позиция, программа разблокирует второй узел и блокирует четвертый. И так до тех пор, пока не будет достигнута нужная позиция. Затем программа вставляет новый узел и разблокирует узлы с обеих сторон.

Такая последовательность блокирования и разблокирования узлов невозможна при использовании встроенных блокировок, но ее легко реализовать с помощью класса `ReentrantLock`, потому что мы можем вызывать `lock()` и `unlock()` в любом месте программы. Ниже демонстрируется класс, реализующий сортированный список и описанный выше способ вставки новых узлов:

ThreadsLocks/LinkedList/src/main/java/com/paulbutcher/
ConcurrentSortedList.java¹⁶

```

1 class ConcurrentSortedList {
-
-   private class Node {
-       int value;
5       Node prev;
-       Node next;
-       ReentrantLock lock = new ReentrantLock();
-
-       Node() {}
10      Node(int value, Node prev, Node next) {
-          this.value = value; this.prev = prev; this.next = next;
-      }
-  }
15
-   private final Node head;
-   private final Node tail;
-
-   public ConcurrentSortedList() {
20       head = new Node(); tail = new Node();
-       head.next = tail; tail.prev = head;
-   }
-
-   public void insert(int value) {
25       Node current = head;
-       current.lock.lock();

```

¹⁶ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/LinkedList/src/main/java/com/paulbutcher/ConcurrentSortedList.java>

```

-     Node next = current.next;
-     try {
-         while (true) {
30         next.lock.lock();
-         try {
-             if (next == tail || next.value < value) {
-                 Node node = new Node(value, current, next);
-                 next.prev = node;
35                 current.next = node;
-                 return;
-             }
-         } finally { current.lock.unlock(); }
-         current = next;
40         next = current.next;
-     }
- } finally { next.lock.unlock(); }
- }

```

Метод `insert()` гарантирует, что список всегда будет отсортирован, отыскивая первый узел больше нового и выполняя вставку нового узла непосредственно перед ним.

В строке 26 приобретается блокировка на первый узел в списке, а в строке 30 – на следующий за ним. Затем проверяется, достигнуто ли место для вставки нового узла. Если нет, блокировка для текущего узла освобождается в строке 38, и цикл повторяется. В противном случае в строках 33–36 создается новый узел, вставляется в список, и управление возвращается вызывающему коду. Освобождение блокировок происходит в двух блоках `finally` (строки 38 и 42).

Такой подход можно использовать не только для реализации конкурентной вставки новых узлов, но и для выполнения других операций со списками. Например, ниже приводится метод, подсчитывающий число узлов в списке, – обход списка в обратном направлении выполняется только ради демонстрации:

ThreadsLocks/LinkedList/src/main/java/com/paulbutcher/
ConcurrentSortedList.java¹⁷

```

public int size() {
    Node current = tail;
    int count = 0;

    while (current.prev != head) {
        ReentrantLock lock = current.lock;

```

¹⁷ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/LinkedList/src/main/java/com/paulbutcher/ConcurrentSortedList.java>

```
lock.lock();  
try {  
    ++count;  
    current = current.prev;  
} finally { lock.unlock(); }  
}  
return count;  
}
```

Далее мы познакомимся с еще одной особенностью `ReentrantLock` — условными переменными (condition variables).



Вопрос Джо: не нарушает ли этот метод правило, требующее приобретать блокировки в фиксированном порядке?

Метод `insert()` класса `ConcurrentSortedList` приобретает блокировки, начиная с головы списка, и перемещается в сторону его хвоста. Метод `size()`, представленный выше, приобретает блокировки, начиная с хвоста списка, и перемещается в сторону его головы. Не нарушается ли в данном случае правило, требующее приобретать блокировки в фиксированном порядке?

Нет! Потому что метод `size()` никогда не удерживает более одной блокировки одновременно.

Условные переменные

В конкурентном программировании часто возникает необходимость дождаться некоторого события. Например, может потребоваться дождаться момента, когда в очереди появится хотя бы один элемент, прежде чем очистить ее, или когда в буфере появится свободное пространство для добавления в него нового элемента. Для решения подобных задач как раз и предназначены условные переменные (condition variables).

Для эффективного использования условной переменной желательно следовать довольно специфическому шаблону:

```
ReentrantLock lock = new ReentrantLock();  
Condition condition = lock.newCondition();  
  
lock.lock();  
try {
```

```
while (!"условие истинно")
    condition.await();
    «использовать разделяемые ресурсы»
} finally { lock.unlock(); }
```

Условная переменная связывается с блокировкой, и поток должен приобрести эту блокировку, прежде чем переходить к ожиданию условия. После получения блокировки поток проверяет, выполнено ли условие. Если да, тогда поток выполняет требуемые операции и освобождает блокировку.

В противном случае, если условие не выполнено, он вызывает метод `await()`, который *автоматически* освободит блокировку и заблокирует поток на условной переменной. С точки зрения другого потока, операция выполняется *атомарно*, то есть выглядит как единственная операция, которая либо выполняется, либо нет, — атомарная операция не может быть приостановлена на полпути для передачи управления другому потоку.

Когда другой поток вызовет метод `signal()` или `signalAll()`, чтобы сообщить, что теперь условная переменная содержит истинное значение, метод `await()` разблокирует поток и автоматически приобретет блокировку. Важно отметить, что когда `await()` вернет управление потоку, это означает лишь, что условие *может быть* истинным. Именно поэтому `await()` вызывается в цикле — поток должен вернуться назад, чтобы снова проверить истинность условия, и потенциально вновь может заблокироваться в вызове `await()`.

В результате мы получаем еще один вариант решения задачи о пяти философх:

ThreadsLocks/DiningPhilosophersCondition/src/main/java/com/paulbutcher/Philosopher.java¹⁸

```
class Philosopher extends Thread {

    private boolean eating;
    private Philosopher left;
    private Philosopher right;
    private ReentrantLock table;
    private Condition condition;
    private Random random;

    public Philosopher(ReentrantLock table) {
        eating = false;
    }
```

¹⁸ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/DiningPhilosophersCondition/src/main/java/com/paulbutcher/Philosopher.java>


```
this.table = table;
condition = table.newCondition();
random = new Random();
}

public void setLeft(Philosopher left) { this.left = left; }
public void setRight(Philosopher right) { this.right = right; }

public void run() {
    try {
        while (true) {
            think();
            eat();
        }
    } catch (InterruptedException e) {}
}

private void think() throws InterruptedException {
    table.lock();
    try {
        eating = false;
        left.condition.signal();
        right.condition.signal();
    } finally { table.unlock(); }
    Thread.sleep(1000);
}

private void eat() throws InterruptedException {
    table.lock();
    try {
        while (left.eating || right.eating)
            condition.await();
        eating = true;
    } finally { table.unlock(); }
    Thread.sleep(1000);
}
}
```

Это решение отличается от решений на основе единственной блокировки (`table`), что мы уже видели, и в нем отсутствует класс `Chopstick` (палочка). Вместо него мы используем тот факт, что философ может есть, только если его соседи слева и справа размышляют. Иными словами, проголодавшийся философ должен дожидаться следующего условия:

```
!(left.eating || right.eating)
```

Проголодавшийся философ сначала блокирует стол (`table`), чтобы никто из других философов не смог изменить своего состояния,

и затем проверяет – не ест ли кто-либо из его соседей. Если оба соседа находятся в размышлениях, данный философ приступает к еде и освобождает блокировку. В противном случае вызывается метод `await()` (который, как вы помните, автоматически освободит блокировку `table`).

Чтобы перейти к размышлениям после еды, философ должен сначала приобрести блокировку, установить признак `eating` в значение `false`, сообщить своим соседям, что они могут приступить к еде, и, наконец, освободить блокировку. Если кто-то из его соседей в данный момент ожидает возможности приступить к еде, его ожидание прерывается, блокировка захватывается, и выполняется проверка возможности начать прием пищи.

Хотя это решение выглядит сложнее, чем все остальные, которые мы видели выше, оно является более оптимальным. В предыдущих решениях нередко возникает ситуация, когда только один философ может есть, а остальные, удерживая одну палочку, вынуждены ждать, пока освободится вторая. В данном решении всякий раз, когда философ имеет теоретическую возможность поест (то есть когда оба его соседа размышляют), он сможет поест.

Это все, что мне хотелось рассказать о классе `ReentrantLock`, однако существует еще одна альтернатива встроенным блокировкам, о которой мы поговорим далее, – атомарные переменные (`atomic variables`).

Атомарные переменные

В первый день мы исправили ошибку в многопоточной реализации счетчика, сделав метод `increment()` синхронизированным. Как оказывается, пакет `java.util.concurrent.atomic` обеспечивает более удачную альтернативу:

ThreadsLocks/CountingBetter/src/main/java/com/paulbutcher/Counting.java¹⁹

```
public class Counting {
    public static void main(String[] args) throws InterruptedException {
        > final AtomicInteger counter = new AtomicInteger();

        class CountingThread extends Thread {
            public void run() {
                for(int x = 0; x < 10000; ++x)
```

¹⁹ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/CountingBetter/src/main/java/com/paulbutcher/Counting.java>

```
> counter.incrementAndGet();  
    }  
}  
  
CountingThread t1 = new CountingThread();  
CountingThread t2 = new CountingThread();  
  
t1.start(); t2.start();  
t1.join(); t2.join();  
  
System.out.println(counter.get());  
}  
}
```

Метод `incrementAndGet()` класса `AtomicInteger` функционально эквивалентен инструкции `++count` (в этом классе реализован также метод `getAndIncrement`, эквивалентный инструкции `count++`). Однако, в отличие от инструкции `++count`, этот метод выполняется атомарно.

Использование атомарных переменных вместо блокировок дает целый ряд преимуществ. Во-первых, отсутствует возможность забыть приобрести блокировку перед выполнением операций. Например, в этом коде не может возникнуть проблема, как в классе `Counter`, связанная с видимостью памяти, из-за того что `getCount()` не был синхронизирован. Во-вторых, благодаря отсутствию блокировок при использовании атомарных переменных исчезает вероятность попасть в ситуацию взаимоблокировки.

Наконец, атомарные переменные являются основой для *неблокирующих алгоритмов*, обеспечивающих синхронизацию без блокировок. Для тех, кому программирование с применением блокировок кажется чересчур сложным и кому хотелось бы писать код без блокировок, могу сказать, что пакет `java.util.concurrent` содержит все необходимые инструменты для этого. Но рассмотрением этих инструментов мы займемся в третий день, а пока закончим день второй.

В завершение второго дня

Опираясь на фундамент, заложенный в первый день, мы познакомились с более сложными и гибкими механизмами из пакетов `java.util.concurrent.locks` и `java.util.concurrent.atomic`. Однако, несмотря на важность знания этих механизмов, вам редко придется непосредственно использовать блокировки на практике, в чем вы убедитесь в третий день.



Вопрос Джо: а что можно сказать о `volatile`-переменных?

В языке Java есть возможность объявить переменную со спецификатором `volatile`. Операции чтения и записи с такими переменными не будут переупорядочиваться. Мы могли бы исправить реализацию класса `Puzzle` (см. выше), объявив переменную `answerReady` как `volatile`.

Проблема в том, что спецификатор `volatile` является очень слабой формой синхронизации, и его применение, например, не могло бы нам исправить проблему в классе `Counter`, потому что объявление переменной `count` со спецификатором `volatile` не гарантирует атомарности операции `count++`.

В наши дни, когда оптимизация виртуальных машин JVM достигла небывалых высот и блокировки имеют весьма низкие накладные расходы, необходимость в `volatile`-переменных возникает намного реже. Если вы вдруг задумаетесь об использовании спецификатора `volatile`, попробуйте для начала воспользоваться классами из пакета `java.util.concurrent.atomic`.

Что мы узнали во второй день

Мы увидели, как класс `ReentrantLock` и пакет `java.util.concurrent.atomic` способны помочь в преодолении ограничений, свойственных встроенным блокировкам, чтобы потоки выполнения могли:

- прерываться во время попытки приобрести блокировку;
- ожидать освобождения блокировки не дольше установленного времени;
- приобретать и освобождать блокировки в любом порядке;
- использовать условные переменные для ожидания произвольных событий;
- вообще не использовать блокировки за счет применения условных переменных.

Упражнения ко второму дню

Найдите:

- конструктор класса `ReentrantLock` поддерживает необязательный параметр `fair` (от англ. *fairness* – честность, справедливость. – *Прим. перев.*). Что подразумевается под словом «справедливость» в отношении блокировок? Когда могут при-

годиться «справедливые» блокировки? По каким причинам вы могли бы отказаться от использования «справедливых» блокировок?

- познакомьтесь с классом `ReentrantReadWriteLock`. Чем он отличается от класса `ReentrantLock`? В каких случаях его можно было бы использовать?
- что такое «случайное пробуждение» (*spurious wakeup*)? Когда оно происходит и почему в хорошо продуманной программе нет причин беспокоиться об этом явлении?
- познакомьтесь с классом `AtomicIntegerFieldUpdater`. Чем он отличается от класса `AtomicInteger`? В каких случаях его можно было бы использовать?

Задания:

- подумайте, что произойдет, если в версии решения о пяти философях на основе условной переменной цикл `while` заменить простой инструкцией `if`? Какие ошибки могут встретиться? Что случится, если вызов `signal()` заменить вызовом `signalAll()`? К каким проблемам это может привести (если может)?
- встроенные блокировки как более ограниченный аналог класса `ReentrantLock` также поддерживают более ограниченную разновидность условных переменных. Перепишите решение задачи о пяти философях, используя встроенные блокировки и их методы `wait()` и `notify()` или `notifyAll()`. Почему такая реализация оказывается менее эффективной, чем на основе `ReentrantLock`?
- напишите версию класса `ConcurrentSortedList`, которая использовала бы единственную блокировку вместо приема блокирования методом перебора. Сравните производительность обеих версий. Дает ли прием блокирования методом перебора прирост производительности? В каких ситуациях этот прием мог бы оказаться более предпочтительным, а в каких – менее?

День 3: на плечах гигантов

Помимо улучшенных блокировок, с которыми мы познакомились во второй день, пакет `java.util.concurrent` содержит множество универсальных, высокопроизводительных и хорошо отлаженных

конкурентных структур данных и утилит. Сегодня мы увидим, что в большинстве случаев они оказываются более удачным выбором, чем разработка собственных решений.

Еще раз о создании потоков

В первый день мы видели, как запускать дополнительные потоки, но, как оказывается, на практике редко возникает необходимость создавать потоки непосредственно. Ниже приводится пример реализации простого сервера, который возвращает все, что было ему послано:

ThreadsLocks/EchoServer/src/main/java/com/paulbutcher/EchoServer.java²⁰

```
public class EchoServer {

    public static void main(String[] args) throws IOException {

        class ConnectionHandler implements Runnable {
            InputStream in; OutputStream out;
            ConnectionHandler(Socket socket) throws IOException {
                in = socket.getInputStream();
                out = socket.getOutputStream();
            }

            public void run() {
                try {
                    int n;
                    byte[] buffer = new byte[1024];
                    while((n = in.read(buffer)) != -1) {
                        out.write(buffer, 0, n);
                        out.flush();
                    }
                } catch (IOException e) {}
            }
        }

        ServerSocket server = new ServerSocket(4567);
        while (true) {
            > Socket socket = server.accept();
            > Thread handler = new Thread(new ConnectionHandler(socket));
            > handler.start();
        }
    }
}
```

²⁰ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/EchoServer/src/main/java/com/paulbutcher/EchoServer.java>

Выделенные строки реализуют типичный шаблон – они принимают входящее соединение и сразу же создают поток для его обработки. Такое решение прекрасно работает, но оно страдает парой проблем. Во-первых, несмотря на то что операция создания потока относительно недорога, она все же имеет определенную цену, и в данном решении программа будет платить эту цену для каждого соединения. Во-вторых, эта программа будет создавать столько потоков, сколько имеется входящих соединений, – при быстром нарастании числа соединений число потоков может вырасти до такого уровня, что сервер практически остановится и, возможно даже, завершится аварийно. Это будет приятным открытием для любого, кто пожелает подвергнуть ваш сервер атаке вида «отказ в обслуживании» (denial of service).

Мы можем избежать этих проблем, создав пул потоков:

```
ThreadsLocks/EchoServerBetter/src/main/java/com/paulbutcher/  
EchoServer.java21
```

```
int threadPoolSize = Runtime.getRuntime().availableProcessors() * 2;  
ExecutorService executor = Executors.newFixedThreadPool(threadPoolSize);  
while (true) {  
    Socket socket = server.accept();  
    executor.execute(new ConnectionHandler(socket));  
}
```

Этот фрагмент создаст пул с числом потоков, в два раза превышающим число процессоров в системе. Если количество входящих соединений превысит это число, они будут ставиться в очередь и ждать, пока не освободится какой-нибудь из имеющихся потоков. Такое решение не только избавляет от накладных расходов на создание потоков для обработки новых соединений, но и гарантирует сохранение работоспособности сервера даже при высоких нагрузках (соединения хоть и не будут обрабатываться достаточно быстро, но, по крайней мере, некоторые из них будут обработаны).

Копирование при записи

В первый день мы узнали, как в конкурентных программах можно безопасно вызывать обработчики. Как вы наверняка помните, в методе `updateProgress()` мы создавали защитную копию. Как оказывается, в стандартной библиотеке Java уже имеется готовое и более оптимальное решение: `CopyOnWriteArrayList`:

²¹ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/EchoServerBetter/src/main/java/com/paulbutcher/EchoServer.jav>



Вопрос Джо: насколько большим должен быть пул потоков?

Оптимальное число потоков может различаться в зависимости от числа ядер в системе, от характера потоков (являются ли они преимущественно потоками ввода/вывода или занимаются исключительно вычислениями), от наличия других задач, решаемых системой, и многих других факторов.

Однако в общем случае для вычислительных задач лучше выбирать число потоков по числу ядер. Для задач ввода/вывода число потоков можно увеличить.

Но самый лучший вариант состоит в том, чтобы воссоздать реальную нагрузку на систему и определить оптимальное число потоков экспериментальным путем, с секундомером в руках.

ThreadsLocks/HttpDownloadBetter/src/main/java/com/paulbutcher/Downloader.java²²

```
private CopyOnWriteArrayList<ProgressListener> listeners;

public void addListener(ProgressListener listener) {
    listeners.add(listener);
}

public void removeListener(ProgressListener listener) {
    listeners.remove(listener);
}

private void updateProgress(int n) {
    for (ProgressListener listener: listeners)
        listener.onProgress(n);
}
```

Как следует из его имени, класс `CopyOnWriteArrayList` использует нашу стратегию защищенной копии. Только вместо создания копии перед итерациями по списку он создает копию перед его изменением. Любые имеющиеся итераторы будут продолжать ссылаться на предыдущую копию. Этот подход пригоден не для всех ситуаций, но отлично подходит в данном случае.

Во-первых, как можно видеть в листинге выше, код получается более ясным и кратким. Фактически от первой, простейшей и небезопасной версии, он отличается только определением списка `listeners`. Во-вторых, это решение оказывается более эффективным, потому что

²² <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/HttpDownloadBetter/src/main/java/com/paulbutcher/Downloader.java>

избавляет нас от необходимости создавать копию в каждом вызове `updateProgress()` — копии теперь создаются лишь при изменении списка обработчиков (что происходит намного реже).

Законченная программа

До сих пор мы рассматривали инструменты по отдельности. В оставшейся части этой главы мы займемся решением небольшой, но имеющей практическую ценность проблемы: мы попробуем узнать, какое слово чаще всего встречается в Википедии.

Сама по себе задача несложная — нужно загрузить дамп в формате XML и написать программу, которая будет извлекать из него текст и подсчитывать слова.²³ Учитывая, что дамп имеет объем порядка 40 Гбайт, его анализ может занять значительное время. Можем ли мы как-то ускорить это за счет распараллеливания задачи?

Начнем с самого начала — как долго простая последовательная программа будет обрабатывать первые 100 000 страниц?

ThreadsLocks/WordCount/src/main/java/com/paulbutcher/WordCount.java²⁴

```
public class WordCount {
    private static final HashMap<String, Integer> counts =
        new HashMap<String, Integer>();

    public static void main(String[] args) throws Exception {
        Iterable<Page> pages = new Pages(100000, "enwiki.xml");
        for (Page page: pages) {
            Iterable<String> words = new Words(page.getText());
            for (String word: words)
                countWord(word);
        }

        private static void countWord(String word) {
            Integer currentCount = counts.get(word);
            if (currentCount == null)
                counts.put(word, 1);
            else
                counts.put(word, currentCount + 1);
        }
    }
}
```

На моем MacBook Pro эта программа выполняется за 105 секунд.

²³ <http://dumps.wikimedia.org/enwiki/>

²⁴ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCount/src/main/java/com/paulbutcher/WordCount.java>

Итак, нам нужно решить, какие операции могут выполняться параллельно. В каждой итерации главного цикла решаются две задачи – парсинг данных в формате XML для получения страницы в виде объекта `Page` и подсчет слов в этой странице.

Задачи подобного типа решаются по классическому шаблону – *производитель–потребитель* (producer-consumer). Вместо единственного потока, который сначала «производит» страницу, а затем «потребляет», мы создадим два потока: поток-производитель и поток-потребитель.

Ниже приводится класс `Parser`, реализующий «производителя»:

```
ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/
Parser.java25
```

```
class Parser implements Runnable {
    private BlockingQueue<Page> queue;

    public Parser(BlockingQueue<Page> queue) {
        this.queue = queue;
    }

    public void run() {
        try {
            > Iterable<Page> pages = new Pages(100000, "enwiki.xml");
            > for (Page page: pages)
            >     queue.put(page);
            > } catch (Exception e) { e.printStackTrace(); }
        }
    }
}
```

Метод `run()` включает внешний цикл от первоначального последовательного решения, но вместо подсчета слов во вновь полученной странице он добавляет ее в хвост очереди.

Далее следует реализация «потребителя»:

```
ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/
Counter.java26
```

```
class Counter implements Runnable {
    private BlockingQueue<Page> queue;
    private Map<String, Integer> counts;

    public Counter(BlockingQueue<Page> queue,
```

²⁵ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Parser.java>

²⁶ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Counter.java>


```

        Map<String, Integer> counts) {
    this.queue = queue;
    this.counts = counts;
}

public void run() {
    try {
        while(true) {
>         Page page = queue.take();
>         if (page.isPoisonPill())
>             break;

>         Iterable<String> words = new Words(page.getText());
>         for (String word: words)
>             countWord(word);
        }
    } catch (Exception e) { e.printStackTrace(); }
}
}

```

Как и следовало ожидать, она содержит внутренний цикл из первоначальной последовательной версии, который извлекает данные для анализа из очереди.

Наконец, ниже приводится измененная версия главного цикла, создающая два потока:

ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/WordCount.java²⁷

```

ArrayBlockingQueue<Page> queue = new ArrayBlockingQueue<Page>(100);
HashMap<String, Integer> counts = new HashMap<String, Integer>();

Thread counter = new Thread(new Counter(queue, counts));
Thread parser = new Thread(new Parser(queue));

counter.start();
parser.start();
parser.join();
queue.put(new PoisonPill());
counter.join();

```

Класс `ArrayBlockingQueue` из пакета `java.util.concurrent` — это конкурентная очередь, идеально подходящая для реализации шаблона производитель–потребитель. Она не только поддерживает эффективные и конкурентные методы `put()` и `take()`, но также блокирует

²⁷ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/WordCount.java>

их, когда это необходимо. Попытка вызвать метод `take()` для пустой очереди заблокирует поток до момента, пока в очереди не появится хотя бы один элемент, точно так же попытка вызвать метод `put()` для переполненной очереди заблокирует поток до момента, пока в ней не появится свободное место.



Вопрос Джо: зачем блокировать доступ к очереди?

Наряду с блокирующими очередями в пакете `java.util.concurrent` имеется класс `ConcurrentLinkedQueue`, реализующий неблокирующую очередь. Похоже, что именно такая очередь нам и нужна, так почему же она не подходит для данной задачи?

Проблема в том, что производитель и потребитель иногда (точнее, практически всегда) могут работать с разной скоростью. Например, если производитель работает быстрее потребителя, очередь будет становиться все больше и больше. Учитывая, что дампы Википедии имеют размер порядка 40 Гбайт, разность скоростей может привести к тому, что очередь просто не поместится в памяти.

Применение блокирующей очереди, напротив, позволит производителю опередить потребителя лишь ненамного.

Другой интересный аспект этого решения – определение потребителем момента завершения работы:

```
ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Counter.java28
```

```
if (page.isPoisonPill())  
    break;
```

Как следует из имени метода, *poison pill* (пилюля с ядом) – это особый элемент в очереди, указывающий, что данные для обработки закончились, и потребитель должен завершить работу. Это очень похоже на то, как в C/C++ используется символ `null` для обозначения конца строки.

Могу вас порадовать – нам удалось сократить время работы программы со 105 до 95 секунд.

Неплохо, но может быть еще лучше. Прелесть шаблона производитель–потребитель в том, что он позволяет не только выполнять

²⁸ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountProducerConsumer/src/main/java/com/paulbutcher/Counter.java>

производство и потребление параллельно, но также запустить сразу несколько производителей и/или потребителей.

Однако как определить, на чем следует сосредоточиться? На производителе или на потребителе?

Какой код выполняется дольше? Временно изменив код так, чтобы выполнялся один только производитель, я определил, что парсинг первых 100 000 страниц занимает примерно 10 секунд.

Если прикинуть, полученный результат легко объяснить. Итак, первоначальная последовательная версия справлялась с работой за 105 секунд, а конкурентная версия справляется за 95 секунд. Парсинг занимает 10 секунд, соответственно, подсчет слов – 95 секунд. Парсинг и подсчет слов выполняются параллельно, то есть за счет параллельного выполнения парсинга и подсчета слов мы получаем общее время, равное наибольшей операции, – в данном случае 95 секунд.

Таким образом, чтобы еще больше увеличить производительность, следует распараллелить процесс подсчета и создать несколько потребителей. На рис. 5 показано, как можно реализовать такую схему.

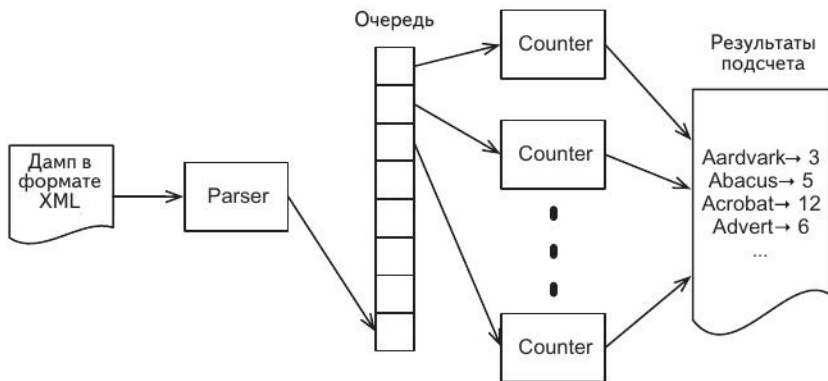


Рис. 5. Один производитель, несколько потребителей

Так как мы собираемся выполнять подсчет слов в нескольких потоках параллельно, нам нужно синхронизировать доступ к ассоциативному массиву `counts`.

Первое, что приходит в голову, – задействовать *синхронизированный ассоциативный массив*, который можно создать вызовом метода `synchronizedMap()` класса `Collections`. К сожалению, синхронизированные коллекции не поддерживают атомарных методов для чтения–изменения–записи, поэтому нужно что-то другое. Если мы все еще

хотим использовать хэш-массив типа `HashMap`, нам нужно синхронизировать доступ к нему самостоятельно.

Ниже приводится измененная версия метода `countWord()`, которая делает все, что нам требуется:

```
ThreadsLocks/WordCountSynchronizedHashMap/src/main/java/com/
paulbutcher/Counter.java29
```

```
private void countWord(String word) {
> lock.lock();
    try {
        Integer currentCount = counts.get(word);
        if (currentCount == null)
            counts.put(word, 1);
        else
            counts.put(word, currentCount + 1);
    } finally { lock.unlock(); }
}
```

А вот так выглядит измененная версия главного цикла, запускающая несколько потоков-потребителей:

```
ThreadsLocks/WordCountSynchronizedHashMap/src/main/java/com/
paulbutcher/WordCount.java30
```

```
ArrayBlockingQueue<Page> queue = new ArrayBlockingQueue<Page>(100);
HashMap<String, Integer> counts = new HashMap<String, Integer>();
ExecutorService executor = Executors.newCachedThreadPool();
for (int i = 0; i < NUM_COUNTERS; ++i)
    executor.execute(new Counter(queue, counts));
Thread parser = new Thread(new Parser(queue));
parser.start();
parser.join();
for (int i = 0; i < NUM_COUNTERS; ++i)
    queue.put(new PoisonPill());
executor.shutdown();
executor.awaitTermination(10L, TimeUnit.MINUTES);
```

Очень похоже на предыдущую версию, за исключением перехода на использование пула потоков, который упрощает управление множеством потоков. Отметьте также, что в этой версии важно не ошибиться и добавить соответствующее число «пилюль с ядом», чтобы обеспечить нормальное завершение работы.

²⁹ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountSynchronizedHashMap/src/main/java/com/paulbutcher/Counter.java>

³⁰ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountSynchronizedHashMap/src/main/java/com/paulbutcher/WordCount.java>

Теоретически все выглядит безупречно, но наши надежды разбились о действительность. Ниже показано, сколько времени действует программа с одним и с двумя потребителями (увеличение скорости измерено относительно последовательной версии):

Потребителей	Время (сек.)	Увеличение скорости
1	101	1.04
2	212	0.49

Почему добавление еще одного потребителя привело к замедлению? И не к небольшому, а к существенному замедлению, более чем вдвое?

Ответ на этот вопрос прост: из-за излишней *состязательности* — слишком много потоков пытается одновременно использовать единственный разделяемый ресурс. В нашем случае потоки-потребители блокируют доступ к ассоциативному массиву `counts` друг для друга и тратят чересчур много времени на ожидание снятия блокировки, тогда как могли бы потратить его на более полезную работу. Это обстоятельство и привело к такому снижению производительности.

К счастью, эта проблема разрешима. Похоже, что класс `ConcurrentHashMap` из `java.util.concurrent` является именно тем, что нам нужно. Он не только предоставляет методы, выполняющие атомарно операцию чтения–изменения–записи, но и был спроектирован для поддержки конкурентного доступа (с применением приема сокращения областей конкуренции между потоками (*lock striping*)).

Далее приводится измененная версия `countWord()`, использующая `ConcurrentHashMap`:

ThreadsLocks/WordCountConcurrentHashMap/src/main/java/com/paulbutcher/Counter.java³¹

```
private void countWord(String word) {
    while (true) {
        Integer currentCount = counts.get(word);
        if (currentCount == null) {
            if (counts.putIfAbsent(word, 1) == null)
                break;
        } else if (counts.replace(word, currentCount, currentCount + 1)) {
            break;
        }
    }
}
```

³¹ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountConcurrentHashMap/src/main/java/com/paulbutcher/Counter.java>


```
}  
}  
}
```

Давайте разберем эту версию метода подробнее, чтобы лучше понять, как он действует. Вместо метода `put()` теперь используется комбинация методов `putIfAbsent()` и `replace()`. Ниже приводится выдержка из документации с описанием метода `putIfAbsent()`:

Связывает ключ `key` с указанным значением `value`. Если ключ отсутствует, он будет создан, примерно как показано ниже:

```
if (!map.containsKey(key))  
    return map.put(key, value);  
else  
    return map.get(key);
```

Данная операция выполняется атомарно.

И выдержка с описанием `replace()`:

Связывает с ключом `key` новое значение `newValue`, только если текущее значение ключа равно указанному значению `oldValue`:

```
if (map.containsKey(key) && map.get(key).equals(oldValue)) {  
    map.put(key, newValue);  
    return true;  
} else return false;
```

Данная операция выполняется атомарно.

Как следует из описания, всякий раз, вызывая эти функции, мы должны проверять возвращаемые ими значения, чтобы убедиться в успехе выполняемых операций. Если какой-то из методов вернет `false`, следует вернуться и повторить попытку.

Результаты хронометража этой версии выглядят куда как более вдохновляющими:

Потребителей	Время (сек.)	Увеличение скорости
1	120	0,81
2	83	1,26
3	65	1,61
4	63	1,67
5	70	1,50
6	79	1,33

Ура! На этот раз увеличение числа потребителей привело к увеличению производительности, по крайней мере пока их количество не стало больше четырех. После этого дальнейшее увеличение числа потребителей вновь стало приводить к замедлению.

Да, результат в 63 секунды намного лучше, чем 105 секунд, которые тратит последовательная версия программы, но нам не удалось достичь даже двукратного увеличения производительности. Мой MacBook оснащен процессором с четырьмя ядрами – разве не должны мы были получить ускорение, близкое к четырехкратному?

Немного поразмыслив, легко заметить, что наше решение порождает излишнюю состязательность за массив `counts`. Вместо того чтобы конкурентно обновлять единственное разделяемое множество счетчиков, каждый поток-потребитель мог бы поддерживать собственный массив. А после окончания вычислений можно просто объединить все эти массивы в один:

ThreadsLocks/WordCountBatchConcurrentHashMap/src/main/java/com/paulbutcher/Counter.java³²

```
private void mergeCounts() {
    for (Map.Entry<String, Integer> e: localCounts.entrySet()) {
        String word = e.getKey();
        Integer count = e.getValue();
        while (true) {
            Integer currentCount = counts.get(word);
            if (currentCount == null) {
                if (counts.putIfAbsent(word, count) == null)
                    break;
            } else if (counts.replace(word, currentCount, currentCount + count)) {
                break;
            }
        }
    }
}
```

Эта версия показывает результат, более близкий к четырехкратному ускорению:

Потребителей	Время (сек.)	Увеличение скорости
1	95	1,10
2	57	1,83
3	40	2,62

³² <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/WordCountBatchConcurrentHashMap/src/main/java/com/paulbutcher/Counter.java>

Потребителей	Время (сек.)	Увеличение скорости
4	39	2,69
5	35	2,96
6	33	3,14
7	41	2,55

Эта версия не только показывает более быстрый прирост производительности с увеличением числа потоков-потребителей, но и продолжает давать прирост даже после того, как число потоков превысило число физических ядер. Это возможно потому, что каждое ядро процессора моего MacBook поддерживает два виртуальных ядра (hyperthreads) – метод `availableProcessors()` фактически возвращает число 8, тогда как в действительности имеется всего четыре физических ядра.

На рис. 6 изображен график изменения производительности всех трех версий. Эту картину вы будете наблюдать снова и снова при работе с программами, реализующими параллельные алгоритмы. С увеличением числа потоков производительность сначала линейно возрастает, затем рост замедляется, и, наконец, производительность начинает падать, причем дальнейшее увеличение числа потоков только ухудшает ситуацию.

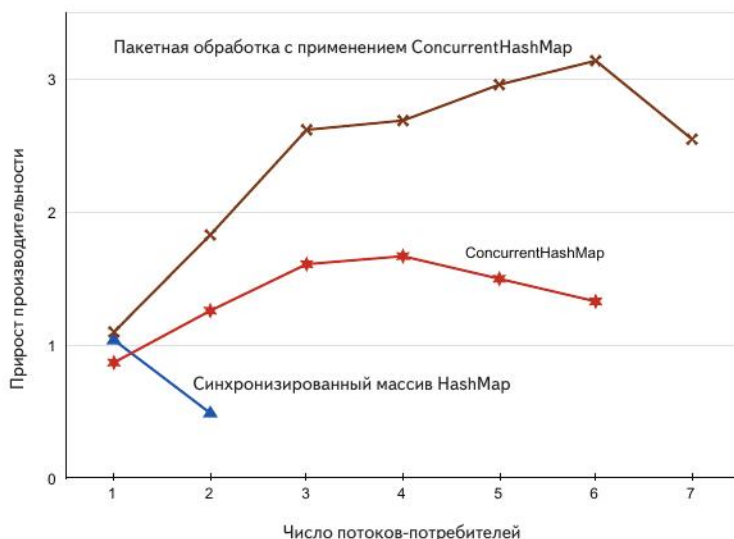


Рис. 6. Производительность программы подсчета слов в зависимости от числа потоков-потребителей

Давайте приостановимся и поразмышляем о том, что было сделано. Мы создали относительно сложную программу на основе шаблона производитель–потребитель, в которой множество потоков-потребителей координируют свои действия друг с другом посредством конкурентной очереди и конкурентного ассоциативного массива. При этом мы не использовали блокировки явно, задействовав в программе механизмы, предоставляемые стандартной библиотекой.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения приемов программирования с применением потоков выполнения и блокировок.

Что мы узнали в третий день

Мы узнали, какие инструменты имеются в пакете `java.util.concurrent`, которые не только упрощают создание конкурентных программ, но также делают эти программы надежнее и эффективнее, за счет:

- использования пулов потоков вместо непосредственного их создания;
- создания более простого и эффективного кода, управляющего обработчиками с помощью `CopyOnWriteArrayList`;
- эффективной организации взаимодействий между потоками-производителями и потоками-потребителями посредством `ArrayBlockingQueue`;
- использования поддержки конкурентного доступа к ассоциативному массиву в виде класса `ConcurrentHashMap`.

Упражнения к третьему дню

Найдите:

- документацию с описанием `ForkJoinPool` – чем отличается этот класс от пула потоков? В каких случаях предпочтительнее использовать `ForkJoinPool`, а в каких пул потоков?
- что такое *захват задачи* (work-stealing) и в каких ситуациях может пригодиться этот алгоритм? Как бы вы реализовали алгоритм захвата задачи с применением инструментов из пакета `java.util.concurrent`?
- перечислите различия между `CountDownLatch` и `CyclicBarrier`? В каких случаях можно было бы использовать каждый из них?

- что такое закон Адмала? Что он говорит о максимальном увеличении производительности, которого теоретически можно достичь в нашей программе подсчета слов?

Задания:

- перепишите код, реализующий шаблон производитель–потребитель, используя отдельный флаг «конец данных» вместо «пилюли с ядом». Убедитесь, что ваше решение корректно работает в случаях, когда производитель действует быстрее потребителя, и наоборот. Что произойдет, если потребитель уже попытается удалить что-либо из очереди в момент установки флага «конец данных»? Почему, по вашему мнению, решение на основе «пилюли с ядом» получило такое широкое признание?
- опробуйте разные версии программы подсчета слов на своем компьютере, а также на любых других, доступных вам. Насколько сильно отличаются графики производительности на разных компьютерах? Как вы думаете, если бы у вас был компьютер с 32 ядрами, смогли бы вы приблизиться к 32-кратному приросту производительности?

В завершение

Модель программирования с применением потоков и блокировок является, пожалуй, самой противоречивой из всех, что рассматриваются в этой книге. Она считается одной из самых сложных в реализации, из-за чего многие программисты стараются избегать ее любой ценой. Другие не понимают, из-за чего весь сыр-бор, – существует всего несколько простых правил, следование которым делает программирование потоков не сложнее программирования обычных последовательных программ.

Рассмотрим сильные и слабые стороны этой модели.

Сильные стороны

Главным преимуществом потоков и блокировок является широкая их применимость. Как вы уже наверняка понимаете, данная модель лежит в основе многих других моделей, рассматриваемых в этой книге, соответственно, ее можно использовать для самых разных задач. Благодаря «близости к железу» данная модель, которая является всего лишь формализацией действий аппаратной части компьютера,

может быть очень эффективной при правильном использовании. Из этого следует, что потоки и блокировки можно использовать для решения проблем с практически любой степенью детализации, от мелкой до грубой.

Кроме того, данная модель легко интегрируется в большинство языков программирования. От создателей императивных и объектно-ориентированных языков требуется совсем немного усилий, чтобы добавить поддержку потоков и блокировок.

Слабые стороны

Потоки и блокировки не имеют прямого отношения к параллелизму (напомню, что конкуренция и параллелизм являются связанными, но разными понятиями – см. раздел «Конкуренция или параллелизм?» в главе 1). Как было показано на примере программы подсчета слов, потоки и блокировки можно использовать для распараллеливания последовательного алгоритма, но при этом придется использовать примитивы поддержки конкуренции, что ведет к появлению недетерминизма.

Помимо нескольких экспериментальных систем с распределенной памятью, потоки и блокировки поддерживаются только в архитектурах с разделяемой памятью. Если вам необходимо обеспечить поддержку распределенной памяти (например, чтобы поддержать географическую разбросанность или добиться высокой надежности), вам придется поискать другой инструмент. Это также означает, что потоки и блокировки не могут использоваться для решения задач, слишком больших, чтобы уместиться в рамках одной системы.

Но самый большой недостаток модели потоков и блокировок заключается в *большой сложности* этой модели. Для создателей языков не составляет большого труда добавить поддержку потоков и блокировок, но они оставляют нас, бедных программистов, практически без помощи.

Слон в маленькой комнате

Как мне кажется, многопоточное программирование считается сложным не потому, что так сложно писать многопоточный код, а потому, что его сложно *тестировать*. Ошибки в многопоточной программе могут даже не обнаруживаться.

Возьмем для примера модель памяти. Как мы видели в разделе «Видимость памяти», если потоки будут обращаться к совместно используемой памяти, не синхронизируя своих действий, может про-

изойти все, что угодно. Но как определить, что что-то пошло не так? Как написать тест, который поможет убедиться, что ни один из потоков не пытается обратиться к памяти, не выполнив процедуру синхронизации?

К сожалению, это невозможно. Вы, конечно, можете написать нагрузочный тест для своего кода, но прохождение этого теста не будет означать, что код работает безупречно. Мы знаем, например, что решение задачи о пяти философх (`Philosopher.java`) подвержено проблеме взаимоблокировки, но у меня этот код работал больше недели, никак не проявляя ее.

Тот факт, что ошибки в многопоточных программах могут проявлять себя очень редко, является самой большой проблемой – не раз меня поднимали по ночам, чтобы сообщить, что сервер «завис» без видимых на то причин. Если бы подобные ошибки случались раз в десять минут, я быстро нашел бы причину, но если сервер может работать месяцами, прежде чем ошибка повторится вновь, такую проблему отладить почти невозможно.

Хуже этого является только возможность допустить ошибку в многопоточной программе, которая никак не проявляется, независимо от того, как долго и насколько полно мы будем ее тестировать. Чувствительность кода к изменению порядка доступа к памяти еще не означает, что это как-то проявится. Мы вообще не будем подозревать о существовании проблемы, пока не обновим JVM или не перейдем на использование другой аппаратуры, и тогда мы внезапно столкнемся с таинственной и никому непонятной ошибкой.

Сопровождение

Все описанные проблемы достаточно неприятны для того, кто пишет код, но код редко остается в своем первоизданном виде. Одно дело – убедиться, что все операции с памятью правильно синхронизируются, все блокировки приобретаются в правильном порядке и под их защитой не вызываются никакие сторонние функции. И совсем другое – гарантировать, что все будет работать спустя год после того, как к программе приложат руку десять разных программистов. В прошлом десятилетии мы узнали, как с помощью автоматизированного тестирования надежно проверить результаты рефакторинга кода, но из-за невозможности написать надежный тест для поиска ошибок в многопоточной программе мы не можем гарантировать безопасность даже самой простой процедуры рефакторинга многопоточного кода.

Как результат единственным выходом из ситуации является крайне не внимательное отношение к изменениям, которые вносятся в многопоточный код. И затем еще более внимательная проверка внесенных изменений. Разумеется, это совсем не просто.

Поиск причин методом наработки на отказ

Диагностика проблем в многопоточных приложениях, на мой взгляд, напоминает, как инженеры в Формуле 1 диагностируют отказ двигателя. Двигатель работает безупречно в течение многих часов и вдруг безо всяких видимых причин эффектно взрывается, выбрасывая густые клубы дыма и осколки картера.

Когда автомобиль возвращают в боксы, бедные инженеры видят перед собой груды металла и как-то должны определить, что послужило причиной аварии. Причина может быть самой незначительной – разрушение подшипника масляного насоса или поломка клапана, но как это выяснить, имея лишь мешанину из обломков?

Обычно они включают сбор всех данных телеметрии, каких только возможно, и отправляют пилота на трек в болиде с новым двигателем, надеясь, что в следующий раз, когда произойдет нечто подобное, данные будут содержать полезную информацию.

Другие языки

Желающим более глубоко разобраться в вопросах программирования с применением потоков и блокировок я могу порекомендовать книгу «Java Concurrency in Practice» [Goe06], которая была написана авторами пакета `java.util.concurrent`. В разных языках имеются свои особенности программирования потоков, но общие принципы, рассматривавшиеся в этой главе, остаются неизменными. Правила, такие как синхронизация доступа к разделяемым переменным, приобретение блокировок в фиксированном порядке и отказ от вызова сторонних методов при удерживаемой блокировке, действуют в любых языках программирования.

В частности, хотя в этой главе мы говорили о модели памяти в Java, переупорядочение операций доступа к памяти в конкурентном коде не является чем-то, свойственным только для Java. Разница лишь в том, что большинство языков не имеют четко определенной модели памяти, которая ограничивала бы, как и когда происходит такое переупорядочение. Язык Java стал первым широко известным языком, получившим четко определенную модель памяти. Языки C и C++ лишь

недавно обрели модель памяти, определение которой было включено в стандарты C11 и C++ 11.

Напоследок

При всех сложностях многопоточное программирование еще долго будет оставаться на сцене. Однако в оставшейся части книги мы рассмотрим другие альтернативы, о которых надлежит помнить.

В следующей главе мы рассмотрим приемы функционального программирования, помогающие избежать многих проблем, связанных с потоками и блокировками, за счет отказа от использования изменчивого состояния. Даже если вам никогда не придется писать функциональный код, понимание принципов функционального программирования пригодится вам так или иначе — как вы увидите сами, они лежат в основе многих других моделей конкуренции.



ГЛАВА 3.

Функциональное программирование

Функциональное программирование напоминает автомобиль с водородным двигателем – совершенный, футуристический и пока не получивший широкого распространения, но станет чем-то обыденным лет так через двадцать.

В отличие от императивных программ, которые состоят из последовательностей инструкций, изменяющих глобальное состояние программ во время выполнения, *функциональные программы* производят вычисления, напоминающие вычисление выражений. Эти выражения состоят из истинных, математических функций, которые одновременно являются функциями первого порядка (first-class functions) – то есть ими можно манипулировать как самыми обычными значениями – и не имеют побочных эффектов. Последнее особенно важно для конкурентного программирования, потому что отсутствие побочных эффектов существенно упрощает обеспечение безопасности в многопоточном окружении. Это также первая модель, позволяющая представлять параллелизм непосредственно.

Если какие-то действия вредят вам, перестаньте выполнять их

Правила блокирования, с которыми мы познакомились в главе 2 «Потоки выполнения и блокировки», следует соблюдать только при работе с данными, которые совместно используются разными потоками выполнения и могут изменяться ими. Иными словами: при работе с *разделяемым и изменяемым состоянием*. К данным, которые не изменяются (являются *неизменяемыми*), можно обращаться из разных потоков без применения блокировок.

Именно это обстоятельство делает функциональное программирование особенно привлекательным, когда речь заходит о конкуренции и параллелизме, – функциональные программы лишены изменяемого состояния, поэтому они не страдают ни от одной из проблем, свойственных разделяемому и изменяемому состоянию.

В этой главе мы будем рассматривать функциональное программирование на языке Clojure,¹ диалекте языка Lisp, который выполняется под управлением виртуальной машины JVM. Clojure считается динамически типизированным языком; и если вам приходилось писать программы на Ruby или Python, вы будете чувствовать себя как дома после знакомства с новым для вас синтаксисом. Clojure является гибридным функциональным языком, но в этой главе мы сосредоточимся исключительно на его функциональном подмножестве. Я немного расскажу об этом языке, но если вы пожелаете узнать больше, рекомендую обратиться к книге Стюарта Хэллоуэя (Stuart Halloway) и Аарона Бедра (Aaron Bedra) «Programming Clojure» [HB12].

В первый день мы познакомимся с основами функционального программирования и посмотрим, насколько просто поддаются распараллеливанию функциональные алгоритмы. Во второй день мы погрузимся в изучение библиотеки редукторов (reducers) в Clojure и посмотрим, как фактически осуществляется распараллеливание. Наконец, на третий день мы перейдем от параллелизма к конкуренции и создадим конкурентное функциональное веб-приложение с применением механизмов `future` и `promise`.

День 1: программирование без изменяемого состояния

Когда программисты впервые сталкиваются с функциональным программированием, часто первой их реакцией является недоверие – им кажется невероятной возможность создания нетривиальных программ, которые не изменяют значения переменных. Далее мы увидим, что это не только возможно, но зачастую писать такие программы намного проще.

Опасности изменяемого состояния

Сегодня мы сосредоточимся на приемах параллельного программирования. Мы напишем простую функциональную программу и за-

¹ <http://clojure.org>

тем посмотрим, как ее функциональная природа делает практически тривиальным ее распараллеливание.

Но сначала рассмотрим пару примеров на Java, чтобы показать, насколько полезным может быть отсутствие изменяемого состояния.

Скрытое изменяемое состояние

Ниже демонстрируется класс, не имеющий изменяемого состояния, и потому он должен быть совершенно безопасным при использовании в многопоточном окружении:

```
FunctionalProgramming/DateFormatBug/src/main/java/com/paulbutcher/  
DateParser.java2
```

```
class DateParser {  
    private final DateFormat format = new SimpleDateFormat("yyyy-MM-dd");  
  
    public Date parse(String s) throws ParseException {  
        return format.parse(s);  
    }  
}
```

Когда я попытался запустить простенькую программку, использующую этот класс в нескольких потоках (исходный код программы можно найти в загружаемом пакете примеров для этой книги), она завершилась с сообщением об ошибке:

```
Caught: java.lang.NumberFormatException: For input string: ".12012E4.12012E4"  
Expected: Sun Jan 01 00:00:00 GMT 2012, got: Wed Apr 15 00:00:00 BST 2015
```

Запустив ее во второй раз, я получил уже другое сообщение об ошибке:

```
Caught: java.lang.ArrayIndexOutOfBoundsException: -1
```

И в третий раз программа завершилась аварийно:

```
Caught: java.lang.NumberFormatException: multiple points  
Caught: java.lang.NumberFormatException: multiple points
```

Очевидно, что класс оказался в принципе несовместим с многопоточным окружением, но почему? Он имеет единственную

² <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/DateFormatBug/src/main/java/com/paulbutcher/DateParser.java>

переменную-член, которая является неизменяемой, так как объявлена со спецификатором `final`.

Проблема в том, что класс `SimpleDateFormat` имеет изменяемое состояние, скрытое глубоко в его недрах. Вы можете утверждать, что это связано с ошибкой в реализации класса³, но для нашего обсуждения это не имеет никакого значения. Проблема в том, что такие языки программирования, как Java, способствуют созданию кода со скрытым изменяемым состоянием, как в данном примере, и делают практически невозможным определить, когда такое изменяемое состояние вступает в игру — из API класса `SimpleDateFormat` нельзя сделать вывод о его опасности или безопасности в многопоточном окружении.

Скрытое изменяемое состояние — не единственный источник беспокойства, в чем вы убедитесь далее.

Ускользающее изменяемое состояние

Представьте, что вы занимаетесь созданием веб-службы, которая управляет расписанием спортивных соревнований. Помимо всего прочего, она должна предоставлять возможность изменять составы команд, которую можно было бы реализовать, как показано ниже:

```
public class Tournament {
    private List<Player> players = new LinkedList<Player>();

    public synchronized void addPlayer(Player p) {
        players.add(p);
    }

    public synchronized Iterator<Player> getPlayerIterator() {
        return players.iterator();
    }
}
```

На первый взгляд, код выглядит вполне безобидно — список `players` объявлен приватным и доступен только посредством методов `addPlayer()` и `getPlayerIterator()`, каждый из которых объявлен синхронизированным. К сожалению, в многопоточном окружении эта реализация небезопасна, потому что итератор, возвращаемый методом `getPlayerIterator()`, ссылается на изменяемое состояние внутри `players` — если другой поток вызовет `addPlayer()` в то время, как итератор еще используется, он вызовет исключение

³ http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4228335

`ConcurrentModificationException` или что похуже. Изменяемое состояние ускользнуло из-под защиты класса `Tournament`.

Скрытое и ускользающее состояние – это лишь две опасности изменяемого состояния для конкурентных программ из великого их множества. Эти опасности исчезнут, если нам удастся найти способ полностью избавиться от изменяемого состояния. Именно такой способ предоставляет нам функциональное программирование.

Краткий экскурс в язык Clojure

Чтобы освоить Lisp-подобный синтаксис языка Clojure, достаточно всего нескольких минут.

Проще всего экспериментировать с Clojure в его интерактивной оболочке REPL (`read-evaluate-printloop` – цикл чтения–вычисления–вывода), которую можно запустить командой `lein repl` (`lein` – это стандартный инструмент сборки для Clojure). В этой оболочке можно вводить программный код и сразу же получать результат его выполнения, минуя этапы создания файла с исходным кодом и его компиляции, что очень удобно при исследовании незнакомого кода. После запуска оболочки REPL появится следующее приглашение к вводу:

```
user=>
```

Любой код на языке Clojure, который вы введете в этой строке, будет выполнен немедленно.

Код на языке Clojure практически целиком состоит из списков в круглых скобках, которые называют *s-выражениями*. Вызов функции, который в большинстве языков можно записать как `max(3, 5)`, на языке Clojure выглядит так:

```
user=> (max 3 5)
5
```

То же относится и к арифметическим операторам. Ниже приводится пример вычисления выражения $1 + 2 * 3$:

```
user=> (+ 1 (* 2 3))
7
```

Определение констант выполняется с помощью формы `def`:


```
user=> (def meaning-of-life 42)
#'user/meaning-of-life
user=> meaning-of-life
42
```

Даже управляющие конструкции являются s-выражениями:

```
user=> (if (< meaning-of-life 0) "negative" "non-negative")
"non-negative"
```

Несмотря на то что в Clojure почти все является s-выражениями, существует несколько исключений. Литералы векторов (массивов) заключаются в квадратные скобки:

```
user=> (def droids ["Huey" "Dewey" "Louie"])
#'user/droids
user=> (count droids)
3

user=> (droids 0)
"Huey"
user=> (droids 2)
"Louie"
```

А литералы ассоциативных массивов заключаются в фигурные скобки:

```
user=> (def me {:name "Paul" :age 45 :sex :male})
#'user/me
user=> (:age me)
45
```

Ключи в ассоциативных массивах в подавляющем большинстве случаев являются *ключевыми словами* (keywords), которые начинаются с двоеточия и очень похожи на символы в Ruby или интернированные строки в Java.

Наконец, определение функции осуществляется с помощью формы `defn`, с аргументами, указанными в виде вектора:

```
user=> (defn percentage [x p] (* x (/ p 100.0)))
#'user/percentage
user=> (percentage 200 10)
20.0
```

На этом мы завершаем экскурс в язык Clojure. Я показал вам некоторые аспекты языка, которые будут встречаться нам наиболее часто.

Первая функциональная программа

Я уже говорил, что самой интересной особенностью функционального программирования является отсутствие изменяемого состояния, но пока не показал на практике, как такое может быть. Позвольте мне восполнить этот недостаток.

Представьте, что нам требуется найти сумму последовательности чисел. В императивных языках, таких как Java, мы могли реализовать решение этой задачи примерно так:

```
public int sum(int[] numbers) {  
    int accumulator = 0;  
    for (int n: numbers)  
        accumulator += n;  
    return accumulator;  
}
```

Такое решение далеко не функционально из-за наличия изменяемой переменной `accumulator`: ее значение изменяется в каждой итерации цикла `for`. Решение на языке Clojure, представленное ниже, напротив, является функциональным, так как не имеет изменяемого состояния:

FunctionalProgramming/Sum/src/sum/core.clj⁴

```
(defn recursive-sum [numbers]  
  (if (empty? numbers)  
      0  
      (+ (first numbers) (recursive-sum (rest numbers)))))
```

Это *рекурсивное* решение – функция `recursive-sum` вызывает саму себя. Если список `numbers` пуст, она просто вернет 0. В противном случае она вернет результат сложения первого элемента списка `numbers` с суммой чисел в оставшейся части списка.

Рекурсивное решение дает верный результат, но мы можем найти более удачное решение. Ниже представлено решение, которое не только проще, но и эффективнее:

FunctionalProgramming/Sum/src/sum/core.clj⁵

```
(defn reduce-sum [numbers]  
  (reduce (fn [acc x] (+ acc x)) 0 numbers))
```

⁴ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Sum/src/sum/core.clj>

⁵ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Sum/src/sum/core.clj>

Здесь используется стандартная функция Clojure `reduce`, которая принимает три аргумента – функцию, начальное значение и коллекцию.

Тут мы передаем функции `reduce` анонимную функцию, объявленную с помощью формы `fn`, которая принимает два аргумента и возвращает их сумму. Она будет вызвана функцией `reduce` для каждого элемента в коллекции – в первый вызов будут переданы начальное значение (0 в данном случае) и первый элемент коллекции; во второй вызов будут переданы результат первого вызова и второй элемент коллекции; и так далее.

Но это не все. Мы можем еще улучшить этот код: обратите внимание, что оператор `+` сам по себе является функцией, принимающей два аргумента и возвращающей их сумму. Мы можем передать ее непосредственно функции `reduce`, не создавая анонимную функцию:

```
FunctionalProgramming/Sum/src/sum/core.clj6
```

```
(defn sum [numbers]
  (reduce + numbers))
```

Итак, у нас получилось решение, которое проще и короче императивного. Вы увидите, что это часто случается при преобразовании императивного кода в функциональный.

Параллелизм без усилий

Итак, мы теперь знаем, как примерно выглядит функциональный код, и настала пора посмотреть, как выглядит параллельный функциональный код. Давайте попробуем распараллелить нашу функцию `sum`. Как оказывается, это очень просто:

```
FunctionalProgramming/Sum/src/sum/core.clj7
```

```
(ns sum.core
  (:require [clojure.core.reducers :as r]))

(defn parallel-sum [numbers]
  (r/fold + numbers))
```

⁶ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Sum/src/sum/core.clj>

⁷ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Sum/src/sum/core.clj>



Вопрос Джо: что случится, если функции `reduce` передать пустую коллекцию?

Наша последняя версия функции `sum` не передает начальное значение функции `reduce`:

```
(reduce + numbers)
```

Поэтому у кого-то может возникнуть вопрос: что получится, если передать ей пустую коллекцию? Функция сделает все правильно и вернет ноль:

```
sum.core=> (sum [])
0
```

Но откуда функция `reduce` знает, что нужно вернуть именно ноль, а не, скажем, `nil` или `1`? Дело в том, что она опирается на одну интересную особенность многих операторов в Clojure – они знают свои значения тождественности. Функция `+`, например, может принять любое число аргументов, включая и нулевое:

```
user=> (+ 1 2)
3
user=> (+ 1 2 3 4)
10
user=> (+ 42)
42
user=> (+)
0
```

При вызове без аргументов она вернет аддитивное тождество, `0`. Аналогично функция `*` знает, что мультипликативное тождество равно `1`:

```
user=> (*)
1
```

Если функцию `reduce` вызвать без начального значения, в качестве его она будет использовать результат вызова указанной функции без аргументов.

Кстати говоря, благодаря тому что функция `+` может принимать произвольное число аргументов, мы можем реализовать свою функцию `sum` с помощью функции `apply`, которая принимает функцию вместе с вектором и вызывает ее, передавая указанный вектор в качестве списка аргументов:

```
FunctionalProgramming/Sum/src/sum/core.clj *
```

```
(defn apply-sum [numbers]
  (apply + numbers))
```

Но, в отличие от версии на основе `reduce`, эта версия труднее поддается распараллеливанию.

* <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Sum/src/sum/core.clj>

Единственное отличие этой версии – в том, что она использует функцию `fold` из пакета `clojure.core.reducers` (для которого мы определили псевдоним `r`, чтобы не заставлять себя вводить длинное имя пакета) вместо `reduce`.

Ниже представлен листинг сеанса работы в интерактивной оболочке REPL, позволяющий оценить производительность параллельной версии:

```
sum.core=> (def numbers (into [] (range 0 10000000)))
#'sum.core/numbers
sum.core=> (time (sum numbers))
"Elapsed time: 1099.154 msecs"
49999995000000
sum.core=> (time (sum numbers))
"Elapsed time: 125.349 msecs"
49999995000000
sum.core=> (time (parallel-sum numbers))
"Elapsed time: 236.609 msecs"
49999995000000
sum.core=> (time (parallel-sum numbers))
"Elapsed time: 49.835 msecs"
49999995000000
```

Здесь сначала создается вектор, содержащий все целые числа в диапазоне от 0 до 10 000 000, который создается вставкой результата `(range 0 10000000)` в пустой вектор `[]` с помощью `into`. Затем с помощью макроса `time` выводится время выполнения указанного кода. Как это часто бывает с кодом в JVM, нужно запустить его несколько раз, чтобы дать возможность динамическому компилятору (`just-in-time compiler`) скомпилировать этот код и получить более достоверное время.

Итак, на моем четырехъядерном Mac функция `fold` уменьшила время выполнения со 125 мсек до 50 мсек, то есть дала 2.5-кратный прирост скорости. Устройство функции `fold` мы рассмотрим завтра, но прежде давайте посмотрим, как выглядит функциональная версия нашего примера подсчета слов в Википедии.

Функциональный подсчет слов

Сегодня мы создадим последовательную реализацию счетчика слов, а параллельную версию – завтра. Нам потребуются три компонента:

- функция, которая будет принимать дампы Википедии в формате XML и возвращать последовательность страниц;

- функция, которая будет принимать страницу и возвращать последовательность слов в этой странице;
- функция, которая будет принимать последовательность слов и возвращать ассоциативный массив с их частотами.

Мы не будем обсуждать реализацию первых двух функций – эта книга посвящена конкуренции, а не обработке строк или данных в формате XML (если вас интересуют тонкости, обращайтесь к исходному коду в загружаемом пакете примеров). Но мы подробно рассмотрим, как осуществляется подсчет слов, поскольку именно эту операцию нам предстоит распараллелить.

Функциональные ассоциативные массивы

Так как нам требуется вернуть ассоциативный массив с частотами слов, нужно разобраться с парой функций для работы с такими массивами `get` и `assoc`:

```
user=> (def counts {"apple" 2 "orange" 1})
#'user/counts
user=> (get counts "apple" 0)
2
user=> (get counts "banana" 0)
0
user=> (assoc counts "banana" 1)
{"banana" 1, "orange" 1, "apple" 2}
user=> (assoc counts "apple" 3)
{"orange" 1, "apple" 3}
```

Итак, функция `get` просто отыскивает ключ в ассоциативном массиве и возвращает его значение или значение по умолчанию, если искомым ключ не найден. А функция `assoc` принимает ассоциативный массив вместе с ключом и значением и возвращает новый ассоциативный массив, с добавленными в него ключом и значением.

Частоты

Теперь нам должно хватить знаний, чтобы написать функцию, которая будет принимать последовательность слов и возвращать ассоциативный массив, в котором каждое слово (в виде ключа) будет связано с числом его вхождений:

FunctionalProgramming/WordCount/src/wordcount/word_frequencies.clj⁸

```
(defn word-frequencies [words]
  (reduce
```

⁸ http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/WordCount/src/wordcount/word_frequencies.clj


```
(fn [counts word] (assoc counts word (inc (get counts word 0))))
({} words))
```

На сей раз в качестве начального значения мы передаем функции `reduce` пустой ассоциативный массив `{}`. А затем для каждого слова в `words` мы увеличиваем счетчик текущего слова. Ниже показан пример использования этой функции:

```
user=> (word-frequencies ["one" "potato" "two" "potato" "three" "potato"
"four"])
{"four" 1, "three" 1, "two" 1, "potato" 3, "one" 1}
```

Как оказывается, стандартная библиотека Clojure предвосхитила наши желания и — в ней имеется стандартная функция с именем `frequencies`, которая принимает любые коллекции и возвращает ассоциативный массив с частотами встречаемости их элементов:

```
user=> (frequencies ["one" "potato" "two" "potato" "three" "potato"
"four"])
{"one" 1, "potato" 3, "two" 1, "three" 1, "four" 1}
```

Теперь, получив возможность считать слова, нам осталось лишь связать ее с обработкой XML.

Еще функции для работы с последовательностями

Чтобы увидеть, как это сделать, нужно познакомиться еще с несколькими функциями. Для начала рассмотрим функцию `map`:

```
user=> (map inc [0 1 2 3 4 5])
(1 2 3 4 5 6)
user=> (map (fn [x] (* 2 x)) [0 1 2 3 4 5])
(0 2 4 6 8 10)
```

Получая функцию и последовательность, `map` возвращает новую последовательность, содержащую результаты применения указанной функции к каждому элементу исходной последовательности.

Мы можем немного упростить вторую версию, задействовав функцию `partial`, которая принимает функцию с одним или более ее аргументами и возвращает частично примененную функцию:

```
user=> (def multiply-by-2 (partial * 2))
#'user/multiply-by-2
user=> (multiply-by-2 3)
6
user=> (map (partial * 2) [0 1 2 3 4 5])
(0 2 4 6 8 10)
```

Наконец, представьте, что имеется функция, которая принимает строку с регулярным выражением, с помощью которого разбивает строку на слова и возвращает их в виде последовательности слов:

```
user=> (defn get-words [text] (re-seq #"\\w+" text))
#'user/get-words
user=> (get-words "one two three four")
("one" "two" "three" "four")
```

Как вы уже наверняка поняли, отображение этой функции на последовательность строк даст нам последовательность последовательностей:

```
user=> (map get-words ["one two three" "four five six" "seven eight nine"])
(("one" "two" "three") ("four" "five" "six") ("seven" "eight" "nine"))
```

В случае если потребуется получить единую последовательность, содержащую объединение всех подпоследовательностей, можно воспользоваться функцией `mapcat`:

```
user=> (mapcat get-words ["one two three" "four five six" "seven eight nine"])
("one" "two" "three" "four" "five" "six" "seven" "eight" "nine")
```

Теперь у нас есть все необходимое для создания функции подсчета слов.

Объединяем все вместе

Ниже приводится определение функции `count-words-sequential`. Она принимает последовательность страниц и возвращает ассоциативный массив с частотами вхождений слов в эти страницы:

```
FunctionalProgramming/WordCount/src/wordcount/core.clj9
```

```
(defn count-words-sequential [pages]
  (frequencies (mapcat get-words pages)))
```

Она начинается с преобразования последовательности страниц в последовательность слов, вызовом `(mapcat get-words pages)`. Затем последовательность слов передается функции `frequencies`.

Сравните получившийся результат с императивной версией, что приводится в разделе «Законченная программа» в главе 2. И снова функциональное решение оказалось существенно проще и короче императивного эквивалента.

⁹ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/WordCount/src/wordcount/core.clj>

Лень – это благо

Есть одно обстоятельство, которое не может не вызывать беспокойства, – дампы Википедии занимают порядка 40 Гбайт. Если позволить функции `count-words` сохранить все слова в виде одной огромной последовательности, она почти наверняка исчерпает всю доступную память.

Однако этого не происходит, а причина в том, что последовательности в Clojure являются *ленными* (*lazy*) – элементы ленивой последовательности генерируются, только когда они необходимы. Посмотрим, что это означает на практике.

Функция `range` в языке Clojure производит последовательность чисел:

```
user=> (range 0 10)
(0 1 2 3 4 5 6 7 8 9)
```

Выполняя этот код, интерактивная оболочка REPL *реализует* (то есть полностью вычисляет) всю последовательность и выводит ее.

Ничто не мешает вам реализовать по-настоящему большие последовательности и тем самым превратить свой компьютер в дорогостоящий обогреватель. Попробуйте выполнить следующий код, и вам придется ждать довольно долго, прежде чем вы увидите результат (если, конечно, в вашем компьютере достаточно памяти):

```
user=> (range 0 100000000)
```

Но попробуйте выполнить следующий код, и вы получите результат немедленно:

```
user=> (take 10 (range 0 100000000))
(0 1 2 3 4 5 6 7 8 9)
```

Поскольку функция `take` интересуется только первыми десятью элементами аргумента-последовательности, функция `range` производит лишь эти элементы. Это правило действует на любом уровне вложенности вызовов функций:

```
user=> (take 10 (map (partial * 2) (range 0 100000000)))
(0 2 4 6 8 10 12 14 16 18)
```

В программе можно даже обрабатывать бесконечные последовательности. Функция `iterate`, к примеру, генерирует бесконечную по-

следовательность, повторно применяя указанную функцию к начальному значению, возвращая результат и повторяя все снова:

```
user=> (take 10 (iterate inc 0))  
(0 1 2 3 4 5 6 7 8 9)  
user=> (take 10 (iterate (partial + 2) 0))  
(0 2 4 6 8 10 12 14 16 18)
```

Наконец, ленивые последовательности не только позволяют нам не генерировать элементы в конце последовательности, но и дают возможность пропустить начальные элементы, если они не нужны. Следующий пример может потребовать некоторого времени на выполнение, но он никогда не вызовет исчерпание памяти:

```
user=> (take-last 5 (range 0 100000000))  
(99999995 99999996 99999997 99999998 99999999)
```

Так как последовательность страниц, возвращаемая функцией `get-pages`, является ленивой, функция `count-words` оказывается способной обработать 40-гигабайтный дамп Википедии без каких-либо проблем. И, что самое приятное, она легко поддается распараллеливанию, но об этом мы поговорим завтра.

В завершение первого дня

Итак, первый день закончился. Завтра мы напишем параллельную версию программы подсчета слов и познакомимся с функцией `fold` поближе.

Что мы узнали в первый день

Конкурентное программирование на императивных языках сопряжено с большими сложностями из-за распространенности изменяемого состояния. Функциональное программирование упрощает создание конкурентного программного кода, устраняя изменяемое состояние. Мы видели, как:

- применить функцию ко всем элементам последовательности с помощью `map` или `mapcat`;
- пользоваться ленивыми последовательностями для обработки огромных и даже бесконечных последовательностей;
- свернуть последовательность в единственное (возможно, очень сложное) значение с помощью `reduce`;
- распараллелить операцию свертки с помощью `fold`.

Упражнения к первому дню

Найдите:

- «шпаргалку» по языку Clojure со списком наиболее часто используемых функций;
- описание функции `lazy-seq`, которая позволит вам создавать собственные ленивые последовательности.

Задания:

- в отличие от многих функциональных языков, Clojure не предоставляет оптимизацию хвостовой рекурсии, поэтому рекурсия очень редко используется в идиоматическом коде на Clojure. Перепишите функцию `recursive-sum` так, чтобы она использовала специальные формы `loop` и `recur`;
- перепишите функцию `reduce-sum` так, чтобы она использовала макрос `#()` вместо `(fn ...)`.

День 2: функциональный параллелизм

Сегодня мы продолжим обсуждение функционального программирования и его возможностей распараллеливания задач, поближе познакомившись с функцией `fold`. Но перед этим посмотрим на параллельную версию подсчета слов в Википедии.

По одной странице за раз

В первый день мы узнали, что функция `map` создает новую последовательность, применяя указанную функцию к каждому элементу исходной последовательности. Однако нет ничего, что требовало бы, чтобы такое применение происходило последовательно, — в Clojure имеется функция `pmap`, которая действует точно так же, как `map`, но применяет указанную функцию параллельно. Она действует *частично лениво*, в том смысле что какая-то часть параллельных вычислений выполняется заранее, но при этом она не производит всех результатов, если этого не требуется.

Мы могли бы, например, организовать параллельное преобразование последовательности страниц Википедии в последовательность ассоциативных массивов слов для этих страниц, как показано ниже:

```
(pmap # (frequencies (get-words %)) pages)
```


В данном случае мы определили функцию для передачи `rmap` с помощью макроса `#(...)`, который предоставляет более краткий способ создания анонимных функций. Аргументы определяются как `%1`, `%2` и т. д. Если функция принимает единственный аргумент, можно использовать более короткую форму записи %:

```
#(frequencies (get-words %))
```

Предыдущий код эквивалентен следующему:

```
(fn [page] (frequencies (get-words page)))
```

Ниже показано, как он действует:

```
wordcount.core=> (def pages ["one potato two potato three potato four"
                             #_=> "five potato six potato seven potato more"])
#'wordcount.core/pages
wordcount.core=> (pmap #(frequencies (get-words %)) pages)
({{"one" 1, "potato" 3, "two" 1, "three" 1, "four" 1}
 {"five" 1, "potato" 3, "six" 1, "seven" 1, "more" 1})
```

Далее мы можем получить общие счетчики слов, свернув эту последовательность в единственный ассоциативный массив. Наша функция свертки должна принимать два ассоциативных массива и объединять их так, чтобы:

- ключи в получившемся ассоциативном массиве являлись объединением ключей двух исходных ассоциативных массивов и,
- если ключ имеется только в одном из двух исходных ассоциативных массивов, его значение с ключом просто переписывается в ассоциативный массив результата или,
- если ключ имеется в обоих исходных ассоциативных массивах, в соответствующий ключ ассоциативного массива результата записывается сумма их значений в исходных массивах.

Диаграмма на рис. 7 иллюстрирует, какой результат мы хотим получить.

Мы могли бы сами написать такую функцию свертки, но (как это часто бывает) подобная функция уже имеется в стандартной библиотеке. Вот выдержка из документации с ее описанием:

```
(merge-with f & maps)
```

Возвращает ассоциативный массив, являющийся объединением всех последующих исходных ассоциативных массивов с первым — если один и тот же ключ присутствует более чем в

одном массиве, результат объединения значений этих ключей (слева направо) будет определяться вызовом (`f val-in-result val-in-latter`).

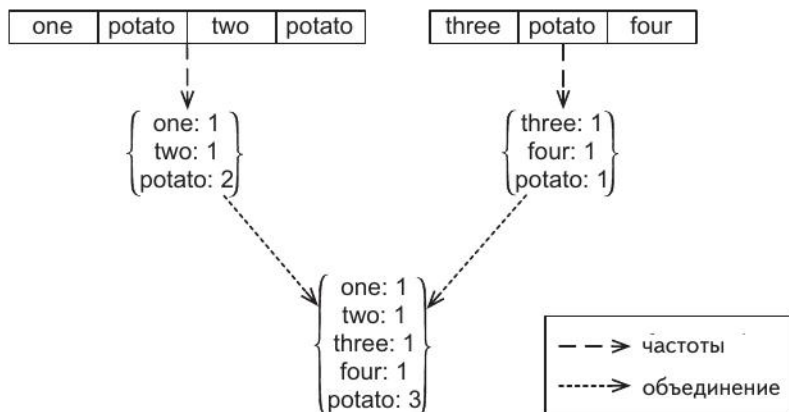


Рис. 7. Объединение двух ассоциативных массивов

Напомним, что функция `partial` возвращает частично примененную функцию, соответственно, (`partial merge-with +`) даст нам функцию, принимающую два ассоциативных массива и объединяющую их с помощью функции `+`, если один и тот же ключ будет обнаружен в обоих массивах:

```

user=> (def merge-counts (partial merge-with +))
#'user/merge-counts
user=> (merge-counts {:x 1 :y 2} {:y 1 :z 1})
{:z 1, :y 3, :x 1}

```

Объединив все, что мы написали, получим параллельную версию функции подсчета слов:

FunctionalProgramming/WordCount/src/wordcount/core.clj¹⁰

```

(defn count-words-parallel [pages]
  (reduce (partial merge-with +)
    (pmap #(frequencies (get-words %)) pages)))

```

Теперь давайте посмотрим, насколько хорошо она работает.

¹⁰ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/WordCount/src/wordcount/core.clj>

Разделение данных на пакеты для увеличения производительности

На моем MacBook Pro последовательная версия выполняется около 140 секунд, подсчитывая слова в первых 100 000 страницах Википедии. Параллельная версия выполняется 94 секунды, то есть в 1.5 раза быстрее. У нас получилось увеличить производительность за счет распараллеливания задачи, но не настолько, как хотелось бы.

Причина здесь та же самая, что мешала нам на прошлой неделе в решении на основе потоков и блокировок. Мы выполняем подсчет и слияние страница за страницей, что ведет к необходимости выполнять большое число операций слияния. Мы можем уменьшить количество слияний, обрабатывая страницы не по одной, а пакетами:

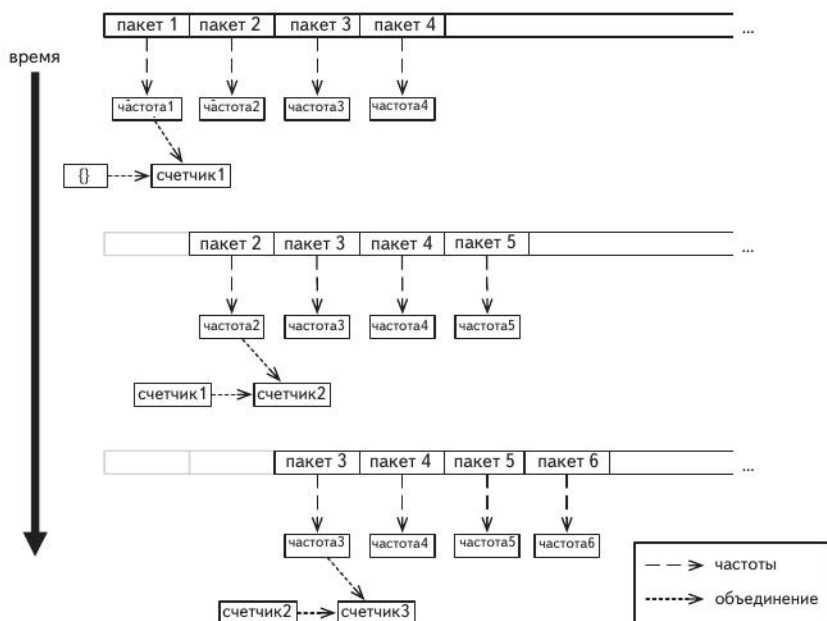


Рис. 8. Разделение данных на пакеты

Ниже приводится реализация `word-count` для обработки пакетов объемом по 100 страниц:

FunctionalProgramming/WordCount/src/wordcount/core.clj¹¹

```
(defn count-words [pages]
  (reduce (partial merge-with +)
    (pmap count-words-sequential (partition-all 100 pages))))
```

С помощью функции `partition-all` она разбивает общую последовательность страниц на пакеты (более мелкие последовательности):

```
user=> (partition-all 4 [1 2 3 4 5 6 7 8 9 10])
((1 2 3 4) (5 6 7 8) (9 10))
```

Затем определяет частоту встречаемости слов в каждом пакете вызовом `word-count-sequential` и, наконец, объединяет результаты, как и прежде. Данная версия справляется с задачей подсчета частоты встречаемости слов в первых 100 000 страницах Википедии за 44 секунды, то есть в 3.2 раза быстрее последовательной версии.

Редуценты (reducers)

В первый день мы видели, что использование функции `fold` вместо `reduce` может дать весьма неплохой прирост производительности. Чтобы разобраться, как функции `fold` удастся достичь таких результатов, необходимо познакомиться с библиотекой редуцентов (`reducers`) в языке Clojure.

Редуценты (`reducers`) позволяют описать, как получить желаемую коллекцию. Обычная версия `map` принимает функцию и последовательность (возможно, ленивую) и возвращает другую последовательность (также, возможно, ленивую):

```
user=> (map (partial * 2) [1 2 3 4])
(2 4 6 8)
```

Версия из `clojure.core.reducers`, вызванная с теми же аргументами, напротив, вернет *редуцируемое значение* (`reducible`):

```
user=> (require '[clojure.core.reducers :as r])
nil
user=> (r/map (partial * 2) [1 2 3 4])
#<reducers$folder$reify__1599 clojure.core.reducers$folder$reify__1599@151964cd>
```

¹¹ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/WordCount/src/wordcount/core.clj>

Редуцируемое значение нельзя использовать непосредственно — оно предназначено для последующей передачи функции `reduce`:

```
user=> (reduce conj [] (r/map (partial * 2) [1 2 3 4]))  
[2 4 6 8]
```

Анонимная функция, переданная функции `reduce` в предыдущем фрагменте, принимает коллекцию в первом аргументе (первоначально — пустой вектор, `[]`) и обращается к функции `conj`, чтобы добавить в него второй аргумент. Результатом является коллекция, обработанная функцией `map`.

Ниже приводится эквивалентная версия, которая действует точно так же потому, что `into` использует функцию `reduce`:

```
user=> (into [] (r/map (partial * 2) [1 2 3 4]))  
[2 4 6 8]
```

Помимо версий `map` и `mapcat`, в библиотеке `reducer` имеются также версии других функций обработки последовательностей из модуля `clojure.core`. И, так же как их эквиваленты из `clojure.core`, они могут объединяться в цепочки вызовов:

```
user=> (into [] (r/map (partial + 1) (r/filter even? [1 2 3 4])))  
[3 5]
```

Редукент вместо самого результата возвращает рецепт получения результата — рецепт, который может быть выполнен только функцией `reduce` или `fold`. Такой порядок имеет два основных преимущества:

- более высокая эффективность в сравнении с цепочками вызовов функций, возвращающих ленивые последовательности, за счет отсутствия необходимости создавать промежуточные последовательности;
- возможность с помощью `fold` распараллелить всю цепочку операций с исходной коллекцией.

Внутреннее устройство редукентов

Чтобы понять, как действуют редукенты, мы реализуем свою собственную функцию-редукент — немного упрощенную, но все еще весьма эффективную версию библиотеки `clojure.core.reducers`. Но для этого нам сначала нужно познакомиться с *протоколами* в языке Clojure. Протоколы близко напоминают интерфейсы в Java — это кол-

лекции методов, определяющие некоторые абстракции. Коллекции в языке Clojure поддерживаются функцией `reduce` посредством протокола `CollReduce`:

```
(defprotocol CollReduce
  (coll-reduce [coll f] [coll f init]))
```

Этот протокол определяет единственную функцию с именем `coll-reduce`, имеющую несколько *арностей* (arities), — она может принимать два (`coll` и `f`) или три (`coll`, `f` и `init`) аргумента. Первый аргумент играет ту же роль, что и ссылка `this` в языке Java, обеспечивая полиморфизм. Взгляните на следующий фрагмент:

```
(coll-reduce coll f)
```

Эквивалентный код на языке Java можно записать так:

```
coll.collReduce(f);
```

Функция `reduce` просто вызывает `coll-reduce`, делегируя задачу свертки самой коллекции. Убедиться в этом можно, реализовав собственную версию `reduce`:

```
FunctionalProgramming/Reducers/src/reducers/core.clj12
```

```
(defn my-reduce
  ([f coll] (coll-reduce coll f))
  ([f init coll] (coll-reduce coll f init)))
```

Здесь демонстрируется особенность формы `defn`, о которой не рассказывалось выше и благодаря которой можно определять функции, принимающие разное число аргументов (в данном случае два или три). В обоих случаях она просто передает свои аргументы функции `coll-reduce`. А теперь проверим, как она работает:

```
reducers.core=> (my-reduce + [1 2 3 4])
10
reducers.core=> (my-reduce + 10 [1 2 3 4])
20
```

Далее посмотрим, как можно реализовать собственную версию `map`:

¹² <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Reducers/src/reducers/core.clj>

FunctionalProgramming/Reducers/src/reducers/core.clj¹³

```
(defn make-reducer [reducible transformf]
  (reify
    CollReduce
    (coll-reduce [_ fl]
      (coll-reduce reducible (transformf fl) (fl)))
    (coll-reduce [_ fl init]
      (coll-reduce reducible (transformf fl) init))))

(defn my-map [mapf reducible]
  (make-reducer reducible
    (fn [reducef]
      (fn [acc v]
        (reducef acc (mapf v)))))))
```

Мы использовали функцию `make-reducer`, принимающую редуцированное значение, и функцию преобразования, возвращающую реализацию протокола `CollReduce` (определяется с помощью формы `reify`). Форма `reify`, реализующая протокол, напоминает ключевое слово `new` в Java — она возвращает анонимный экземпляр интерфейса.

Данный экземпляр протокола `CollReduce` вызывает метод `coll-reduce` редуцированного значения `reducible`, используя функцию `transformf` для преобразования аргумента `fl`.



Вопрос Джо: что означает символ подчеркивания?

В языке Clojure часто используется символ подчеркивания («_») в качестве имени неиспользуемого параметра. Мы могли бы записать пример, представленный выше, немного иначе:

```
(coll-reduce [this fl]
  (coll-reduce reducible (transformf fl) (fl)))
```

Но использование символа подчеркивания делает более очевидным тот факт, что ссылка `this` не используется в функции.

Функция преобразования `transformf`, что передается функции `make-reducer`, — это функция, принимающая другую функцию и возвращающая преобразованную версию этой функции. Ниже приводится функция преобразования, определяемая внутри `my-map`:

¹³ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Reducers/src/reducers/core.clj>

```
(fn [reducef]
  (fn [acc v]
    (reducef acc (mapf v)))))
```

Напомним, что функция преобразования вызывается по одному разу для каждого элемента коллекции. В первом аргументе ей передается аккумулятор (`acc`), а во втором — значение элемента коллекции (`v`). Таким образом, данная функция преобразования `reducef` возвращает функцию, которая вызывает `reducef` со вторым аргументом, измененным с помощью `mapf` (функции, переданной функции `my-map`). Посмотрим, как она работает:

```
reducers.core=> (into [] (my-map (partial * 2) [1 2 3 4]))
[2 4 6 8]
reducers.core=> (into [] (my-map (partial + 1) [1 2 3 4]))
[2 3 4 5]
```

Как вы уже наверняка поняли, мы можем также объединять вызовы `my-map` в цепочки:

```
reducers.core=> (into [] (my-map (partial * 2) (my-map (partial + 1)
[1 2 3 4]))))
[4 6 8 10]
```

При внимательном изучении можно заметить, что этот код выполняет единственное преобразование с помощью функции, полученной путем композиции (`partial * 2`) с (`partial + 1`).

Теперь мы знаем, как редукенты поддерживают `reduce`. Далее рассмотрим, как функция `fold` распараллеливает эти преобразования.

Разделяй и властвуй

Вместо последовательной свертки коллекции `fold` использует метод *половинного деления*. Сначала она делит коллекцию на две части, затем делит пополам эти части и т. д., пока коллекция не будет поделена на фрагменты, меньшие некоторого предела (по умолчанию 512). Затем к каждому фрагменту применяется операция свертки, выполняющаяся последовательно, а результаты объединяются попарно, пока не будет получен общий результат. В результате получается своеобразное двоичное дерево, как показано на рис. 9.

Операции свертки и объединения выполняются параллельно, потому что `fold` создает соответствующее дерево параллельно выполняющихся задач (используя инфраструктуру Fork/Join в Java 7). Задачи, являющиеся листьями дерева, выполняют операции свертки. За-

дачи на следующих уровнях ждут результатов этих операций и, когда они появятся, объединяют их. Процесс объединения продолжается, пока не получится единственный результат в корне дерева.

Если `fold` передать единственную функцию (+ в нашем примере), она будет использоваться и для свертки, и для объединения. Однако, как вы увидите ниже, иногда имеет смысл использовать разные функции для свертки и объединения.

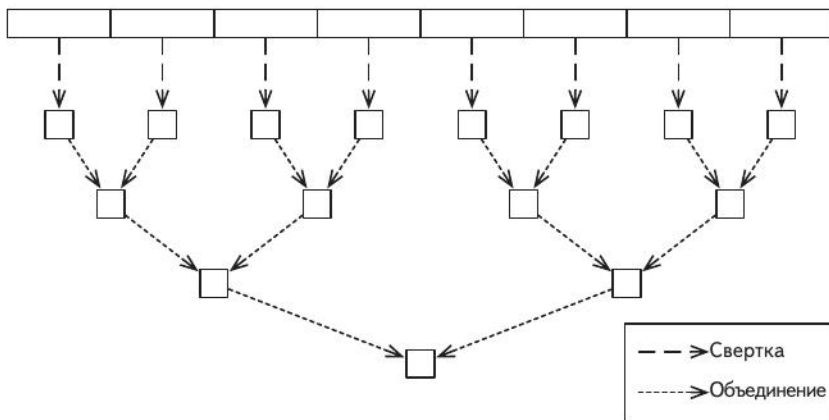


Рис. 9. Дерево свертки

Поддержка функции `fold`

Коллекции, поддающиеся свертке, помимо протокола `CollReduce`, поддерживают также протокол `CollFold`:

```
(defprotocol CollFold
  (coll-fold [coll n combinef reducef]))
```

Так же как `reduce` вызывает метод `coll-reduce`, `fold` вызывает `coll-fold`:

FunctionalProgramming/Reducers/src/reducers/core.clj¹⁴

```
(defn my-fold
  ([reducef coll]
   (my-fold reducef reducef coll))
  ([combinef reducef coll]
   (my-fold 512 combinef reducef coll)))
```

¹⁴ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Reducers/src/reducers/core.clj>

```
([n combinef reducef coll]
 (coll-fold coll n combinef reducef)))
```

Версии с двумя и с тремя аргументами просто вызывают `my-fold` рекурсивно, используя значения по умолчанию `combinef` и `n`, если они не указаны. Версия с четырьмя аргументами вызывает реализацию `coll-fold` в коллекции.

Единственное изменение, которое требуется сделать, чтобы обеспечить поддержку параллельного выполнения операций в `fold`, — включить в `make-reducer` реализацию протокола `CollFold` в дополнение к реализации `CollReduce`:

FunctionalProgramming/Reducers/src/reducers/core.clj¹⁵

```
(defn make-reducer [reducible transformf]
  (reify
    > CollFold
    > (coll-fold [_ n combinef reducef]
    > (coll-fold reducible n combinef (transformf reducef)))
    CollReduce
    (coll-reduce [_ fl]
      (coll-reduce reducible (transformf fl) (fl)))
    (coll-reduce [_ fl init]
      (coll-reduce reducible (transformf fl) init))))
```

Реализация `CollFold` очень похожа на реализацию `CollReduce` — она преобразует функцию свертки и передает остальные аргументы функции `coll-fold`. Посмотрим, как она работает:

```
reducers.core=> (def v (into [] (range 10000)))
#'reducers.core/v
reducers.core=> (my-fold + v)
49995000
reducers.core=> (my-fold + (my-map (partial * 2) v))
99990000
```

Далее мы рассмотрим пример передачи разных функций свертки и объединения функции `fold`.

Подсчет слов с помощью `fold`

Наша старая, добрая функция `frequencies` являются отличным примером, когда необходимо задействовать разные функции свертки и объединения при использовании `fold`:

¹⁵ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Reducers/src/reducers/core.clj>

FunctionalProgramming/Reducers/src/reducers/parallel_frequencies.clj¹⁶

```
(defn parallel-frequencies [coll]
  (r/fold
    (partial merge-with +)
    (fn [counts x] (assoc counts x (inc (get counts x 0))))
    coll))
```

Эта версия напоминает пакетную, параллельную реализацию word-count, которую мы видели выше, — пакеты свертываются в ассоциативные массивы, которые затем объединяются с помощью (partial merge-with +).

Мы не можем испытать эту версию с нашими страницами из Википедии, потому что fold не работает с ленивыми последовательностями (из-за отсутствия возможности выполнить деление ленивой последовательности пополам). Но мы можем проверить ее, например, на большом векторе случайных чисел.

Функция repeatedly создает бесконечную ленивую последовательность, вновь и вновь вызывая функцию, переданную ей в виде аргумента. В данном случае мы использовали функцию rand-int, возвращающую случайные целые числа:

```
user=> (take 10 (repeatedly #(rand-int 10)))
(2 6 2 8 8 5 9 2 5 5)
```

С ее помощью можно создать большой вектор со случайными числами, как показано ниже:

```
reducers.core=> (def numbers (into [] (take 10000000 (repeatedly
  #(rand-int 10)))))
#'reducers.core/numbers
```

А затем подсчитать число вхождений каждого числа с помощью frequencies и parallel-frequencies:

```
reducers.core=> (require ['reducers.parallel-frequencies :refer :all])
nil
reducers.core=> (time (frequencies numbers))
"Elapsed time: 1500.306 msecs"
{0 1000983, 1 999528, 2 1000515, 3 1000283, 4 997717, 5 1000101,
 6 999993, ...}
reducers.core=> (time (parallel-frequencies numbers))
"Elapsed time: 436.691 msecs"
{0 1000983, 1 999528, 2 1000515, 3 1000283, 4 997717, 5 1000101,
 6 999993, ...}
```

¹⁶ http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/Reducers/src/reducers/parallel_frequencies.clj

Как видите, последовательной версии потребовалось примерно 1500 мсек, а параллельной – чуть больше 400 мсек, то есть примерно в 3.5 раза меньше.

В завершение второго дня

Подошел к концу второй день нашего обсуждения поддержки параллелизма в языке Clojure. Завтра мы перейдем к реализации конкуренции с применением механизмов *future* и *promise* и посмотрим, как они дают возможность использовать потоковый стиль программирования.

Что мы узнали во второй день

Операции с последовательностями в языке Clojure легко и просто поддаются распараллеливанию:

- операцию отображения (`map`) можно распараллелить с помощью функции `pmap`, параллельной версии `map`;
- для большей эффективности можно организовать параллельную обработку пакетами, задействовав функцию `partition-all`;
- подобно `pmap`, функция `fold` распараллеливает операции свертки, используя алгоритм половинного деления;
- версии функций (таких как `map`, `mapcat` и `filter`) из библиотеки `clojure.core.reducers` возвращают редуцируемые значения вместо самих последовательностей, которые можно считать рецептами, описывающими порядок свертки последовательности.

Упражнения ко второму дню

Найдите:

- видеоролик Рича Хикки (Rich Hickey), где он рассказывает о редуцентах (`reducers`) на конференции QCon 2012;
- документацию с описанием `pcalls` и `pvalues` – чем они отличаются от `pmap`? Можно ли реализовать их с помощью `pmap`?

Задания:

- создайте функции `my-flatten` и `my-mapcat` по аналогии с `my-map` (см. реализацию `my-map`). Обратите внимание, что обе они сложнее `my-map`, потому что должны будут развернуть каждый элемент исходной последовательности в один или более элементов выходной последовательности. В случае затрудне-

ний загляните в реализацию, входящую в состав загружаемого пакета примеров для этой книги;

- создайте функцию `my-filter`. Она также немного сложнее функции `my-map`, потому что должна отбросить некоторые элементы исходной последовательности.

День 3: функциональная конкуренция

В течение двух предыдущих дней мы сосредоточились на параллелизме. Сегодня мы перейдем к обсуждению вопросов, связанных с конкуренцией. Но перед этим попробуем понять, почему функциональное программирование до такой степени упрощает параллельное программирование.

Та же структура, разный порядок вычислений

Главной темой, пронизывающей все, что мы обсуждали в предыдущие два дня, является простота, с которой функциональное программирование позволяет обходить проблемы, связанные с порядком вычислений. Если вычисления могут протекать независимо, мы можем выполнять их в любом порядке, в том числе и параллельно.

Все следующие фрагменты кода выполняют одни и те же вычисления, возвращают один и тот же результат и имеют практически идентичную структуру, но операции в них выполняются в разном порядке:

```
(reduce + (map (partial * 2) (range 10000)))
```

Выполняет свертку ленивой последовательности — элементы любой ленивой последовательности генерируются только в момент обращения к ним.

```
(reduce + (doall (map (partial * 2) (range 10000))))
```

Сначала генерирует последовательность целиком (`doall` обеспечивает полную реализацию ленивой последовательности), а затем выполняет ее свертку.

```
(reduce + (pmap (partial * 2) (range 10000)))
```

Выполняет свертку частично ленивой последовательности, части которой генерируются параллельно.

```
(reduce + (r/map (partial * 2) (range 10000)))
```

Выполняет свертку единственной ленивой последовательности с помощью единственной функции, сконструированной путем объединения функций `+` и `(partial * 2)`.

```
(r/fold + (r/map (partial * 2) (into [] (range 10000))))
```

Сначала генерирует весь диапазон чисел и затем выполняет параллельную свертку, создавая дерево из операций свертки и объединения.

В императивных языках программирования, таких как Java, порядок, в каком выполняются операции, тесно связан с порядком их следования в исходном коде. Компилятор и среда выполнения могут менять местами некоторые операции (что требует повышенного внимания с нашей стороны при использовании потоков и блокировок, как было показано в разделе «Загадочная память» в главе 2), но в общем случае они выполняются в том порядке, в каком мы их записали в исходном коде.

Функциональные языки имеют более *декларативные* черты. Обычно функциональная программа – это не последовательность инструкций, описывающая, *как* получить результат, а скорее рецепт, описывающий, *что* должно получиться на выходе. Порядок вычислений, которые необходимо выполнить для достижения поставленной цели, в них не имеет большого значения – такая свобода от жестко заданного порядка и обеспечивает простоту распараллеливания функционального кода.

В следующем разделе мы увидим, почему функциональные программы могут быть независимыми от порядка вычислений, а императивные – нет.

Ссылочная прозрачность

Истинные функции являются *ссылочно-прозрачными* (referentially transparent) – где бы ни вызывалась эта функция, мы можем заменить вызов его результатом, не повлияв на поведение программы. Взгляните на следующий пример:

```
(+ 1 (+ 2 3))
```

Он в точности эквивалентен нижеприведенному фрагменту:

```
(+ 1 5)
```


И действительно, функциональный код часто анализируют, подставляя вместо вызовов функций их результаты, пока не будет получен конечный результат. Например, мы можем вычислить $(+ (+ 1 2) (+ 3 4))$ разными способами:

$(+ (+ 1 2) (+ 3 4)) \rightarrow (+ (+ 1 2) 7) \rightarrow (+ 3 7) \rightarrow 10$

или:

$(+ (+ 1 2) (+ 3 4)) \rightarrow (+ 3 (+ 3 4)) \rightarrow (+ 3 7) \rightarrow 10$

Конечно, то же самое верно и в отношении оператора $+$ в языке Java, но в функциональной программе *все* функции являются ссылочно-прозрачными. Именно это обстоятельство позволяет безопасно вносить радикальные изменения в порядок вычислений, как уже не раз было показано выше.



Вопрос Джо: действительно ли Clojure является гибридным функциональным языком?

Как будет показано в следующей главе, Clojure *не* является чисто функциональным языком программирования – он дает возможность писать функции, которые имеют побочные эффекты и потому не являются ссылочно-прозрачными.

Как оказывается, это обстоятельство не является существенным для практики, потому что побочные эффекты крайне редки в идиоматическом коде на языке Clojure и весьма очевидны, когда имеют место быть. Есть несколько простых правил, следуя которым, вы сможете создавать функции с побочными эффектами и избегать проблем с изменяющимся порядком вычислений.

Потоки данных

Я думаю, вам небезынтересно будет посмотреть, как движутся потоки данных между функциями. На рис. 10 изображен граф движения потоков данных внутри выражения $(+ (+ 1 2) (+ 3 4))$.

Результаты выражений $(+ 1 2)$ и $(+ 3 4)$ никак не зависят друг от друга, поэтому теоретически они могут вычисляться в любом порядке, даже конкурентно (то есть параллельно). Однако последнее сложение можно выполнить только после вычисления обоих подвыражений.

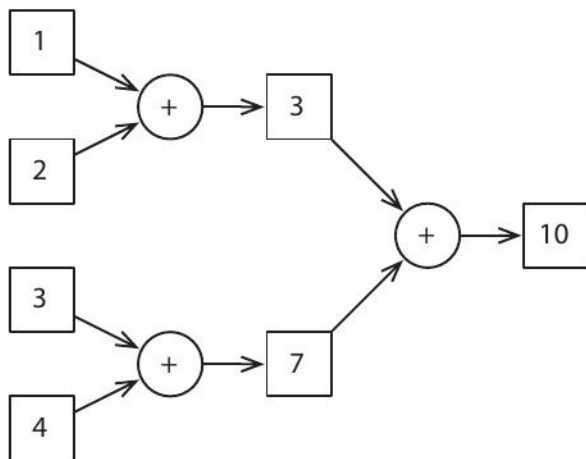


Рис. 10. Граф движения потоков данных внутри выражения $(+ (+ 1 2) (+ 3 4))$

В теории среда выполнения может начать определение результата с левой стороны графа и «проталкивать» данные вправо. Всякий раз, когда данные, связанные с входными параметрами функций, становятся доступными, эти функции вызываются. При этом все функции могли бы (опять же теоретически) выполняться конкурентно. Такой стиль выполнения называется *потокowym программированием*. Язык Clojure позволяет использовать только что описанную стратегию с применением механизмов `future` и `promise`.

Механизм `future`

Форма `future` принимает блок кода и выполняет его в другом потоке, возвращая значение в виде объекта.

```

user=> (def sum (future (+ 1 2 3 4 5)))
#'user/sum
user=> sum
#<core$future_call$reify__6110@5d4ee7d0: 15>
  
```

Получить значение объекта `future` можно, разыменовав его с помощью `deref` или `@`:

```

user=> (deref sum)
15
user=> @sum
15
  
```

Операция разыменования объекта `future` будет заблокирована, пока его значение не будет вычислено (то есть *реализовано*). Мы можем использовать эту особенность, чтобы в точности воспроизвести граф потока данных, изображенный на рис. 10:

```
user=> (let [a (future (+ 1 2))
            #_=> b (future (+ 3 4))]
        #_=> (+ @a @b))
10
```

Здесь мы использовали форму `let`, чтобы связать имя `a` с выражением `(future (+ 1 2))`, а имя `b` с выражением `(future (+ 3 4))`. Вычисление выражения `(+ 1 2)` в данном случае будет выполняться в одном потоке, а выражения `(+ 3 4)` — в другом. Наконец, внешнее сложение будет заблокировано до момента, пока не будут выполнены внутренние сложения.

Конечно, бессмысленно использовать механизм `future` для таких коротких операций, как сложение двух чисел, поэтому двинемся дальше и рассмотрим более реалистичный пример. Но перед этим познакомимся с еще одним механизмом конкурентных вычислений в языке Clojure — `promise`.

Механизм *promise*

Механизм `promise` близко напоминает механизм `future` в том смысле, что значение вычисляется асинхронно и доступно путем разыменования с помощью `deref` или `@`, причем операция разыменования так же блокируется до момента, пока значение не будет вычислено. Разница заключается в том, что при создании объекта `promise` никакие вычисления не запускаются — результат должен быть установлен с помощью `deliver`. Эта особенность иллюстрируется в следующем листинге сеанса работы в интерактивной оболочке REPL:

```
user=> (def meaning-of-life (promise))
#'user/meaning-of-life
user=> (future (println "The meaning of life is:" @meaning-of-life))
#<core$future_call$reify__6110@224e59d9: :pending>
user=> (deliver meaning-of-life 42)
#<core$promise$reify__6153@52c9f3c7: 42>
The meaning of life is: 42
```

Здесь сначала создается объект `promise` с именем `meaning-of-life`, а затем вызывается `future` для создания потока, который выведет значение этого объекта (использование `future` для создания еще одного

потока выполнения, как в данном примере, является типичной идеей (Clojure). Наконец, вызывается функция `deliver` для установки значения объекта `promise`, в результате чего разблокируется поток, созданный перед этим.

Теперь, когда мы знаем, как работают механизмы `future` и `promise`, рассмотрим их применение в настоящем приложении.

Функциональная веб-служба

В этом разделе мы реализуем веб-службу, которая в режиме реального времени принимает поток данных (например, стенограмму телевизионной передачи) и передает ее дальше. Поток данных делится на «фрагменты», и каждому фрагменту присваивается свой порядковый номер. В роли потока данных мы будем использовать первое четверостишие из поэмы «Бармаглот» (из повести-сказки «Алиса в Зазеркалье» Льюиса Кэрролла (Lewis Carroll))¹⁷:

0 Twas brillig, and the slithy toves
1 Did gyre and gimble in the wabe:
2 All mimsy were the borogoves,
3 And the mome raths outgrabe.

Фрагмент с порядковым номером 0 отправляется веб-службе в виде запроса `PUT` с телом «Twas brillig, and the slithy toves» по адресу `/snippet/0`. Фрагмент 1 отправляется по адресу `/snippet/1` и т. д.

Веб-служба имеет очень простой API, но он не так прост в реализации, как могло бы показаться. Во-первых, из-за того что веб-служба будет выполняться в конкурентном окружении веб-сервера, она должна быть потокобезопасной. Во-вторых, компьютерные сети таковы, каковы они есть, они могут терять фрагменты, повторно передавать их, доставлять один и тот же фрагмент многократно, и порядок прибытия фрагментов может отличаться от порядка отправки.

Если фрагменты должны обрабатываться в строгой последовательности (независимо от того, в каком порядке они прибывают), необходимо следить за тем, какие фрагменты уже приняты и какие из них обработаны. И каждый раз, принимая новый фрагмент, требуется проверить, какие фрагменты можно обработать (если таковые имеются). Реализовать такую последовательную обработку совсем

¹⁷ Варкалось. Хливкие шорьки
Пырялись по наве,
И хрюкотали зелюки,
Как мюмзики в мове.
Перевод Дины Орловской. – Прим. перев.

непросто – далее я покажу, как с помощью механизмов поддержки конкуренции можно получить достаточно простое решение.

На рис. 11 изображена структура решения, которое будет реализовано далее.

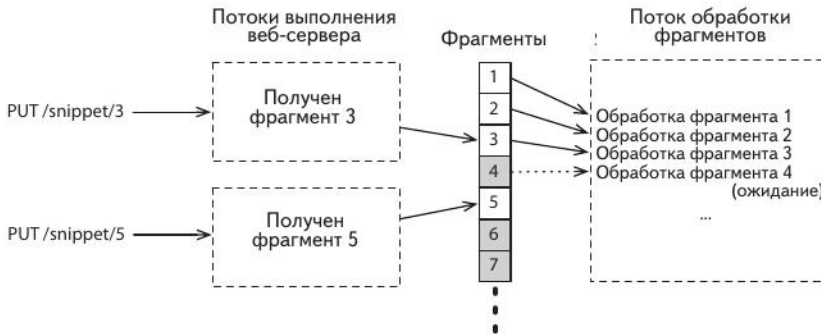


Рис. 11. Структура решения для обработки потока данных

Слева изображены потоки выполнения, запускаемые веб-сервером для обработки входящих запросов. Справа – потоки выполнения, последовательно обрабатывающие фрагменты и ожидающие, пока требуемые фрагменты будут получены. В следующем разделе мы поговорим о `snippets` – структуре данных, используемой для взаимодействий между этими потоками выполнения.

Прием фрагментов

Ниже показано, как реализуется слежение за принятыми фрагментами:

```
FunctionalProgramming/TranscriptHandler/src/server/core.clj18
```

```
(def snippets (repeatedly promise))
```

Итак, `snippets` – это бесконечная ленивая последовательность объектов `promise` (можете вставить сюда свою шутку из разряда «программисты против маркетологов» про бесконечную последовательность невыполненных обещаний¹⁹). Эти объекты `promise` будут

¹⁸ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler/src/server/core.clj>

¹⁹ Здесь игра слов: `promise` переводится как обещание, обязательство, соответственно, конструкция `repeatedly promise` для англоязычного читателя будет выглядеть как «неоднократные обещания». – Прим. перев.

реализовываться (то есть получать значения) функцией `accept-snippet` при получении фрагментов:

```
FunctionalProgramming/TranscriptHandler/src/server/core.clj20
```

```
(defn accept-snippet [n text]
  (deliver (nth snippets n) text))
```

Чтобы обеспечить последовательную обработку фрагментов, нужно просто создать поток выполнения, который будет разыменовывать каждый объект `promise` по очереди. В качестве иллюстрации ниже приводится реализация такого потока, который просто выводит фрагменты в порядке их следования:

```
FunctionalProgramming/TranscriptHandler/src/server/core.clj21
```

```
(future
  (doseq [snippet (map deref snippets)]
    (println snippet)))
```

Здесь последовательная обработка последовательности реализована с помощью `doseq`. В данном случае обрабатываемая последовательность представляет собой ленивую последовательность из разыменованных объектов `promise`, каждый из которых связан со своим фрагментом данных.

Нам осталось лишь включить весь этот код в веб-службу, как показано ниже, где с этой целью используется библиотека `Compojure`:²²

```
FunctionalProgramming/TranscriptHandler/src/server/core.clj23
```

```
(defroutes app-routes
  (PUT "/snippet/:n" [n :as {:keys [body]}]
    (accept-snippet (edn/read-string n) (slurp body))
    (response "OK")))

(defn -main [& args]
  (run-jetty (site app-routes) {:port 3000}))
```

²⁰ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler/src/server/core.clj>

²¹ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler/src/server/core.clj>

²² <https://github.com/weavejester/compojure>

²³ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler/src/server/core.clj>

Здесь определяется единственный маршрут для запросов `PUT`, который вызывает нашу функцию `accept-snippet`. Веб-служба будет выполняться под управлением встраиваемого веб-сервера `Jetty`²⁴, который, как и большинство веб-серверов, является многопоточным, вследствие чего наш код должен быть потокобезопасным.

После запуска веб-сервера (командой `lein run`) можно воспользоваться утилитой `curl`, чтобы убедиться, что веб-служба действует в полном соответствии с нашими ожиданиями. Отправим, для примера, фрагмент с порядковым номером 0:

```
$ curl -X put -d "Twas brillig, and the slithy toves" \  
-H "Content-Type: text/plain" localhost:3000/snippet/0  
OK
```

На экране тут же появится:

```
Twas brillig, and the slithy toves
```

Но если отправить фрагмент 2 до того, как будет отправлен фрагмент 1, на экран ничего выведено не будет:

```
$ curl -X put -d "All mimsy were the borogoves," \  
-H "Content-Type: text/plain" localhost:3000/snippet/2  
OK
```

Однако после отправки фрагмента 1:

```
$ curl -X put -d "Did gyre and gimble in the wabe:" \  
-H "Content-Type: text/plain" localhost:3000/snippet/1  
OK
```

веб-служба выведет оба фрагмента, 1 и 2:

```
Did gyre and gimble in the wabe:  
All mimsy were the borogoves,
```

Если отправить один и тот же фрагмент еще раз, это не вызовет никаких проблем, потому что функция `deliver` ничего не делает, если вызвать ее повторно с уже реализованным объектом `promise`. То есть следующая попытка не вызовет ни ошибки, ни вывода фрагмента на экран:

```
$ curl -X put -d "Did gyre and gimble in the wabe:" \  
-H "Content-Type: text/plain" localhost:3000/snippet/1  
OK
```

²⁴ <http://www.eclipse.org/jetty/>

Теперь, когда мы увидели, как можно обрабатывать фрагменты, перейдем к решению более интересной задачи. Представьте, что нам нужна еще одна веб-служба, передающая дальше любые предложения, которые она получит. Для этого мы изменим обработчик потока данных, чтобы обеспечить передачу всего, что будет получено веб-службой.

Предложения

Прежде чем перейти к экспериментам с новой веб-службой, необходимо реализовать код, который будет превращать последовательность фрагментов в последовательность предложений. Границы предложений могут появляться в любом месте внутри фрагментов, соответственно, мы должны либо разбивать фрагменты на предложения, либо объединять их в одно предложение.

Начнем с разбиения фрагментов на предложения:

FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj²⁵

```
(defn sentence-split [text]
  (map trim (re-seq #"^[^\.\!?\?:;]+\.[^\.\!?\?:;]*" text)))
```

Здесь мы передаем регулярное выражение, соответствующее предложению, функции `re-seq`, которая возвращает последовательность совпадений и использует `trim` для удаления любых лишних пробелов:

```
server.core=> (sentence-split "This is a sentence. Is this?! A fragment")
("This is a sentence." "Is this?!" "A fragment")
```

Далее, добавив еще немного волшебства регулярных выражений, мы получаем функцию, с помощью которой сможем узнавать, является ли строка предложением:

FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj²⁶

```
(defn is-sentence? [text]
  (re-matches #"^[^\.\!?\?:;]*$" text))

server.core=> (is-sentence? "This is a sentence.")
"This is a sentence."
server.core=> (is-sentence? "A sentence doesn't end with a comma,")
nil
```

²⁵ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj>

²⁶ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj>

Наконец, свяжем все это воедино, в виде функции `strings->sentences`, принимающей последовательность строк и возвращающей последовательность предложений:

FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj²⁷

```
(defn sentence-join [x y]
  (if (is-sentence? x) y (str x " " y)))

(defn strings->sentences [strings]
  (filter is-sentence?
    (reductions sentence-join
      (mapcat sentence-split strings))))
```

Она использует функцию `reductions`. Как можно предположить, судя по имени, она действует подобно функции `reduce`; но возвращает не единственное значение, а последовательность всех промежуточных значений:

```
server.core=> (reduce + [1 2 3 4])
10
server.core=> (reductions + [1 2 3 4])
(1 3 6 10)
```

В нашем случае в качестве функции свертки мы используем `sentence-join`. Если ее первый аргумент является предложением, она просто вернет свой второй аргумент. Но если первый аргумент не является предложением, она вернет первый и второй аргументы, объединив их и вставив между ними пробел:

```
server.core=> (sentence-join "A complete sentence." "Start of another")
"Start of another"
server.core=> (sentence-join "This is" "a sentence.")
"This is a sentence."
```

В соединении с функцией `reductions` мы получаем следующее:

```
server.core=> (def fragments ["A" "sentence." "And another." "Last"
  "sentence."])
#'server.core/fragments
server.core=> (reductions sentence-join fragments)
("A" "A sentence." "And another." "Last" "Last sentence.")
```

В заключение отфильтруем результаты с помощью `is-sentence?`:

²⁷ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler2/src/server/sentences.clj>

```
server.core=> (filter is-sentence? (reductions sentence-join fragments))
("A sentence." "And another." "Last sentence.")
```

Теперь, когда у нас появилась последовательность предложений, ее можно передать дальше.

Перевод предложений

Механизм `future` часто используется для организации взаимодействий с другими веб-службами. Он позволяет выполнять операции, такие как передача данных по сети, в другом потоке, давая возможность главному потоку заниматься другими делами. Ниже представлена функция `translate`, возвращающая объект `future`, который после реализации будет содержать перевод аргумента на другой язык:

FunctionalProgramming/TranscriptHandler2/src/server/core.clj²⁸

```
(def translator "http://localhost:3001/translate")

(defn translate [text]
  (future
    (:body (client/post translator {:body text}))))
```

Она использует функцию `client/post` из библиотеки `clj-http`, которая отправляет запрос `POST` и получает ответ.²⁹ Мы можем использовать ее для преобразования результата, возвращаемого функцией `strings->sentences`, во множество предложений, переведенных на другой язык:

FunctionalProgramming/TranscriptHandler2/src/server/core.clj³⁰

```
(def translations
  (delay
    (map translate (strings->sentences (map deref snippets)))))
```

Здесь мы впервые использовали функцию `delay`, создающую ленивое значение, которое будет реализовано при попытке разыменования.

²⁸ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler2/src/server/core.clj>

²⁹ <https://github.com/dakrone/clj-http>

³⁰ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler2/src/server/core.clj>

Объединяем все вместе

Ниже приводится полный исходный код веб-службы:

FunctionalProgramming/TranscriptHandler2/src/server/core.clj³¹

```
1 (def snippets (repeatedly promise))
-
- (def translator "http://localhost:3001/translate")
-
5 (defn translate [text]
-   (future
-     (:body (client/post translator {:body text}))))
-
- (def translations
10   (delay
-     (map translate (strings->sentences (map deref snippets)))))
-
- (defn accept-snippet [n text]
-   (deliver (nth snippets n) text))
15
- (defn get-translation [n]
-   @(nth @translations n))
-
- (defroutes app-routes
20   (PUT "/snippet/:n" [n :as {:keys [body]})
-     (accept-snippet (edn/read-string n) (slurp body))
-     (response "OK"))
-   (GET "/translation/:n" [n]
-     (response (get-translation (edn/read-string n)))))
25
- (defn -main [& args]
-   (run-jetty (wrap-charset (api app-routes)) {:port 3000}))
```

Кроме кода, осуществляющего перевод предложений, мы добавили новую конечную точку GET, чтобы дать возможность извлекать перевод (строка 23). Она включает в работу функцию `get-translation` (строка 16), которая обращается к последовательности переведенных предложений, созданной ранее.

Если хотите увидеть все это в действии, запустите сервер вместе с сервером перевода, который включен в загружаемый пакет примеров для книги. Затем запустите приложение `TranscriptTest` (также включено в загружаемый пакет примеров), и вы должны увидеть предложения из поэмы «Бармаглот», переведенные на французский язык:

³¹ <http://media.pragprog.com/titles/pb7con/code/FunctionalProgramming/TranscriptHandler2/src/server/core.clj>

\$ lein run

Il brilgue, les têtes lubricilleux Se gyrent en vrillant dans le guave:

Enmîmés sont les gougebosqueux Et le mômérade horsgrave.

Garde-toi du Jaseroque, mon fils!

La gueule qui mord; la griffe qui prend!

Garde-toi de l'oiseau Jube, évite Le frumieux Band-à-prend!

"..."

Итак, мы реализовали конкурентную веб-службу, использующую комбинацию из ленивых последовательностей и объектов `future` и `promise`. В ней отсутствуют изменяемое состояние и блокировки, и ее реализация получилась намного проще, чем реализация эквивалентной службы на императивном языке.



Вопрос Джо: Не пора ли хвататься за голову?

Наша веб-служба использует две ленивые последовательности, `snippets` и `translations`. Это означает, держитесь крепче, что данные последовательности (см. раздел «Лень – это благо» выше) будут постоянно расти, потребляя все больше и больше памяти с течением времени.

В следующей главе мы увидим, как пользоваться ссылочными типами `Clojure` для исправления этой проблемы и расширения веб-службы возможностью перевода на множество языков.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения приемов функционального программирования реализации параллельных и конкурентных программ.

Что мы узнали в третий день

Функции в функциональном программировании являются ссылочно-прозрачными. Это обстоятельство позволяет свободно изменять порядок вызова этих функций, не опасаясь, что это повлияет на поведение программы. В частности, это дает возможность использовать потоковый стиль программирования (в языке `Clojure` поддерживается посредством механизмов `future` и `promise`), когда код выполняется только после появления всех необходимых данных. Мы видели, как потоковый стиль может упростить реализацию веб-службы.

Упражнения к третьему дню

Найдите:

- чем отличаются функции `future` и `future-call`? Как бы вы реализовали одну на основе другой?
- как определить, что объект `future` был реализован, не заблокировав при этом потока выполнения? Как остановить поток, ожидающий реализации `future`?

Задания:

- измените сервер перевода так, чтобы запрос `GET` к адресу `/translation/:n` не блокировался, когда перевод еще недоступен, а возвращал бы код состояния HTTP 409;
- реализуйте сервер перевода на императивном языке программирования. Получилось ли императивное решение более простым, чем функциональное? Уверены ли вы, что созданное вами императивное решение не подвержено состоянию гонки?

В завершение

Существует распространенное заблуждение, касающееся параллелизма, – многие полагают, что параллельное программирование обязательно влечет за собой проблему недетерминированного поведения. Если операции не выполняются последовательно, рассуждают они, и мы не можем больше полагаться на определенный порядок выполнения операций, мы вынуждены будем постоянно беспокоиться о состоянии гонки.

Конечно, есть некоторые конкурентные программы, которые всегда будут недетерминированными. И это неизбежно – некоторые проблемы требуют применения решений, которые основаны на синхронизации. Но не все параллельные программы обязательно будут недетерминированными. Сумма всех чисел от 0 до 10 000 не может измениться только лишь потому, что операции сложения будут выполняться параллельно, а не последовательно. Частоты встречаемости слов в том же дампе Википедии всегда будут одними и теми же, независимо от того, сколько потоков выполнения будут осуществлять подсчет.

В подавляющем большинстве случаев появление состояния гонки в традиционных приложениях, реализованных на основе потоков и блокировок, обусловлено особенностями решения, а не потому, что недетерминизм является свойством решаемой задачи.

Благодаря тому, что функциональный код является ссылочно-прозрачным, мы легко можем изменять порядок выполнения операций, не опасаясь, что это приведет к изменению конечных результатов. Как мы уже видели, возможность параллельного выполнения взаимно независимых функций позволяет легко распараллеливать функциональный код.



Вопрос Джо: а где же монады и моноиды?

Введения в функциональное программирование часто скатываются в описание математических понятий, таких как монады, моноиды и теория категорий. Мы только что закончили целую главу, посвященную функциональному программированию, и ни разу не столкнулись с этими понятиями. Почему?

Самое большое, пожалуй, влияние на особенности любого языка программирования оказывает его система типов. Создание программного кода на статически типизированном языке, таком как Java или Scala, совершенно отличается от создания кода на языке с динамической системой типов, таком как Ruby или Python.

Все хлопоты по правильному определению типов данных статические системы типов перекладывают на плечи программистов. Это позволяет компиляторам выявлять многие ошибки на этапе компиляции и применять специфические оптимизации, повышающие производительность кода.

При использовании динамически типизированного языка программист избавляется от этих хлопот, но соглашается с риском, что некоторые ошибки будут возникать во время выполнения, а скомпилированный код окажется менее эффективным.

Те же различия наблюдаются и в функциональном мире. Статически типизированные функциональные языки, такие как Haskell, используют понятие монад и моноидов, чтобы позволить с помощью своей системы типов точно обозначить ограничения на использование функций и значений и сохранить возможность создания побочных эффектов, оставаясь при этом функциональными языками.

Несмотря на то что знание и понимание этих математических концепций, безусловно, будет полезно при разработке программ на языке Clojure, отсутствие статической системы типов избавляет от необходимости обсуждать их. Но такое положение вещей имеет и отрицательную сторону – динамическая типизация требует, чтобы программист уделял особое внимание использованию функций и значений в соответствующих контекстах – компилятор не сможет предупредить вас, если вы попытаетесь использовать функцию или данные не по назначению.

Сильные стороны

Главным достоинством функционального программирования является уверенность. Уверенность, что ваша программа делает именно то, что задумывалось. Как только вы научитесь мыслить в функциональном стиле (для чего может потребоваться некоторое время, особенно если вы имеете многолетний опыт императивного программирования), функциональные программы будут получаться более простыми, проще поддаваться тестированию и изучению, чем их императивные эквиваленты.

Имея действующее функциональное решение, благодаря ссылочной прозрачности вы легко сможете распараллелить его или организовать конкурентное выполнение задач. Так как функциональный код не имеет изменяемого состояния, в нем просто отсутствует большинство проблем конкуренции, так характерных для традиционных программ на основе потоков и блокировок.

Слабые стороны

Многие полагают, что функциональный код менее эффективен, чем его императивный эквивалент. Да, некоторые решения определенно влекут за собой снижение производительности, однако проигрыш наверняка будет меньше, чем вы ожидаете, а небольшая потеря производительности – не слишком высокая цена за более высокую надежность и масштабируемость.

Другие языки

В Java 8 недавно появилось множество нововведений, которые упрощают программирование в функциональном стиле, наиболее существенными из них являются поддержка *lambda-выражений*³² и *потоков данных*³³.

Потоки данных (streams) поддерживают агрегатные операции, с помощью которых можно организовать параллельную обработку данных способом, напоминающим использование редукторов в Clojure.

Никакое описание функционального программирования не может считаться полным без упоминания о языке Haskell.³⁴ В языке Haskell имеются эквиваленты всего, что демонстрировалось в этой главе, и даже больше. Отличное введение в параллельное и конкурентное

³² <http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>

³³ <http://docs.oracle.com/javase/tutorial/collections/streams/index.html>

³⁴ <http://haskell.org/>

программирование на языке Haskell можно найти в книге Саймона Марлоу (Simon Marlow).³⁵

Напоследок

Помимо превосходной поддержки конкуренции и параллелизма, демонстрировавшейся в этой главе, функциональное программирование имеет огромное число других интересных особенностей. Сейчас уже кажется неизбежным, что в будущем функциональное программирование будет играть все более важную роль.

С другой стороны, изменяемое состояние еще долго будет сопровождать нас. В следующей главе мы познакомимся с тем, как поддержка побочных эффектов в языке Clojure уживается с его поддержкой конкуренции.

³⁵ <http://community.haskell.org/~simonmar/par-tutorial.pdf>

Марлоу Саймон. Параллельное и конкурентное программирование на Haskell». – ISBN: 978-5-94074-984-4. – ДМК-Пресс, 2014. – *Прим. перев.*



ГЛАВА 4.

Путь Clojure – разделение идентичности и состояния

Современный гибридный легковой автомобиль содержит две силовые установки: двигатель внутреннего сгорания и электромотор. В зависимости от условий иногда он приводится в движение только электромотором, иногда только бензиновым двигателем, а иногда и тем, и другим вместе. Язык Clojure представляет собой похожий гибрид, соединяющий возможности функционального программирования с изменяемым состоянием, — «путь Clojure» заключается в использовании возможностей обоих подходов, чтобы поднять параллельное программирование на новую высоту.

Лучшее из двух миров

Функциональное программирование отлично подходит для решения многих задач, однако есть такие задачи, решение которых трудно представить без использования изменяемого состояния. Да, для таких задач тоже можно найти функциональное решение, но часто бывает гораздо проще использовать традиционное решение. В этой главе выйдем за рамки подмножества чисто функциональных инструментов Clojure, с которым мы познакомились в предыдущей главе, и посмотрим, как на этом языке создавать конкурентные решения для подобных задач.

В первый день мы познакомимся с атомами, простейшим из типов данных в языке Clojure, поддерживающих конкуренцию, и посмотрим, как в комплексе с неизменяемыми структурами данных они позволяют отделять идентичность от состояния. Во второй день мы займемся исследованием других изменяемых структур данных в Clojure: агентов и программной транзакционной памяти. Наконец, в

третий день мы реализуем алгоритм на основе атомов и программной транзакционной памяти (Software Transactional Memory, STM), и обсудим достоинства и недостатки двух решений.

День 1: атомы и сохраненные структуры данных

Чисто функциональные языки не поддерживают изменяемые данные ни в каком виде. Clojure, напротив, является *гибридным* функциональным языком — он поддерживает множество разновидностей изменяемых переменных, поддерживающих конкуренцию, каждая из которых предназначена для решения своего круга задач. Все они вместе с сохраненными структурами данных (что означает в данном случае определение *сохраненные* (persistence), я расскажу ниже) помогают избежать множества проблем, присущих традиционным конкурентным программам с разделяемым и изменяемым состоянием.

Разница между гибридным функциональным и императивным языками заключается в акцентах. В императивном языке переменные являются изменяемыми по умолчанию, и идиоматический код часто изменяет их значения. В гибридном функциональном языке переменные являются неизменяемыми по умолчанию, и идиоматический код изменяет их, только когда это совершенно необходимо. Как будет показано далее, изменяемые переменные в Clojure позволяют воссоздавать побочные эффекты, оставаясь при этом безопасными и непротиворечивыми.

Сегодня мы увидим, как изменяемые переменные в языке Clojure работают в комплексе с сохраненными структурами данных для разделения идентичности и состояния. Это позволяет нескольким потокам выполнения обращаться к изменяемым переменным конкурентно, без применения блокировок (и, соответственно, без риска возникновения взаимоблокировок) и без проблем, связанных со скрытым или ускользающим скрытым состоянием, как рассказывалось в разделе «Опасности изменяемого состояния» в главе 3. Мы начнем со знакомства с самой простой разновидностью изменяемых переменных в языке Clojure — *атомами*.

Атомы

Атом — это атомарная переменная, очень похожая на то, что мы видели в разделе «Атомарные переменные» в главе 2 (в действительно-

сти атомы Clojure основаны на `java.util.concurrent.atomic`). Ниже приводится пример создания атома и получения его значения:

```
user=> (def my-atom (atom 42))
#'user/my-atom
user=> (deref my-atom)
42
user=> @my-atom
42
```

Создается атом с помощью функции `atom`, которой передается начальное значение. Текущее значение атома можно получить с помощью `deref` или `@`.

Сохранение нового значения в атоме осуществляется с помощью функции `swap!`:

```
user=> (swap! my-atom inc)
43
user=> @my-atom
43
```

Она принимает функцию и передает ей текущее значение атома. Новым значением атома становится значение, возвращаемое этой функцией. Функции можно также передать дополнительные аргументы, например:

```
user=> (swap! my-atom + 2)
45
```

В первом аргументе функции всегда будет передаваться текущее значение атома, а за ним будут следовать любые дополнительные аргументы, переданные функции `swap!`. То есть в данном случае новым значением атома станет результат выражения `(+ 43 2)`.

Иногда бывает желательно установить атом в значение, не зависящее от его текущего значения. В этом случае можно воспользоваться функцией `reset!`:

```
user=> (reset! my-atom 0)
0
user=> @my-atom
0
```

Атомы могут иметь любой тип – многие веб-приложения используют атомарные ассоциативные массивы для хранения информации о сессии, как в следующем примере:

```
user=> (def session (atom {}))
#'user/session
user=> (swap! session assoc :username "paul")
{:username "paul"}
user=> (swap! session assoc :session-id 1234)
{:session-id 1234, :username "paul"}
```

Теперь, поиграв с атомами в интерактивной оболочке REPL, разберем пример их использования в приложении.

Многопоточная веб-служба с изменяемым состоянием

В разделе «Ускользящее изменяемое состояние» (глава 3) мы рассматривали гипотетическую веб-службу, управляющую списком игроков, участвующих в соревнованиях. В этом разделе мы рассмотрим законченный код на языке Clojure, реализующий такую веб-службу и демонстрирующий, как сохраненные структуры данных Clojure не позволяют ускользнуть изменчивому состоянию, как это происходит в Java.

Clojure/TournamentServer/src/server/core.clj¹

```
1 (def players (atom ()))
-
- (defn list-players []
-   (response (json/encode @players)))
5
- (defn create-player [player-name]
-   (swap! players conj player-name)
-   (status (response "") 201))
-
10 (defroutes app-routes
-   (GET "/players" [] (list-players))
-   (PUT "/players/:player-name" [player-name] (create-player player-name)))
- (defn -main [& args]
-   (run-jetty (site app-routes) {:port 3000}))
```

В этом коде определяется пара маршрутов – в ответ на запрос GET по адресу `/players` возвращается текущий список участников (в формате JSON), а запрос PUT по адресу `/players/name` добавит нового участника в список. Данная веб-служба, как и та, что демонстрировалась в предыдущей главе, будет выполняться под управлением многопо-

¹ <http://media.pragprog.com/titles/pb7con/code/Clojure/TournamentServer/src/server/core.clj>

точного встраиваемого сервера Jetty, поэтому наш код должен быть потокобезопасным.

Об особенностях этого кода мы поговорим ниже, а сейчас посмотрим на его работу. Мы можем заставить его работать из командной строки, воспользовавшись утилитой `curl`:

```
$ curl localhost:3000/players
[]
$ curl -X put localhost:3000/players/john
$ curl localhost:3000/players
["john"]

$ curl -X put localhost:3000/players/paul
$ curl -X put localhost:3000/players/george
$ curl -X put localhost:3000/players/ringo
$ curl localhost:3000/players
["ringo", "george", "paul", "john"]
```

Теперь перейдем к более близкому знакомству с кодом. Атом `players` (строка 1) инициализируется пустым списком `()`. Новый игрок добавляется в список с помощью `conj` (строка 7), и клиенту возвращается пустой ответ с кодом HTTP 201 (создано). Список игроков, что возвращается в формате JSON, извлекается из атома `players` с помощью `@` (строка 4).

Код выглядит очень простым (впрочем, так оно и есть), но кое-что в нем может вызывать у вас беспокойство. Функции `list-players` и `create-player` обращаются к списку `players` – почему же в этом коде отсутствует проблема, что проявлялась в коде на Java, который демонстрировался в разделе «Ускользящее изменяемое состояние» (глава 3)? Что случится, если один поток добавит нового в список одновременно с другим потоком, выполняющим итерации по списку и преобразующим его в формат JSON?

Этот код поддерживает возможность выполнения в многопоточной среде, потому что структуры данных в Clojure являются *сохранными* (*persistent*).

Сохранные структуры данных

Определение «сохранные» в данном случае не имеет никакого отношения к сохранению данных на диске или в базе данных. Так называют структуры данных, которые при изменении сохраняют предыдущие версии, что обеспечивает непротиворечивость данных независимо от изменений. Убедиться в этом можно в интерактивной оболочке REPL:


```
user=> (def mapv1 {:name "paul" :age 45})
#'user/mapv1
user=> (def mapv2 (assoc mapv1 :sex :male))
#'user/mapv2
user=> mapv1
{:age 45, :name "paul"}
user=> mapv2
{:age 45, :name "paul", :sex :male}
```

Сохраненные структуры данных действуют так, как если бы при каждом их изменении создавалась полная копия. Однако если бы они были реализованы таким способом, они были бы крайне неэффективными и потому имели бы очень ограниченный круг применения (как класс `CopyOnWriteArrayList`, который мы видели в разделе «Копирование при записи» в главе 2). К счастью, применяется более остроумная реализация, основанная на *совместном использовании структур*.

Проще всего понять суть сохраненных структур на примере списка. Ниже приводится простой список:

```
user=> (def listv1 (list 1 2 3))
#'user/listv1
user=> listv1
(1 2 3)
```

На рис. 12 показано, как этот список хранится в памяти.

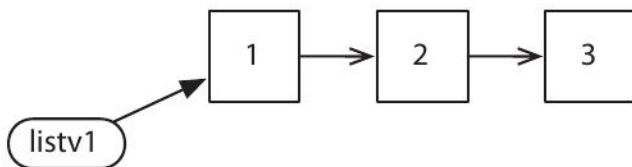


Рис. 12. Исходная версия списка в памяти

Теперь создадим измененную версию с помощью функции `cons`, возвращающей копию списка с элементом, добавленным в начало:

```
user=> (def listv2 (cons 4 listv1))
#'user/listv2
user=> listv2
(4 1 2 3)
```

Новый список может использовать все элементы прежней версии списка без копирования, как показано на рис. 13.

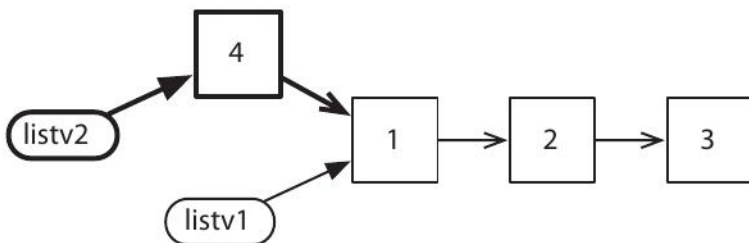


Рис. 13. Измененная версия списка в памяти

Наконец, создадим еще одну версию списка, изображенную на рис. 14:

```
3user=> (def listv3 (cons 5 (rest listv1)))
#'user/listv3
user=> listv3
(5 2 3)
```

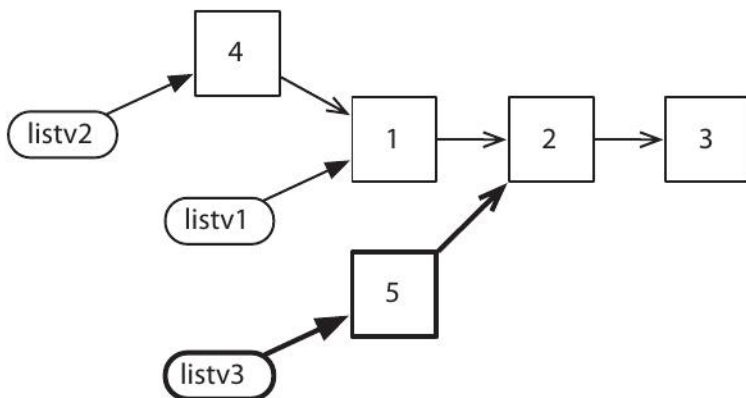


Рис. 14. Еще одна измененная версия списка в памяти

Этот новый экземпляр списка использует часть исходного списка, но копирование при этом не выполняется.

Однако не всегда есть возможность избежать копирования. Работу со списками можно оптимизировать, только если они имеют общий хвост, – если нужно создать два списка с разными хвостами, у нас не остается никаких вариантов, кроме копирования. Например:

```
user=> (def listv1 (list 1 2 3 4))
#'user/listv1
user=> (def listv2 (take 2 listv1))
#'user/listv2
```

```
user=> listv2
(1 2)
```

В результате мы получаем списки, которые в памяти располагаются, как показано на рис. 15.

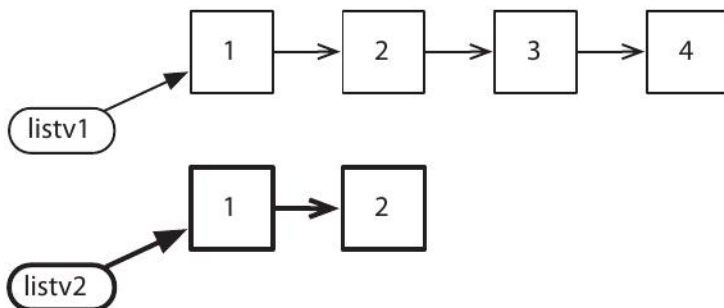


Рис. 15. Списки с разными хвостами не могут иметь общей части

Все коллекции в языке Clojure являются сохраненными. Сохраненные векторы, ассоциативные массивы и множества имеют более сложную реализацию, чем списки, но нам пока достаточно знать, что они являются разделяемыми структурами и обеспечивают производительность, сопоставимую с их традиционными аналогами в таких языках, как Ruby и Java.



Вопрос Джо: могут ли быть сохраненными нефункциональные структуры данных?

Сохраненные структуры данных можно создавать на нефункциональных языках. Мы уже видели пример такой структуры на Java (`CopyOnWriteArrayList`), да и основные структуры данных Clojure в основном написаны на Java, потому что языка Clojure не существовало, когда они были придуманы, поэтому на данный вопрос можно ответить утвердительно.

Однако реализация сохраненных структур данных на нефункциональном языке сложна – их сложнее выразить и сложнее обеспечить высокую эффективность, потому что язык не предусматривает ничего в помощь: вам придется самим от начала и до конца реализовывать контракт хранения.

Функциональные структуры данных, напротив, автоматически являются сохраненными.

Идентичность или состояние

Сохранные структуры данных бесценны для конкурентного программирования, потому что, получив ссылку на структуру данных, поток выполнения не будет видеть изменений, производимых другим потоком выполнения. Сохранные структуры данных отделяют идентичность от состояния.

Представьте себе прибор, показывающий уровень топлива в бензобаке вашего автомобиля. Прямо сейчас он может показывать, что бензобак наполнен наполовину. Спустя какое-то время – что бензобак почти пуст, а еще через несколько минут (после заезда на заправку), что бензобак полон. *Идентичность* – «прибор, показывающий уровень топлива» – это одно, а *состояние* – постоянно изменяющийся уровень топлива – это другое. В действительности «уровень топлива» – это последовательность разных значений: в 12:30 23 февраля он может иметь значение 0.53; в 14:30 того же дня – 0.12; а в 14:31 получить значение 1.00.

Переменные в императивных языках связывают (сплетают, соединяют, сковывают) идентичность и состояние – единственная идентичность может иметь только одно значение, из-за чего легко не заметить того факта, что состояние в действительности является последовательностью значений, сменяющих друг друга с течением времени. Сохранные структуры данных отделяют идентичность от состояния – если получить текущее состояние, связанное с идентичностью, само по себе состояние будет неизменяемым, независимо от того, что потом произойдет с идентичностью, откуда мы получили данное значение.

Очень яркую аналогию можно найти в трудах Гераклита:

Дважды не войти в одну и ту же реку, ибо притекает другая вода.

Большинство языков допускают одну и ту же ошибку, полагая, что река – единый и постоянный объект; Clojure же, напротив, распознает, что она постоянно меняется.

Повторения

Так как Clojure является функциональным языком, атомы не используют блокировок – внутренне они используют метод `compareAndSet()` из `java.util.concurrent.AtomicReference`. Это означает, что они обеспечивают высокую скорость работы и не блокируют потоков выполнения (как следствие отсутствует опасность взаимоблокировок). Но это также означает, что `swap!` должна обрабатывать ситуацию, когда значение атома изменяется другим потоком в промежутке времени

между вызовом функции, генерирующей новое значение, и попыткой изменить значение, хранимое атомом.

В такой ситуации `swap!` повторит попытку. Она отбросит значение, полученное от функции изменения, и вызовет ее еще раз, передав ей новое значение атома. Нечто похожее мы наблюдали, когда использовали `ConcurrentHashMap` в разделе «Законченная программа» в главе 2. Из этого следует, что функция, которая передается функции `swap!`, не должна иметь побочных эффектов, в противном случае эти побочные эффекты могут происходить более чем один раз.

Это одна из многих ситуаций, где функциональная природа Clojure дает дополнительные преимущества — функциональный код по своей природе не имеет побочных эффектов.

Валидаторы

Представьте, что нам нужен атом, который никогда не сможет получить отрицательного значения. Обеспечить это можно, определив *валидатор* при создании атома:

```
user=> (def non-negative (atom 0 :validator #(>= % 0)))
#'user/non-negative
user=> (reset! non-negative 42)
42
user=> (reset! non-negative -1)
IllegalStateException Invalid reference state
```

Валидатор — это функция, которая вызывается при любой попытке изменить значение атома. Если она вернет `true`, попытка будет считаться успешной, но если она вернет `false`, попытка изменения будет отвергнута.

Валидатор вызывается до изменения значения атома и, так же как функция, генерирующая новое значение, может быть вызван более одного раза, если `swap!` повторит попытку. По этой причине валидаторы тоже не должны иметь побочных эффектов.

Функции-наблюдатели

Для атомов можно также назначать *функции-наблюдатели* (watchers):

```
user=> (def a (atom 0))
#'user/a
user=> (add-watch a :print #(println "Changed from " %3 " to " %4))
#<Atom@542ab4b1: 0>
user=> (swap! a + 2)
Changed from 0 to 2
2
```


При добавлении функции-наблюдателя нужно определить ключ и саму функцию. Ключи используются для идентификации функций (так, например, если с атомом связано несколько функций-наблюдателей, мы можем удалять отдельные функции, указывая соответствующие ключи). Функция-наблюдатель вызывается при каждом изменении значения атома. Она получает четыре аргумента – ключ, который передавался в вызов `add-watch`, ссылка на атом, предыдущее значение и новое значение.

В предыдущем коде снова использовался макрос `#( ... )` для определения анонимной функции, которая выводит старое (`%3`) и новое (`%4`) значения атома.

В отличие от валидаторов, функции-наблюдатели вызываются уже после изменения значения и потому вызываются только один раз, независимо от того, сколько повторных попыток выполнит функция `swap!`. Соответственно, для функции-наблюдателя допустимо иметь побочные эффекты. Однако имейте в виду, что к моменту, когда будет вызвана функция-наблюдатель, значение атома может измениться другим потоком выполнения, поэтому функции-наблюдатели всегда должны использовать значения, полученные в виде аргументов, и никогда не разыменовывать ссылку на атом.

Гибридная веб-служба

В разделе «Функциональная веб-служба» в главе 3 мы написали на языке Clojure чисто функциональную веб-службу. Хотя она замечательно работает, в ней есть пара существенных ограничений – она может выполнять перевод только на один язык, и она постоянно потребляет все больше и больше памяти. В этом разделе мы увидим, как исправить обе эти проблемы, сохранив функциональный стиль оригинала.

Управление сеансами

Обеспечить возможность выполнения множества переводов можно путем введения понятия *сеанса*. Каждый сеанс имеет уникальный числовой идентификатор, генерируемый, как показано ниже:

```
Clojure/TranscriptHandler/src/server/session.clj2
```

```
(def last-session-id (atom 0))
```

² <http://media.pragprog.com/titles/pb7con/code/Clojure/TranscriptHandler/src/server/session.clj>

```
(defn next-session-id []
  (swap! last-session-id inc))
```

Здесь используется атом `last-session-id`, который увеличивается каждый раз при получении нового идентификатора сеанса. Как результат при каждом вызове функции `next-session-id` она возвращает число, на единицу больше предыдущего:

```
server.core=> (in-ns 'server.session)
#<Namespace server.session>
server.session=> (next-session-id)
1
server.session=> (next-session-id)
2
server.session=> (next-session-id)
3
```

Мы будем следить за активными сеансами с помощью еще одного атома, `sessions`, хранящего ассоциативный массив с идентификаторами и значениями сеансов:

```
(def sessions (atom {}))

(defn new-session [initial]
  (let [session-id (next-session-id)]
    (swap! sessions assoc session-id initial)
    session-id))

(defn get-session [id]
  (@sessions id))
```

Новый сеанс создается передачей начального значения функции `new-session`, которая добавляет новый идентификатор сеанса в `sessions` вызовом `swap!`. Получение сеанса в `get-session` заключается в простом поиске сеанса по его идентификатору.

Срок действия сеанса

Если мы не хотим, чтобы веб-служба постоянно потребляла память, нам нужен некоторый механизм, который удалял бы неиспользуемые сеансы. Мы можем сами реализовать такой механизм (например, в виде функции `delete-session`), но, учитывая, что создается веб-служба, мы не можем надеяться, что клиенты будут корректно завершать свои сеансы, поэтому нам нужно организовать удаление сеансов по истечении некоторого времени. Для этого в предыдущий код нужно внести совсем немного изменений:

Clojure/TranscriptHandler/src/server/session.clj³

```
(def sessions (atom {}))

> (defn now []
>   (System/currentTimeMillis))

(defn new-session [initial]
  (let [session-id (next-session-id)
>       session (assoc initial :last-referenced (atom (now)))])
    (swap! sessions assoc session-id session)
    session-id)

(defn get-session [id]
  (let [session (@sessions id)]
>   (reset! (:last-referenced session) (now))
    session))
```

Мы добавили вспомогательную функцию `now`, возвращающую текущее время. Когда `new-session` создает новый сеанс, она добавляет в сеанс элемент `:last-referenced`, еще один атом, содержащий текущее время. Значение атома будет изменяться вызовом `reset!` при каждом вызове `get-session`.

Теперь, когда каждый сеанс имеет элемент `:last-referenced`, можно организовать периодическую проверку имеющихся сеансов и удалять те из них, которые оставались неиспользуемыми дольше установленного времени:

Clojure/TranscriptHandler/src/server/session.clj⁴

```
(defn session-expiry-time []
  (- (now) (* 10 60 1000)))

(defn expired? [session]
  (< @(:last-referenced session) (session-expiry-time)))

(defn sweep-sessions []
  (swap! sessions #(remove-vals % expired?)))

(def session-sweeper
  (schedule {:min (range 0 60 5)} sweep-sessions))
```

Здесь, в функции `session-sweeper`, используется библиотека `Schejulture` для организации вызова `sweep-sessions` каждые пять минут.⁵ Эта функция, в свою очередь, удаляет (с помощью функции

³ <http://media.pragprog.com/titles/pb7con/code/Clojure/TranscriptHandler/src/server/session.clj>

⁴ <http://media.pragprog.com/titles/pb7con/code/Clojure/TranscriptHandler/src/server/session.clj>

⁵ <https://github.com/AdamClements/schejulture>

`remove-vals` из библиотеки `Useful`⁶) все сеансы, для которых `expired?` вернет `true`, то есть сеансы, последнее обращение к которым происходило за `session-expiry-time` до текущего момента (десять или более минут назад).

Объединяем все вместе

Теперь можно добавить поддержку сеансов в веб-службу. Прежде всего нам нужна функция, которая будет создавать новый сеанс:

```
Clojure/TranscriptHandler/src/server/core.clj7
```

```
(defn create-session []
  (let [snippets (repeatedly promise)
        translations (delay (map translate
                                   (strings->sentences (map deref snippets))))]
    (new-session {:snippets snippets :translations translations})))
```

Мы все еще используем бесконечную ленивую последовательность объектов `promise` для представления входящих фрагментов и отображаем ее в другую последовательность, представляющую перевод, но теперь обе эти последовательности хранятся в сеансе.

Далее нужно изменить `accept-snippet` и `get-translation`, чтобы они выполняли поиск `:snippets` или `:translations` в сеансе:

```
Clojure/TranscriptHandler/src/server/core.clj8
```

```
(defn accept-snippet [session n text]
  (deliver (nth (:snippets session) n) text))

(defn get-translation [session n]
  @(nth @(:translations session) n))
```

Наконец, определим маршруты, которые свяжут эти функции с адресами URI:

```
Clojure/TranscriptHandler/src/server/core.clj9
```

```
(defroutes app-routes
```

⁶ <https://github.com/flatland/useful>

⁷ <http://media.pragprog.com/titles/pb7con/code/Clojure/TranscriptHandler/src/server/core.clj>

⁸ <http://media.pragprog.com/titles/pb7con/code/Clojure/TranscriptHandler/src/server/core.clj>

⁹ <http://media.pragprog.com/titles/pb7con/code/Clojure/TranscriptHandler/src/server/core.clj>

```
(POST "/session/create" []  
  (response (str (create-session))))  
  
(context "/session/:session-id" [session-id]  
  (let [session (get-session (edn/read-string session-id))]  
    (routes  
      (PUT "/snippet/:n" [n :as {:keys [body]}]  
        (accept-snippet session (edn/read-string n) (slurp body))  
        (response "OK"))  
      (GET "/translation/:n" [n]  
        (response (get-translation session (edn/read-string n)))))))
```

Итак, мы получили веб-службу, использующую изменяемые данные в разумных пределах и остающуюся функциональной.

В завершение первого дня

Мы подошли к концу первого дня. Завтра мы познакомимся с агентами и ссылками, двумя другими разновидностями изменяемых переменных в Clojure

Что мы узнали в первый день

Clojure является гибридным функциональным языком, поддерживающим несколько разновидностей изменяемых переменных. Сегодня мы познакомились с наиболее простыми из них – атомами.

- Разница между гибридным функциональным и императивным языками заключается в акцентах.
 - В императивном языке переменные являются изменяемыми по умолчанию, и идиоматический код часто изменяет их значения.
 - В гибридном функциональном языке переменные являются неизменяемыми по умолчанию, и идиоматический код изменяет их, только когда это совершенно необходимо.
- Так как функциональные структуры данных являются сохраняемыми, изменения, выполненные одним потоком, не будут влиять на другой поток, уже получивший ссылку на эту же структуру данных.
- Это позволяет отделить идентичность и состояние, признавая тот факт, что состояние, связанное с идентичностью, в действительности изменяет свое значение с течением времени.

Упражнения к первому дню

Найдите:

- статью Карла Крюкова (Karl Krukow) «Understanding Clojure's PersistentVector Implementation», где описывается, как реализовать сохранную структуру данных, более сложную, чем связанный список;
- продолжение к этой статье, где описывается реализация `PersistentHashMap` на основе «отображения хэша в дерево массивов» («Hash Array Mapped Trie»).

Задания:

- дополните пример `TournamentServer` из раздела «Многопоточная веб-служба с изменяемым состоянием» выше возможностью удаления игроков из списка;
- дополните пример `TranscriptServer` из раздела «Гибридная веб-служба» выше обработкой ситуации, когда фрагмент не поступил в течение ближайших десяти секунд.

День 2: агенты и программная транзакционная память

Вчера мы рассмотрели атомы. Сегодня мы займемся другими разновидностями изменяемых переменных в языке Clojure: агентами и ссылками. Подобно атомам, агенты и ссылки поддерживают использование в конкурентной среде и прекрасно работают в совокупности с сохранными структурами данных для разделения идентичности и состояния. Когда мы перейдем к ссылкам, вы узнаете, что Clojure поддерживает программную транзакционную память (Software Transactional Memory, STM), позволяющую конкурентно изменять переменные без блокировок и с обеспечением их целостности.

Агенты

Агенты напоминают атомы в том, что содержат ссылку на единственное значение, которое можно получить с помощью `deref` или `@`:

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> @my-agent
0
```

Значение агента можно изменить вызовом `send`:

```
user=> (send my-agent inc)
#<Agent@2cadd45e: 1>
user=> @my-agent
1
user=> (send my-agent + 2)
#<Agent@2cadd45e: 1>
user=> @my-agent
3
```

Как и `swap!`, функция `send` принимает другую функцию с набором дополнительных аргументов и вызывает эту функцию, передавая ей текущее значение агента. Возвращаемое значение функции становится новым значением агента.

Разница лишь в том, что `send` возвращает управление немедленно (до того, как значение агента изменится), – функция, переданная в вызов `send`, будет вызвана некогда в будущем. Если функция будет вызвана одновременно несколькими потоками, выполнение функций, переданных в вызов `send`, будет произведено по очереди. То есть никаких повторов выполняться не будет, и, соответственно, эти функции могут производить побочные эффекты.



Вопрос Джо: агенты – это акторы?

Между агентами и акторами (с которыми мы познакомимся в главе 5 «Акторы») в языке Clojure имеется определенное сходство. Однако они имеют отличия, достаточно существенные, чтобы попытка проводить аналогию между ними расценивалась больше как отрицательная, чем полезная:

- агент имеет значение, которое можно получить непосредственно, вызовом `deref`; актор инкапсулирует состояние, но не предоставляет возможности извлечь его непосредственно;
- актор инкапсулирует поведение, отсутствующее у агента, – функцию, реализующую некоторое действие и определяемую отправителем;
- акторы обеспечивают развитую поддержку определения и обработки ошибок, тогда как поддержка обработки ошибок в агентах намного более примитивна;
- акторы могут быть удаленными (`remote`); агенты не поддерживают такую возможность;
- создание акторов может приводить к взаимоблокировкам; создание агентов – никогда.

Ожидание завершения работы агента

При внимательном изучении предыдущего сеанса работы с интерактивной оболочкой REPL можно заметить, что `send` возвращает ссылку на агента. А когда оболочка REPL отображает эту ссылку, она также включает в вывод значение агента – в данном случае число 1:

```
user=> (send my-agent inc)
#<Agent@2cadd45e: 1>
```

Однако в следующий раз вместо числа 3 она вновь выводит число 1:

```
user=> (send my-agent + 2)
#<Agent@2cadd45e: 1>
```

Это объясняется тем, что функция, переданная в вызов `send`, выполняется асинхронно и может еще не завершиться к тому моменту, когда REPL запросит у агента его значение. В случае таких коротких и быстрых операций, как в этом примере, есть вероятность, что к тому моменту, как вы в оболочке REPL попытаетесь получить значение агента, операция уже будет выполнена; но если операция требует больше времени и использует `Thread/sleep`, вы можете увидеть, что это не всегда так:

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> (send my-agent #((Thread/sleep 2000) (inc %)))
#<Agent@224e59d9: 0>
user=> @my-agent
0
user=> @my-agent
1
```

В Clojure имеется функция `await`, которая блокирует доступ к агенту, пока не будут выполнены все операции с агентом, запланированные текущим потоком (существует также функция `await-for`, которая позволяет определить тайм-аут):

```
user=> (def my-agent (agent 0))
#'user/my-agent
user=> (send my-agent #((Thread/sleep 2000) (inc %)))
#<Agent@7f5ff9d0: 0>
user=> (await my-agent)
nil
user=> @my-agent
1
```



Вопрос Джо: что можно сказать о функциях `send-off` и `send-via`?

Помимо функции `send`, агенты поддерживают также функции `send-off` и `send-via`. Единственное отличие состоит в том, что функция `send` выполняет функцию, используя имеющийся пул потоков, функция `send-off` создает новый поток, а `send-via` принимает поток для выполнения в виде аргумента.

Функции `send-off` и `send-via` следует использовать, если функция, которая им передается, может заблокировать выполнение потока или требует значительного времени. Во всем остальном эти функции совершенно идентичны.

Асинхронное изменение имеет очевидные преимущества перед синхронным, особенно когда для вычисления нового значения требуется много времени или процесс вычислений может блокироваться. В связи с этим возникает еще одна сложность, связанная с обработкой ошибок. Далее мы рассмотрим инструменты, имеющиеся в языке Clojure, которые помогут нам в этом.

Обработка ошибок

Подобно атомам, агенты поддерживают функции-валидаторы и функции-наблюдатели. Например, ниже представлен агент, имеющий валидатор, который не допускает, чтобы агент принимал отрицательные значения:

```
user=> (def non-negative (agent 1 :validator (fn [new-val] (>= new-val 0))))
#'user/non-negative
```

Вот что произойдет, если попытаться уменьшать значение агента, пока оно не станет отрицательным:

```
user=> (send non-negative dec)
#<Agent@6257d812: 0>
user=> @non-negative
0
user=> (send non-negative dec)
#<Agent@6257d812: 0>
user=> @non-negative
0
```

Как и предполагалось, агент никогда не получит отрицательного значения. Но что случится, если попытаться использовать агент после того, как возникнет ошибка?

```
user=> (send non-negative inc)
IllegalStateException Invalid reference state clojure.lang.ARef.
validate...
```

```
user=> @non-negative
0
```

Как только в агенте произойдет ошибка, он по умолчанию перейдет в особое, ошибочное состояние, и все попытки изменить его значение будут оканчиваться неудачей. Проверить состояние агента (и выяснить причину) можно с помощью функции `agent-error`, а перевести его в нормальный режим работы – с помощью функции `restart-agent`:

```
user=> (agent-error non-negative)
#<IllegalStateException java.lang.IllegalStateException: Invalid
reference state>
user=> (restart-agent non-negative 0)
0
user=> (agent-error non-negative)
nil
user=> (send non-negative inc)
#<Agent@6257d812: 1>
user=> @non-negative
1
```

По умолчанию агенты создаются с режимом обработки ошибок `:fail`. При желании этот режим можно установить в значение `:continue`, и в этом случае вам не придется вызывать `restart-agent`, чтобы восстановить работоспособность агента. Режим обработки ошибок `:continue` устанавливается по умолчанию, когда определяется *обработчик ошибок* – функция, которая автоматически будет вызываться при появлении ошибок.

Далее мы рассмотрим более практичный пример использования агента.

Журнал в памяти

Я не раз убеждался, насколько полезным для конкурентных программ может быть журнал в памяти – традиционное журналирование может быть слишком тяжеловесным для решения проблем во время отладки из-за необходимости переключения контекстов и выполне-

ния операций ввода/вывода для каждой записи в журнал. Реализация журнала в памяти с применением потоков и блокировок может оказаться весьма сложной задачей, зато на основе агентов реализация того же механизма выглядит почти тривиальной:

Clojure/Logger/src/logger/core.clj¹⁰

```
(def log-entries (agent []))

(defn log [entry]
  (send log-entries conj [(now) entry]))
```

Наш журнал – это агент с именем `log-entries`, инициализируется пустым массивом. Добавление новой записи в этот массив осуществляется функцией `log` с помощью `conj`. Сама запись представляет собой двухэлементный массив – первый элемент хранит время (время вызова `send`, а не время вызова `conj` агентом), а второй элемент хранит собственно сообщение.

Следующий листинг сеанса REPL демонстрирует журнал в действии:

```
logger.core=> (log "Something happened")
#<Agent@bd99597: [[1366822537794 "Something happened"]]>
logger.core=> (log "Something else happened")
#<Agent@bd99597: [[1366822538932 "Something happened"]]>
logger.core=> @log-entries
[[1366822537794 "Something happened"] [1366822538932 "Something else happened"]]
```

В следующем разделе мы рассмотрим еще одну разновидность изменяемых переменных, поддерживаемых в языке Clojure, – ссылки.

Программная транзакционная память

Ссылки представляют еще более сложный механизм, чем атомы и агенты, – программную транзакционную память (Software Transactional Memory, STM). В отличие от атомов и агентов, которые поддерживают изменение только одной переменной, транзакционная память STM позволяет выполнять конкурентные, скоординированные изменения множества переменных, подобно тому как транзакции в базах данных обеспечивают конкурентные и скоординированные изменения множества записей.

Подобно атомам и агентам, ссылки ссылаются на единственное значение, которое можно получить с помощью `deref` или `@`:

¹⁰ <http://media.pragprog.com/titles/pb7con/code/Clojure/Logger/src/logger/core.clj>

```
user=> (def my-ref (ref 0))
#'user/my-ref
user=> @my-ref
0
```

Значение ссылки можно установить с помощью `ref-set`, а эквивалентом `swap!` и `send` является функция `alter`. Однако пользоваться этими функциями немного сложнее, так как для этого недостаточно просто вызывать их:

```
user=> (ref-set my-ref 42)
IllegalStateException No transaction running

user=> (alter my-ref inc)
IllegalStateException No transaction running
```

Изменить значение ссылки можно только в рамках транзакции.

Транзакции

Транзакции STM являются *атомарными*, *согласованными* и *изолированными*.

Атомарность:

С точки зрения кода, выполняющегося в рамках другой транзакции, либо все побочные эффекты транзакции будут иметь место, либо ни один из них.

Согласованность:

Транзакции гарантируют сохранность инвариантов, определяемых посредством валидаторов (по аналогии с атомами и агентами). Если какое-либо из изменений приводит к ошибке при проверке валидаторами, все изменения, произведенные в рамках транзакции, отменяются.

Изолированность:

Несмотря на то что одновременно может выполняться несколько транзакций, эффект конкурентного их выполнения неотличим от эффекта последовательного выполнения.

Эти три характеристики составляют первые три буквы в аббревиатуре ACID (Atomicity (атомарность), Consistency (согласованность), Isolation (изолированность), Durability (надежность)). Недостающая характеристика – Durability (надежность). Дело в том, что данные в транзакционной памяти не сохраняются при пропадании напряжения питания или аварийном завершении работы системы. Если вам требуется обеспечить надежность, используйте базу данных.

Запустить транзакцию можно вызовом `dosync`:

```
user=> (dosync (ref-set my-ref 42))
42
user=> @my-ref
42
user=> (dosync (alter my-ref inc))
43
user=> @my-ref
43
```

Все операции в теле `dosync` выполняются в рамках единой транзакции.



Вопрос Джо: транзакции действительно изолированы?

Полностью изолированные транзакции обычно предпочтительнее в большинстве случаев, но иногда изоляция может оказаться чересчур сильным ограничением. В языке Clojure можно ослабить это требование, используя функцию `commute` вместо `alter`.

Несмотря на то что функция `commute` может выглядеть как весьма привлекательная оптимизация, для правильного ее применения нужно знать множество нюансов, поэтому мы не будем рассматривать ее в этой книге.

Множественные ссылки

Особенно интересной выглядит ситуация, когда в транзакции участвует сразу несколько ссылок (в противном случае мы могли бы просто использовать атом или агента). Классическим примером транзакции служит перевод денег между банковскими счетами – никому не понравится ситуация, когда деньги будут списаны с одного счета, но не будут зачислены на другой счет. Ниже приводится функция, которая либо выполняет обе операции – списания и зачисления, либо ни одну из них:

Clojure/Transfer/src/transfer/core.clj¹¹

```
(defn transfer [from to amount]
  (dosync
    (alter from - amount)
    (alter to + amount)))
```

¹¹ <http://media.pragprog.com/titles/pb7con/code/Clojure/Transfer/src/transfer/core.clj>

Далее приводится пример ее использования:

```
user=> (def checking (ref 1000))
#'user/checking
user=> (def savings (ref 2000))
#'user/savings
user=> (transfer savings checking 100)
1100
user=> @checking
1100
user=> @savings
1900
```

Если механизм STM определит, что в текущей транзакции была предпринята попытка выполнить конфликтующие изменения, он произведет откат транзакции. Это означает, что, как и при изменении атома, транзакции не должны иметь побочных эффектов.

Повтор транзакций

Следуя принципу «лучше один раз увидеть, чем семь раз услышать», давайте посмотрим, происходит ли повтор транзакции, выполнив нагрузочное испытание нашей функции `transfer`:

Clojure/Transfer/src/transfer/core.clj¹²

```
(def attempts (atom 0))
(def transfers (agent 0))

(defn transfer [from to amount]
  (dosync
    > (swap! attempts inc) ; Побочный эффект в транзакции - НЕ ПОСТУПАЙТЕ ТАК!
    > (send transfers inc)
    (alter from - amount)
    (alter to + amount)))
```

Мы нарушили правило, требующее «не иметь побочных эффектов», изменяя значение атома внутри транзакции. В данном случае ничего страшного не произойдет, потому что мы просто собираемся показать, как происходит повтор транзакций, но, пожалуйста, не используйте подобный код в действующих приложениях.

Здесь мы храним два счетчика, в атоме и в агенте. Зачем это нужно, станет понятно чуть ниже.

Вот как выглядит главный метод, осуществляющий нагрузочное тестирование функции `transfer`:

¹² <http://media.pragprog.com/titles/pb7con/code/Clojure/Transfer/src/transfer/core.clj>

Clojure/Transfer/src/transfer/core.clj¹³

```
(def checking (ref 10000))
(def savings (ref 20000))

(defn stress-thread [from to iterations amount]
  (Thread. #(dotimes [_ iterations] (transfer from to amount))))

(defn -main [& args]
  (println "Before: Checking =" @checking " Savings =" @savings)
  (let [t1 (stress-thread checking savings 100 100)
        t2 (stress-thread savings checking 200 100)]
    (.start t1)
    (.start t2)
    (.join t1)
    (.join t2))
  (await transfers)
  (println "Attempts: " @attempts)
  (println "Transfers: " @transfers)
  (println "After: Checking =" @checking " Savings =" @savings))
```

Он запускает два потока. Один поток 100 раз переводит \$100 со счета checking на счет savings, а другой 200 раз переводит \$100 со счета savings на счет checking. Вот что вывела эта программа, когда я попробовал запустить ее у себя:

```
Before: Checking = 10000 Savings = 20000
Attempts: 638
Transfers: 300
After: Checking = 20000 Savings = 10000
```

Отличная новость – конечный результат в точности соответствует ожидаемому, то есть механизм STM благополучно справился со своей задачей, обеспечив правильный конечный результат при наличии нескольких конкурирующих транзакций. Правда, за это пришлось заплатить увеличением числа попыток (в данном случае их было выполнено 338), зато не потребовалось никаких блокировок и нет никакой опасности взаимоблокировок.

Разумеется, это не самый реалистичный пример – два потока, одновременно пытающиеся изменить значения ссылок в цикле, гарантированно будут конфликтовать друг с другом. На практике повторы возникают гораздо реже.

Побочные эффекты и транзакции

Многие из вас наверняка заметили, что счетчик в атоме имеет гораздо большее значение, чем счетчик в агенте, и счетчик в агенте в

¹³ <http://media.pragprog.com/titles/pb7con/code/Clojure/Transfer/src/transfer/core.clj>

точности соответствует числу транзакций. Это вполне объяснимо – агенты поддерживают транзакции.

Если изменение агента выполняется вызовом функции `send` внутри транзакции, эта функция будет вызвана только в случае успеха транзакции. То есть если требуется произвести какой-то побочный эффект в случае успеха транзакции, используйте для этого функцию `send`.



Вопрос Джо: о чем говорит восклицательный знак?

Возможно, вы обратили внимание, что имена некоторых функций оканчиваются восклицательным знаком – о чем это говорит?

В языке Clojure восклицательный знак в именах функций, таких как `swap!` и `reset!`, используется, чтобы показать, что данные функции не являются безопасными с точки зрения транзакций. С другой стороны, мы знаем, что можно совершенно безопасно изменять значение агента в рамках транзакции, поэтому функция, изменяющая состояние агента, носит имя `send`, а не `send!`.

Изменяемое разделяемое состояние

Теперь мы знакомы со всеми тремя механизмами в языке Clojure, обеспечивающими поддержку изменяемого разделяемого состояния. Каждый механизм имеет свою область применения.

Атом позволяет синхронно изменить единственное значение – синхронно, потому что `swap!` возвращает управление уже после фактического изменения. Изменение одного атома не координируется с изменениями в других атомах.

Агент позволяет асинхронно изменить единственное значение – асинхронно, потому что фактическое изменение происходит уже после того, как `send` вернет управление. Изменение одного агента не координируется с изменениями в других агентах.

Ссылки позволяют синхронно и скоординировано изменить несколько значений.

В завершение второго дня

Вот и подошел к концу второй день. Завтра мы рассмотрим ряд более сложных примеров использования изменяемых переменных в Clojure и познакомимся с рекомендациями – какие их разновидности и в каких случаях лучше использовать.

Что мы узнали во второй день

Помимо атомов, в языке Clojure поддерживаются агенты и ссылки:

- атомы обеспечивают возможность *независимого и синхронного* изменения *единственного* значения;
- агенты обеспечивают возможность *независимого и асинхронного* изменения *единственного* значения;
- ссылки обеспечивают возможность *скоординированного и синхронного* изменения *множества* значений.

Упражнения ко второму дню

Найдите:

- презентацию Рича Хикки (Rich Hickey) «Persistent Data Structures and Managed References: Clojure's Approach to Identity and State»;
- презентацию Рича Хикки (Rich Hickey) «Simple Made Easy».

Задания:

- взяв за основу пример TournamentServer из раздела «Многопоточная веб-служба с изменяемым состоянием», реализуйте сервер поддержки соревнований в крестики-нолики, задействовав в нем ссылки и транзакции;
- реализуйте сохранное двоичное дерево поиска, используя списки в роли узлов. Какой объем копирования потребуется выполнить в худшем случае? А в среднем?
- найдите описание древовидной структуры данных Finger Tree и реализуйте на ее основе двоичного дерева поиска. Какой прирост производительности получился в среднем по сравнению с реализацией на основе списков? А в худшем?

День 3: погружение в глубину

К настоящему моменту мы познакомились со всеми компонентами, составляющими «путь Clojure». Сегодня мы рассмотрим более сложные примеры использования этих компонентов и попробуем получить более полное представление о том, в каких случаях следует использовать атомы и транзакционную память.

Решение задачи о философах на основе STM

Для начала вернемся к задаче о пяти философах, с которой мы впервые встретились в главе 2 «Потоки выполнения и блокировки», и реализуем ее решение на языке Clojure с использованием программной транзакционной памяти. Это решение близко напоминает решение на основе условных переменных, представленное в разделе «Условные переменные», но намного проще.

Каждый философ будет представлен отдельной ссылкой, содержащей значение, описывающее текущее состояние философа (:thinking (размышляет) или :eating (ест)). Ссылки хранятся в векторе с именем philosophers:

Clojure/DiningPhilosophersSTM/src/philosophers/core.clj¹⁴

```
(def philosophers (into [] (repeatedly 5 #(ref :thinking))))
```

Каждому философу соответствует свой поток выполнения:

Clojure/DiningPhilosophersSTM/src/philosophers/core.clj¹⁵

```
1 (defn think []
  -   (Thread/sleep (rand 1000)))
  -
  - (defn eat []
  5   (Thread/sleep (rand 1000)))
  -
  - (defn philosopher-thread [n]
  -   (Thread.
  -     #(let [philosopher (philosophers n)
  10         left (philosophers (mod (- n 1) 5))
  -         right (philosophers (mod (+ n 1) 5))]
  -       (while true
  -         (think)
  -         (when (claim-chopsticks philosopher left right)
  15           (eat)
  -           (release-chopsticks philosopher))))))
  -
  - (defn -main [& args]
  -   (let [threads (map philosopher-thread (range 5))]
  20     (doseq [thread threads] (.start thread))
  -     (doseq [thread threads] (.join thread))))
```

¹⁴ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersSTM/src/philosophers/core.clj>

¹⁵ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersSTM/src/philosophers/core.clj>

Как и в решении на Java, каждый поток выполняет бесконечный цикл (строка 12) и переключается между двумя состояниями. Если функции `claim-chopsticks` удастся заполучить палочки слева и справа (строка 14), управляющая конструкция `when` сначала вызывает функцию `eat`, а затем функцию `release-chopsticks`.

Функция `release-chopsticks` имеет довольно прямолинейную реализацию:

```
Clojure/DiningPhilosophersSTM/src/philosophers/core.clj16
```

```
(defn release-chopsticks [philosopher]
  (dosync (ref-set philosopher :thinking)))
```

Она просто запускает транзакцию вызовом `dosync` и устанавливает состояние философа в значение `:thinking` с помощью `ref-set`.

Первая попытка

Наибольший интерес для нас представляет функция `claim-chopsticks` – вот как выглядела первая попытка реализовать ее:

```
(defn claim-chopsticks [philosopher left right]
  (dosync
    (when (and (= @left :thinking) (= @right :thinking))
      (ref-set philosopher :eating))))
```

Как и `release-chopsticks`, она сначала запускает транзакцию. Внутри этой транзакции проверяется состояние философов слева и справа – если они оба находятся в состоянии `:thinking`, вызовом `ref-set` текущий философ переводится в состояние `:eating`. Форма `when` возвращает `nil`, когда условное выражение, переданное ей, получает значение `false`, соответственно, функция `claim-chopsticks` вернет `nil`, если у философа нет возможности взять палочки слева и справа и приступить к еде.

Если попробовать запустить программу с этой реализацией, может показаться, что она прекрасно работает. Однако иногда вы будете видеть, что философы, сидящие рядом, могут есть одновременно, чего не может быть, так как получается, что они оба пользуются одной палочкой. Как такое могло получиться?

Проблема в том, что функция обращается к значениям `left` и `right` с помощью `@`. Программная транзакционная память в Clojure гаран-

¹⁶ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersSTM/src/philosophers/core.clj>

тирует, что никакие две транзакции не смогут выполнить противоречивых изменений в одной и той же ссылке, но в данном случае мы не изменяем ни `left`, ни `right`, а просто проверяем их значения. Некоторая другая транзакция могла бы изменять их, не допуская, чтобы два философа, сидящих по соседству, ели одновременно.

Гарантия от изменения значения

Решить проблему можно, реализовав проверку `left` и `right` с применением `ensure` вместо `@`:

Clojure/DiningPhilosophersSTM/src/philosophers/core.clj¹⁷

```
(defn claim-chopsticks [philosopher left right]
  (dosync
    (when (and (= (ensure left) :thinking) (= (ensure right) :thinking))
      (ref-set philosopher :eating))))
```

Как следует из имени функции `ensure`, она гарантирует, что значение указанной ссылки не сможет быть изменено другой транзакцией. Сравните это решение с прежним решением на основе блокировок. Оно получилось не только проще, но и исключает возможность состояния взаимоблокировки из-за отсутствия блокировок.

В следующем разделе мы рассмотрим альтернативную реализацию, использующую единственный атом вместо множества ссылок и транзакций.

Решение задачи о философах без применения STM

Решение на основе STM не является единственно возможным. Вместо представления философов в виде ссылок и использования транзакций, чтобы гарантировать скоординированное их изменение, можно задействовать единственный атом, представляющий состояние сразу всех философов:

Clojure/DiningPhilosophersAtom/src/philosophers/core.clj¹⁸

```
(def philosophers (atom (into [] (repeat 5 :thinking))))
```

¹⁷ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersSTM/src/philosophers/core.clj>

¹⁸ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersAtom/src/philosophers/core.clj>

Значением атома является вектор состояний. Если философы 0 и 3 едят, например, вектор будет иметь следующий вид:

```
[:eating :thinking :thinking :eating :thinking]
```

Нам нужно внести всего пару изменений в функцию `philosopher-thread`, чтобы она обращалась к каждому философу по его индексу в массиве:

Clojure/DiningPhilosophersAtom/src/philosophers/core.clj¹⁹

```
(defn philosopher-thread [philosopher]
  (Thread.
    #(let [left (mod (- philosopher 1) 5)
          right (mod (+ philosopher 1) 5)]
      (while true
        (think)
        (when (claim-chopsticks! philosopher left right)
          (eat)
          (release-chopsticks! philosopher)))))))
```

Теперь реализация функции `release-chopsticks!` сводится к простому вызову `swap!`, чтобы записать `:thinking` в соответствующий элемент вектора:

Clojure/DiningPhilosophersAtom/src/philosophers/core.clj²⁰

```
(defn release-chopsticks! [philosopher]
  (swap! philosophers assoc philosopher :thinking))
```

Здесь используется функция `assoc`, которую прежде мы применяли только как инструмент для работы с ассоциативными массивами, тем не менее она действует в точном соответствии с нашими ожиданиями:

```
user=> (assoc [:a :a :a :a] 2 :b)
[:a :a :b :a]
```

Наконец, как и прежде, наибольший интерес для нас представляет реализация функции `claim-chopsticks!`:

¹⁹ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersAtom/src/philosophers/core.clj>

²⁰ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersAtom/src/philosophers/core.clj>

Clojure/DiningPhilosophersAtom/src/philosophers/core.clj²¹

```
(defn claim-chopsticks! [philosopher left right]
  (swap! philosophers
    (fn [ps]
      (if (and (= (ps left) :thinking) (= (ps right) :thinking))
        (assoc ps philosopher :eating)
        ps)))
    (= (@philosophers philosopher) :eating))
```

Анонимная функция, что передается функции `swap!`, принимает текущее значение вектора `philosophers` и проверяет состояния соседних философов. Если они оба в текущий момент размышляют, тогда с помощью `assoc` состояние текущего философа изменяется на `:eating`; в противном случае возвращается текущее, неизмененное состояние вектора.

Последняя строка в функции `claim-chopsticks!` проверяет новое значение `philosophers`, чтобы определить, изменилось ли состояние текущего философа в результате вызова `swap!` в `:eating`.

Итак, мы рассмотрели две реализации задачи о философах: одну на основе STM, а другую – на основе атома. Но какой вариант предпочтительнее?

Атомы или STM?

Как было показано в разделе «Изменяемое разделяемое состояние», атомы обеспечивают возможность независимого изменения единственных значений, тогда как ссылки обеспечивают координацию изменения множества значений. Это два совершенно разных набора возможностей, но, как мы увидим в этом разделе, довольно просто взять решение на основе STM, использующее множество ссылок, и преобразовать его в решение на основе единственного атома.

Как оказывается, в этом нет ничего необычного – всегда, когда требуется скоординировать изменение множества значений, можно использовать либо множество ссылок и координировать доступ к ним с помощью транзакций, либо собрать все эти значения в составную структуру данных и сохранить ее в единственном атоме.

Так какой же вариант предпочтительнее?

Во многом ответ на этот вопрос лежит в плоскости личных предпочтений – оба варианта прекрасно работают, поэтому выбирайте тот, который вам ближе и понятнее. В зависимости от особенностей

²¹ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersAtom/src/philosophers/core.clj>

решаемой задачи варианты решений могут иметь разную производительность, поэтому вы можете также провести хронометраж (задействовав комплект своих нагрузочных тестов).

С другой стороны, опытные программисты на Clojure часто считают, что применения атомов достаточно для решения большинства задач, так как функциональная природа языка способствует уменьшению использования изменяемых данных. Как обычно, выбирайте тот вариант, какой вам кажется проще.

Собственная реализация конкуренции

Решение задачи о философах на основе атома работает, но код реализации `claim-chopsticks!` не отличается изяществом. Определенно должна иметься какая-нибудь возможность избежать проверки после вызова `swap!`, чтобы убедиться в возможности получения палочек слева и справа. В идеале хотелось бы иметь версию `swap!`, принимающую предикат и выполняющую изменение значения, только если предикат имеет значение `true`. В этом случае можно было бы переписать `claim-chopsticks!`, как показано ниже:

Clojure/DiningPhilosophersAtom2/src/philosophers/core.clj²²

```
(defn claim-chopsticks! [philosopher left right]
  (swap-when! philosophers
    #(and (= (%1 left) :thinking) (= (%1 right) :thinking))
    assoc philosopher :eating))
```

В языке Clojure нет такой функции, но ничто не мешает нам написать ее:

Clojure/DiningPhilosophersAtom2/src/philosophers/util.clj²³

```
1 (defn swap-when!
  - "Если (pred current-value-of-atom) имеет значение true, атомарно
  - записывает в атом значение (apply f current-value-of-atom args).
  - Имейте в виду, что pred и f могут вызываться много раз и не должны
  - иметь 5 побочных эффектов. Возвращает значение, записанное в атом,
  - если предикат имел значение true, в противном случае возвращает nil."
  - [a pred f & args]
  - (loop []
    - (let [old @a]
```

²² <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersAtom2/src/philosophers/core.clj>

²³ <http://media.pragprog.com/titles/pb7con/code/Clojure/DiningPhilosophersAtom2/src/philosophers/core.clj>

```

10      (if (pred old)
-        (let [new (apply f old args)]
-          (if (compare-and-set! a old new)
-              new
-              (recur)))
15      nil)))

```

Этот листинг содержит довольно много нового для нас. Во-первых, функция включает *строку документирования* (doc-string) – строку между `defn` и списком параметров, – которая описывает ее поведение. Это достойный пример для подражания, и особенно для вспомогательных функций, как эта, предназначенных для многократного использования. Строки документирования можно использовать не только для формирования документации, но и выводить их в интерактивной оболочке REPL:

```

philosophers.core=> (require '[philosophers.util :refer :all])
nil
philosophers.core=> (clojure.repl/doc swap-when!)
-----
philosophers.util/swap-when!
([atom pred f & args])
Если (pred current-value-of-atom) имеет значение true, атомарно
записывает в атом значение (apply f current-value-of-atom args).
Имейте в виду, что pred и f могут вызываться много раз и не должны
иметь побочных эффектов. Возвращает значение, записанное в атом,
если предикат имел значение true, в противном случае возвращает nil.

```

Амперсанд (&) в списке аргументов говорит о том, что `swap-when!` может принимать любое число аргументов (действует подобно трое-точию в Java или звездочке в Ruby). Любые дополнительные аргументы будут помещены в массив, связанный с именем `args`. Мы вызываем функцию `apply`, которая распаковывает свой последний аргумент, чтобы передать его в виде списка аргументов функции `f` (строка 11), – например, ниже демонстрируется эквивалентный способ вызова +:

```

user=> (apply + 1 2 [3 4 5])
15
user=> (+ 1 2 3 4 5)
15

```

Вместо `swap!` наша функция использует более низкоуровневую функцию `compare-and-set!` (строка 12). Она получает атом вместе со старым и новым значениями и атомарно записывает в атом новое значение, но только если его текущее значение равно старому.

Если `compare-and-set!` благополучно изменила значение атома, мы возвращаем новое значение. В противном случае вызовом `recur` (строка 14) возвращаемся обратно в строку 8.



Вопрос Джо: для чего служат макросы `loop` и `recur`?

В отличие от многих функциональных языков, Clojure не поддерживает оптимизацию хвостовой рекурсии, поэтому в идиоматическом коде на Clojure рекурсия используется очень редко. Вместо нее применяются макросы `loop/recur`.

Макрос `loop` определяет позицию в программном коде, куда `recur` сможет выполнять переход (эта пара чем-то напоминает `setjmp()` и `longjmp()` в C/C++). Более подробное описание ищите в документации для Clojure.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения вариантов сочетания в языке Clojure приемов функционального программирования и работы с изменяемыми переменными, поддерживающими конкуренцию.

Что мы узнали в третий день

Функциональная природа Clojure помогает писать код, практически не использующий изменяемых переменных, – для поддержки конкуренции обычно достаточно применения атомов:

- решения на основе STM, использующие множество ссылок и координирующие их изменение посредством транзакций, легко поддаются преобразованию в решения на основе агентов, когда те же ссылки объединяются в общую составную структуру данных, доступную через агента;
- выбор между STM и агентами в значительной степени определяется личными предпочтениями и, возможно, характеристиками производительности;
- применение собственных конструкций поддержки конкуренции часто помогает сделать код чище и проще.

Упражнения к третьему дню

Найдите:

- презентацию Рича Хикки (Rich Hickey) «The Database as a Value» – обратите внимание, как архитектура Datomic позволяет эффективно интерпретировать целую базу данных как единственное значение;²⁴

Задания:

- измените реализацию TournamentServer из упражнений ко второму дню так, чтобы вместо ссылок и транзакций в ней использовались атомы. Сравните решения. Какое из них проще? Какое из них легче читается? Какое имеет более высокую производительность?

В завершение

В языке Clojure реализован весьма прагматичный подход к поддержке конкуренции. Большинство сложностей, связанных с конкурентным программированием, обусловлено наличием совместно используемого изменяемого состояния, поэтому Clojure был создан как функциональный язык, позволяющий писать ссылочно-прозрачный код, в котором отсутствуют побочные эффекты. Но разработчики языка понимали, что при решении многих интереснейших задач трудно обойтись вообще без использования изменяемого состояния, поэтому они добавили в язык несколько разновидностей изменяемых переменных с поддержкой их использования в конкурентном окружении.

Сильные стороны

По вполне понятным причинам, одной из сильных сторон «пути Clojure» является поддержка приемов функционального программирования, которые мы видели в предыдущей главе. Clojure позволяет решать задачи, применяя функциональные подходы, когда это возможно, и отступая от чисто функционального программирования, если это необходимо.

В отличие от переменных в традиционных императивных языках, связывающих вместе идентичность и состояние, сохраненные структуры данных в Clojure дают возможность изменяемым переменным отделить идентичность от состояния. Это устранил целый класс про-

²⁴ <http://www.datomic.com>

блем, характерных для программ, построенных на основе применения блокировок. Опытные программисты на Clojure часто обнаруживают, что идиоматические решения конкурентных задач, что называется, «просто работают».

Слабые стороны

Главный недостаток «пути Clojure» заключается в отсутствии поддержки распределенного программирования. Как следствие в языке отсутствует прямая поддержка нечувствительности к отказам.

Разумеется, благодаря тому что программы на Clojure выполняются под управлением JVM, есть возможность использовать различные сторонние библиотеки поддержки распределенного программирования (в качестве примера можно назвать библиотеку Akka,²⁵ которая, кроме всего прочего, поддерживает модель акторов, обсуждаемую в следующей главе), но применение таких библиотек является отступлением от идиоматики Clojure.

Другие языки

Чисто функциональная природа подразумевает наличие каких-то иных «качеств», однако конкурентный Haskell поддерживает инструменты, весьма похожие на те, что мы видели в этой главе. В частности, в нем имеется полноценная реализация STM, отличное описание которой можно найти в книге Саймона Пейтона Джонса (Simon Peyton Jones) «Beautiful Concurrency».²⁶

Кроме того, для большинства основных языков, не последним из которых является GCC²⁷, имеются свои реализации STM. Однако есть свидетельства, что в императивных языках STM оказывается менее привлекательным решением.²⁸

Напоследок

Разработчикам Clojure удалось соблюсти баланс между функциональным программированием и изменяемым состоянием, позволяющий программистам с опытом использования императивных языков выполнить переход к функциональному программированию с меньшими затруднениями, чем если бы они сразу взялись за чисто

²⁵ <http://blog.darevay.com/2011/06/clojure-and-akka-a-match-made-in/>

²⁶ <http://research.microsoft.com/pubs/74063/beautiful.pdf>

²⁷ <http://gcc.gnu.org/wiki/TransactionalMemory>

²⁸ <http://www.infoq.com/news/2010/05/STM-Dropped>

функциональный язык. При этом в Clojure удалось сохранить большинство преимуществ функционального программирования, в особенности – превосходную поддержку конкуренции.

Такой баланс в значительной степени обусловлен сохранением поддержки изменяемого состояния, но с тщательно продуманной семантикой конкуренции. В следующей главе мы познакомимся с акторами, помогающими окончательно избавиться от изменяемого состояния.



ГЛАВА 5.

Акторы

Актор напоминает автомобиль, взятый в прокат, – вы легко и быстро можете заполучить автомобиль, когда потребуется, а если он сломается, не нужно заботиться о его ремонте; достаточно позвонить в компанию, сдающую автомобили в прокат, и вам тут же пригонят другую машину.

Модель акторов является универсальной моделью конкурентного программирования с особенно широкой областью применения. Она поддерживает обе архитектуры, с разделяемой и распределенной памятью, упрощая создание распределенных приложений и предоставляя особенно мощную поддержку для создания высоконадежных и отказоустойчивых приложений.

Не объекты, а скорее ориентированные на объекты

Функциональное программирование помогает избежать проблем с разделяемым изменяемым состоянием, исключая поддержку изменяемого состояния. Программирование с применением акторов, напротив, сохраняет изменяемое состояние, но исключает совместное его использование.

Акторы напоминают объекты в объектно-ориентированных программах – они инкапсулируют в себе состояние и методы взаимодействий с другими акторами посредством обмена сообщениями. Разница лишь в том, что акторы действуют конкурентно с другими акторами и, в отличие от передачи сообщений в объектно-ориентированном стиле (фактически – вызовов методов), акторы действительно обмениваются сообщениями между собой.

Несмотря на то что модель акторов считается универсальной и может использоваться практически в любых языках, ее чаще считают

неотъемлемой частью языка Erlang.¹ Мы будем рассматривать акторы на примерах их использования в Elixir,² относительно новом языке, программы на котором выполняются под управлением виртуальной машины Erlang (BEAM).

Так же как Clojure (и Erlang), Elixir является гибридным функциональным языком с динамической типизацией. Если вы имеете опыт использования Java или Ruby, вы легко сможете читать код на этом языке. Эта глава не является учебником по языку Elixir (в конце концов, эта книга посвящена проблемам конкуренции, а не языкам программирования), но я познакомлю вас с некоторыми важными его особенностями, которые мы будем использовать далее. Многое из того, что я буду рассказывать, следует просто принимать на веру, если вы пока незнакомы с языком. А желающим познакомиться с ним поближе я рекомендую обратиться к книге «Programming Elixir» [Tho14].

В первый день мы рассмотрим основы модели акторов – создание акторов и прием/отправку сообщений. Во второй день я покажу, как обнаруживать ошибки, и расскажу о философии «и пусть падает», помогающей увеличивать надежность и отказоустойчивость программ с акторами. Наконец, на третий день вы узнаете, как акторы поддерживают распределенное программирование, давая возможность запускать приложения более чем на одном компьютере и благополучно выходить из ситуаций, когда один или более из этих компьютеров выходят из строя.



Вопрос Джо: актер или процесс?

В Erlang и, соответственно, в Elixir акторы называют *процессами*. В большинстве окружений под процессом понимается нечто тягловое, потребляющее массу ресурсов и дорогостоящее в создании. В языке Elixir процессы, напротив, являются очень легковесными – даже более легковесными, чем потоки выполнения в большинстве систем, как с точки зрения потребления ресурсов, так и стоимости запуска. Программы на языке Elixir обычно создают тысячи процессов без каких-либо проблем и не нуждаются в каком-либо эквиваленте пула потоков (см. раздел «Еще раз о создании потоков» в главе 2).

¹ <http://www.erlang.org/>

² <http://elixir-lang.org/>

День 1: сообщения и почтовые ящики

Сегодня мы узнаем, как запускать и останавливать процессы, посылать и принимать сообщения и определять момент завершения процесса.

Наш первый актер

Давайте прямо сейчас попробуем создать простой актер и послать несколько сообщений. Мы сконструируем актер «собеседник», умеющий отвечать простыми фразами в ответ на сообщения.

Роль отправляемых актору сообщений будут играть *кортежи* – последовательности значений. В Elixir кортеж записывается как последовательность в фигурных скобках, например:

```
{:foo, "this", 42}
```

Это кортеж из 3 элементов, первым элементом которого является ключевое слово (ключевые слова в Elixir очень похожи на ключевые слова в Clojure, вплоть до начального двоеточия), вторым элементом является строка, и третьим – целое число.

Ниже приводится реализация нашего актора:

```
Actors/hello_actors/hello_actors.exs3
```

```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age}
years, #{name}")
    end
  end
end
```

Мы разберем этот код более подробно чуть ниже, а пока отмечу, что здесь определяется актер, принимающий три разновидности сообщений и выводящий соответствующую строку в ответ на каждое из них.

Следующий код создает экземпляр актора и посылает ему несколько сообщений:

³ http://media.pragprog.com/titles/pb7con/code/Actors/hello_actors/hello_actors.exs

Actors/hello_actors/hello_actors.exs⁴

```
pid = spawn(&Talker.loop/0)
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:celebrate, "Louie", 16})
sleep(1000)
```

Сначала вызовом функции `spawn` порождается экземпляр актора. Она возвращает *идентификатор процесса*, который тут же связывается с переменной `pid`. Процесс просто выполняет функцию, в данном случае функцию `loop()` из модуля `Talker`, не имеющую аргументов.

Далее вновь созданному актору посылаются три сообщения, и затем вызывается функция `sleep`, чтобы дать актору время на обработку сообщений (использование `sleep()` является не самым лучшим решением – более правильный способ будет описан далее).

В результате этот код должен вывести:

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
```

Теперь, когда вы знаете, как создать актор и как посылать ему сообщения, посмотрим, как же работает вся эта механика.

Почтовые ящики и очереди

Одной из важнейших особенностей программирования акторов является *асинхронная* передача сообщений. Прежде чем попасть в актор, сообщения сначала попадают в *почтовый ящик* (см. рис. 16).



Рис. 16. Сообщения передаются актору через почтовый ящик

⁴ http://media.pragprog.com/titles/pb7con/code/Actors/hello_actors/hello_actors.exs

Это означает *отсутствие тесной связи* с актерами – акторы действуют с собственными скоростями, и передача сообщений актерам не блокирует работу основной программы.

Акторы выполняются конкурентно по отношению к другим актерам, но обработка сообщений выполняется последовательно, в порядке их поступления в почтовый ящик. Переход к следующему сообщению выполняется только после обработки текущего. Беспокоиться о конкуренции нам приходится лишь при отправке сообщений.

Прием сообщений

Обычно актер выполняет бесконечный цикл, вызывая функцию `receive` в ожидании появления сообщения, и затем обрабатывает полученное сообщение. Ниже приводится реализация такого цикла в акторе `Talker`:

Actors/hello_actors/hello_actors.exs⁵

```
def loop do
  receive do
    {:greet, name} -> IO.puts("Hello #{name}")
    {:praise, name} -> IO.puts("#{name}, you're amazing")
    {:celebrate, name, age} -> IO.puts("Here's to another #{age}
years, #{name}")
  end
  loop
end
```

Эта функция реализует бесконечный цикл, рекурсивно вызывая саму себя. Блок `receive` ждет появления сообщения и затем с помощью конструкции сопоставления с шаблоном определяет, как оно должно обрабатываться. Входящие сообщения сравниваются с каждым шаблоном по очереди – если совпадение найдено, переменные в шаблоне (`name` и `age`) связываются с фактическими значениями в сообщении, и выполняется код справа от стрелки (`->`). В каждом случае код выводит текст, сконструированный с помощью механизма интерполяции строк, – он вычисляет выражение внутри `#{...}` и вставляет результат в строку.

Код в примере, что приводился ранее, ждет одну секунду, чтобы дать актору возможность обработать сообщения до того, как программа завершит работу. Это не самое лучшее решение – есть более предпочтительный вариант.

⁵ http://media.pragprog.com/titles/pb7con/code/Actors/hello_actors/hello_actors.exs



Вопрос Джо: не вызовет ли бесконечная рекурсия переполнение стека?

Кто-то из вас наверняка подумал, что функции, выполняющие бесконечную рекурсию, такие как `loop()` в акторе `Talker`, могут вызвать переполнение стека. Могу вас успокоить – подобно многим функциональным языкам (Clojure в этом отношении является исключением – см. врезку «Для чего служат макросы `loop` и `recur`?» в главе 4), Elixir поддерживает *оптимизацию хвостовой рекурсии*. Суть этой оптимизации заключается в замене рекурсивного вызова простой инструкцией перехода, если этот рекурсивный вызов является последней инструкцией, выполняемой в теле функции.

Связывание процессов

Чтобы корректно завершить работу программы, использующей акторы, нам необходимо решить две задачи: сообщить актору, что он должен завершиться после обработки всех сообщений в очереди, и каким-то образом определить, что актор действительно завершился.

Первую задачу можно решить отправкой актору специально сформированного сообщения (подобно тому, как было показано в разделе «Законченная программа» в главе 2):

Actors/hello_actors/hello_actors2.exs⁶

```
defmodule Talker do
  def loop do
    receive do
      {:greet, name} -> IO.puts("Hello #{name}")
      {:praise, name} -> IO.puts("#{name}, you're amazing")
      {:celebrate, name, age} -> IO.puts("Here's to another #{age} years, #{name}")
    > {:shutdown} -> exit(:normal)
  end
end
```

А вторую задачу можно решить, установив флаг `:trap_exit` в значение `true` и породив процесс функцией `spawn_link()` вместо `spawn()`:

⁶ http://media.pragprog.com/titles/pb7con/code/Actors/hello_actors/hello_actors2.exs

Actors/hello_actors/hello_actors2.exs⁷

```
Process.flag(:trap_exit, true)
pid = spawn_link(&Talker.loop/0)
```

В этом случае программа получит извещение (в виде сообщения, сгенерированного системой), когда порожденный ею процесс завершится. Сообщение представляет собой кортеж с тремя элементами:

```
{:EXIT, pid, reason}
```

Все, что нам остается, – отправить сообщение с командой на завершение и дождаться ответа с подтверждением:

Actors/hello_actors/hello_actors2.exs⁸

```
send(pid, {:greet, "Huey"})
send(pid, {:praise, "Dewey"})
send(pid, {:celebrate, "Louie", 16})
➤ send(pid, {:shutdown})

➤ receive do
➤   {:EXIT, ^pid, reason} -> IO.puts("Talker has exited (#{reason})")
➤ end
```

Символ крышки (^) в шаблоне `receive` указывает, что вместо связывания второго элемента кортежа с переменной `pid` требуется сравнить второй элемент со значением `pid`.

Ниже показано, что должна вывести эта новая версия программы:

```
Hello Huey
Dewey, you're amazing
Here's to another 16 years, Louie
Talker has exited (normal)
```

Более подробно о связывании процессов мы поговорим завтра.

Актеры с сохранением состояния

Наш актер `Talker` не хранит информации о своем состоянии. Многие могли бы подумать, что для создания актора с сохранением состояния неизбежно придется использовать изменяемые переменные,

⁷ http://media.pragprog.com/titles/pb7con/code/Actors/hello_actors/hello_actors2.exs

⁸ http://media.pragprog.com/titles/pb7con/code/Actors/hello_actors/hello_actors2.exs

но в действительности для этого достаточно механизма рекурсии. Например, ниже представлен актор со счетчиком, который наращивается при получении нового сообщения:

Actors/counter/counter.ex⁹

```
defmodule Counter do
  def loop(count) do
    receive do
      {:next} ->
        IO.puts("Current count: #{count}")
        loop(count + 1)
    end
  end
end
```

Давайте посмотрим, как действует этот код, воспользовавшись интерактивной оболочкой для языка Elixir, `iex` (Elixir REPL):

```
iex(1)> counter = spawn(Counter, :loop, [1])
#PID<0.47.0>
iex(2)> send(counter, {:next})
Current count: 1
{:next}
iex(3)> send(counter, {:next})
{:next}
Current count: 2
iex(4)> send(counter, {:next})
{:next}
Current count: 3
```

Здесь мы впервые использовали версию `spawn()` с тремя аргументами, которая принимает имя модуля, имя функции в этом модуле и список аргументов, благодаря чему смогли передать начальное значение `count` в `Counter.loop()`. Затем, как и ожидалось, актор начал выводить разные числа после отправки каждого сообщения `{:next}` — мы получили актор с сохранением состояния без использования изменяемых переменных. Более того, актор может безопасно использовать это состояние, потому что сообщения обрабатываются последовательно.

Соккрытие сообщений за фасадом API

Наш актор `Counter` прекрасно работает, но вот пользоваться им не очень удобно. Мы вынуждены помнить, какие аргументы передавать функции `spawn()` и какие сообщения этот актор понимает (`{:next}`,

⁹ <http://media.pragprog.com/titles/pb7con/code/Actors/counter/counter.ex>

:next или {:increment}?). Часто, чтобы облегчить себе жизнь, программисты не вызывают функцию `spawn()` напрямую и не посылают сообщения актору непосредственно, а реализуют отдельные функции, выполняющие эти операции:

Actors/counter/counter.ex¹⁰

```
defmodule Counter do
>   def start(count) do
>     spawn(__MODULE__, :loop, [count])
>   end
>   def next(counter) do
>     send(counter, {:next})
>   end
  def loop(count) do
    receive do
      {:next} ->
        IO.puts("Current count: #{count}")
        loop(count + 1)
    end
  end
end
```

Функция `start()` в этом примере использует *псевдопеременную* `__MODULE__`, хранящую имя текущего модуля. Благодаря новым функциям пользоваться актором намного проще и ниже вероятность допустить ошибку:

```
iex(1)> counter = Counter.start(42)
#PID<0.44.0>
iex(2)> Counter.next(counter)
Current count: 42
{:next}
iex(3)> Counter.next(counter)
{:next}
Current count: 43
```

Мало проку в акторе, который просто выводит свое состояние, поэтому далее мы посмотрим, как реализовать двунаправленное взаимодействие, чтобы два актора могли общаться друг с другом.

Двунаправленное взаимодействие

Как мы уже знаем, сообщения между акторами передаются асинхронно – отправитель не ждет, пока сообщение будет обработано. Но

¹⁰ <http://media.pragprog.com/titles/pb7con/code/Actors/counter/counter.ex>

как быть, если требуется получить ответ? Как поступить, если, к примеру, нужно, чтобы актер `Counter` не выводил значение счетчика, а возвращал его?

Модель акторов не имеет прямой поддержки ответов, но мы можем реализовать отправку ответа, просто включив в сообщение идентификатор процесса-отправителя:

Actors/counter/counter2.ex¹¹

```
defmodule Counter do
  def start(count) do
    spawn(__MODULE__, :loop, [count])
  end
  def next(counter) do
    ref = make_ref()
    send(counter, {:next, self(), ref})
    receive do
    >   {:ok, ^ref, count} -> count
    >   end
  end
  def loop(count) do
    receive do
    >   {:next, sender, ref} ->
    >     send(sender, {:ok, ref, count})
    >     loop(count + 1)
    >   end
  end
end
```

Вместо вывода счетчика эта версия посылает его обратно отправителю оригинального сообщения в виде кортежа с тремя элементами:

```
{:ok, ref, count}
```

Здесь `ref` — это уникальная ссылка, сгенерированная отправителем с помощью `make_ref()`.

Убедимся, что все работает:

```
iex(1)> counter = Counter.start(42)
#PID<0.47.0>
iex(2)> Counter.next(counter)
42
iex(3)> Counter.next(counter)
43
```

¹¹ <http://media.pragprog.com/titles/pb7con/code/Actors/counter/counter2.ex>



Вопрос Джо: почему для ответа используется кортеж?

Новая версия актора `Counter` могла бы отправлять в ответ единственное значение счетчика, не прибегая к использованию кортежа:

```
{:next, sender} ->
  send(sender, count)
```

Хотя это вполне допустимо, в языке Elixir все же принято оформлять сообщения-ответы в виде кортежей, где первый элемент служит признаком успеха. В данном случае в кортеж была включена также уникальная ссылка, сгенерированная клиентом, которая гарантирует правильную идентификацию ответа при наличии нескольких сообщений в клиентском почтовом ящике.

Прежде чем двинуться дальше, внесем еще одно усовершенствование в реализацию актора `Counter` – дадим ему имя, чтобы использовать для ссылки на актор.

Именованние процессов

Чтобы отправить сообщение процессу, необходимо знать его идентификатор. Если этот процесс создан вами, тогда никаких проблем не возникает, но как послать сообщение процессу, который создан не вами?

Эта проблема имеет несколько решений, но самое удобное из них состоит в том, чтобы определить имя для процесса:

```
iex(1)> pid = Counter.start(42)
#PID<0.47.0>
iex(2)> Process.register(pid, :counter)
true
iex(3)> counter = Process.whereis(:counter)
#PID<0.47.0>
iex(4)> Counter.next(counter)
42
```

Здесь мы связали идентификатор процесса с именем, вызвав `Process.register()`, и смогли получить идентификатор процесса по его имени с помощью `Process.whereis()`. Мы можем также получить список всех зарегистрированных имен процессов, вызвав `Process.registered()`:

```
iex(5)> Process.registered
[:kernel_sup, :init, :code_server, :user, :standard_error_sup,
:global_name_server, :application_controller, :file_server_2, :user_drv,
:kernel_safe_sup, :standard_error, :global_group, :error_logger,
:elixir_counter, :counter, :elixir_code_server, :erl_prim_loader,
:elixir_sup, :rex, :inet_db]
```

Как видите, виртуальная машина автоматически регистрирует множество стандартных процессов при запуске. Наконец, что особенно удобно, мы можем передавать функции `send()` имена процессов вместо их идентификаторов:

```
iex(6)> send(:counter, {:next, self(), make_ref()})
{:next, #PID<0.45.0>, #Reference<0.0.0.107>}
```

```
iex(7)> receive do msg -> msg end
{:ok, #Reference<0.0.0.107>, 43}
```

Мы можем воспользоваться всем этим и изменить функции для работы с актором `Counter` так, чтобы нам не нужно было знать идентификатор процесса для их вызова:

Actors/counter/counter3.ex¹²

```
def start(count) do
  pid = spawn(__MODULE__, :loop, [count])
  ➤ Process.register(pid, :counter)
  pid
end
def next do
  ref = make_ref()
  ➤ send(:counter, {:next, self(), ref})
  receive do
    {:ok, ^ref, count} -> count
  end
end
```

Взгляните, насколько удобнее стало пользоваться обновленными функциями:

```
iex(1)> Counter.start(42)
#PID<0.47.0>
iex(2)> Counter.next
42
iex(3)> Counter.next
43
```

И в завершение сегодняшнего дня реализуем параллельную версию функции отображения, похожую на функцию `map` в языке Clojure. Но сначала сделаем небольшое отступление.

¹² <http://media.pragprog.com/titles/pb7con/code/Actors/counter/counter3.ex>

Отступление – функции первого порядка

Как и во всех функциональных языках, функции в языке Elixir являются сущностями первого порядка – их можно связывать с переменными, передавать в аргументах и вообще интерпретировать как самые обычные данные. Ниже приводится пример сеанса `iex`, демонстрирующий передачу анонимной функции в `Enum.map`, чтобы удвоить каждый элемент массива:

```
iex(1)> Enum.map([1, 2, 3, 4], fn(x) -> x * 2 end)
[2, 4, 6, 8]
```

В языке Elixir поддерживается также сокращенный синтаксис `&(...)` определения анонимных функций, похожий на макрос `#(...)` в языке Clojure:

```
iex(2)> Enum.map([1, 2, 3, 4], &(&1 * 2))
[2, 4, 6, 8]
iex(3)> Enum.reduce([1, 2, 3, 4], 0, &(&1 + &2))
10
```

Функции, связанные с переменными, можно вызывать с помощью оператора точки `(.)`:

```
iex(4)> double = &(&1 * 2)
#Function<erl_eval.6.80484245>
iex(5)> double.(3)
6
```

И наконец, можно писать функции, возвращающие другие функции:

```
iex(6)> twice = fn(fun) -> fn(x) -> fun.(fun.(x)) end end
#Function<erl_eval.6.80484245>
iex(7)> twice.(double).(3)
12
```

Теперь мы знаем все, что необходимо для создания параллельной версии `map()`.

Параллельная версия `map()`

Как мы уже знаем, в языке Elixir имеется функция `map()`, которую можно использовать для применения функций к коллекциям, но у нее есть один недостаток – она действует последовательно. Ниже приводится альтернативная реализация, которая обрабатывает элементы коллекции параллельно:

Actors/parallel/parallel.ex¹³

```
defmodule Parallel do
  def map(collection, fun) do
    parent = self()

    processes = Enum.map(collection, fn(e) ->
      spawn_link(fn() ->
        send(parent, {self(), fun.(e)})
      end)
    end)
    Enum.map(processes, fn(pid) ->
      receive do
        {^pid, result} -> result
      end
    end)
  end
end
```

Эта функция выполняется в два этапа. На первом этапе создается по одному процессу для каждого элемента коллекции (для коллекции с 1000 элементов будет создано 1000 процессов). В каждом процессе будет вызвана функция `fun` для соответствующего элемента, результат – отправлен родительскому процессу. На втором этапе родительский процесс ждет получения всех результатов.

Посмотрим, как она работает:

```
iex(1)> slow_double = fn(x) -> :timer.sleep(1000); x * 2 end
#Function<6.80484245 in :erl_eval.expr/5>
iex(2)> :timer.tc(fn() -> Enum.map([1, 2, 3, 4], slow_double) end)
{4003414, [2, 4, 6, 8]}
iex(3)> :timer.tc(fn() -> Parallel.map([1, 2, 3, 4], slow_double) end)
{1001131, [2, 4, 6, 8]}
```

Здесь используется функция `:timer.tc()`, измеряющая время выполнения указанной функции и возвращающая кортеж, содержащий время и возвращаемое значение. Как видите, последовательная версия выполняется чуть дольше четырех секунд, а параллельная – одну секунду.

В завершение первого дня

Мы подошли к концу первого дня. Завтра мы увидим, как модель акторов помогает в обработке ошибок и обеспечении отказоустойчивости.

¹³ <http://media.pragprog.com/titles/pb7con/code/Actors/parallel/parallel.ex>

Что мы узнали в первый день

Акторы (процессы) выполняются конкурентно, без использования разделяемого состояния, и взаимодействуют асинхронно, путем отправки сообщений в почтовый ящик. Сегодня мы узнали, как:

- создать новый процесс с помощью `spawn()`;
- отправить сообщение процессу с помощью `send()`;
- пользоваться сопоставлением с шаблонами для обработки сообщений;
- связать два процесса и принимать уведомления, когда один из них завершается;
- реализовать двунаправленный, синхронный обмен сообщениями на основе стандартного, асинхронного механизма обмена сообщениями;
- зарегистрировать имя для процесса.

Упражнения к первому дню

Найдите:

- документацию с описанием языка Elixir;
- видеоролик, где Эрик Мейер (Erik Meijer) и Клеменс Шиперски (Clemens Szyperski) рассказывают Карлу Хьюитту (Carl Hewitt) о модели акторов на конференции Lang.NEXT 2012.

Задания:

- измерьте стоимость создания процесса в виртуальной машине Erlang. Сравните ее со стоимостью создания потока выполнения в виртуальной машине Java;
- измерьте стоимость использования параллельной версии функции `map`, созданной нами, и сравните ее со стоимостью использования стандартной, последовательной версии `map`. Когда имеет смысл использовать параллельную версию, а когда лучше предпочесть последовательную?
- напишите параллельную версию функции `reduce`, взяв за основу параллельную версию `map`.

День 2: обработка ошибок и отказоустойчивость

Как было показано в разделе «Надежные программы для непредсказуемого мира» (глава 1), одним из ключевых преимуществ конкурен-

ции является возможность писать высоконадежные, отказоустойчивые программы. Сегодня мы узнаем, как акторы позволяют писать такие программы.

Но прежде воспользуемся знаниями, полученными вчера, и напишем чуть более сложный пример, который будет служить нам основой для дальнейшей сегодняшней дискуссии.

Актор кэширования

Мы создадим кэш для хранения веб-страниц. Клиент сможет добавлять страницы в кэш, указывая адреса URL и содержимое страниц, извлекать страницы из кэша по их адресам URL, а также определять объем информации в байтах, хранящейся в кэше.

В качестве основы кэша будет использоваться словарь, отображающий адреса URL в содержимое страниц. Подобно ассоциативным массивам в Clojure, словари в языке Elixir являются сохраняемыми (persistent), ассоциативными структурами данных:

```
iex(1)> d = HashDict.new
#HashDict<[]>
iex(2)> d1 = Dict.put(d, :a, "A value for a")
#HashDict<[a: "A value for a"]>
iex(3)> d2 = Dict.put(d1, :b, "A value for b")
#HashDict<[a: "A value for a", b: "A value for b"]>
iex(4)> d2[:a]
"A value for a"
```

В этом примере мы создали словарь вызовом `HashDict.new`, добавили в него несколько элементов с помощью `Dict.put(dict, key, value)` и выполнили поиск, используя синтаксис `dict[key]`.

Ниже приводится реализация кэша, в которой используются функции, представленные выше:

Actors/cache/cache.ex¹⁴

```
1 defmodule Cache do
-   def loop(pages, size) do
-       receive do
-           {:put, url, page} ->
2               new_pages = Dict.put(pages, url, page)
3               new_size = size + byte_size(page)
4               loop(new_pages, new_size)
-           {:get, sender, ref, url} ->
5               send(sender, {:ok, ref, pages[url]})
-       end
-   end
end
```

¹⁴ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache.ex>

```

10      loop(pages, size)
-      { :size, sender, ref } ->
-      send(sender, { :ok, ref, size })
-      loop(pages, size)
-      { :terminate } -> # Завершить работу - не выполнять рекурсию
15  end
-  end
-  end

```

Она поддерживает два компонента состояния, `pages` и `size`. Первый компонент – словарь, отображающий адреса URL в содержимое страниц; второй – целое число, соответствующее числу байтов, хранящихся в текущий момент в кэше (обновляется функцией `byte_size()` в строке 6).

Как и прежде, мы не будем заставлять пользователей запоминать порядок запуска актора и передачи сообщений, спрятав детали за фасадом API. Начнем с функции `start_link()`:

Actors/cache/cache.ex¹⁵

```

def start_link do
  pid = spawn_link(__MODULE__, :loop, [HashDict.new, 0])
  Process.register(pid, :cache)
  pid
end

```

Она передает пустой словарь и нулевой размер функции `loop()` и регистрирует порожденный процесс под именем `:cache`. Далее демонстрируются функции `put()`, `get()`, `size()` и `terminate()`:

Actors/cache/cache.ex¹⁶

```

def put(url, page) do
  send(:cache, { :put, url, page })
end

def get(url) do
  ref = make_ref()
  send(:cache, { :get, self(), ref, url })
  receive do
    { :ok, ^ref, page } -> page
  end
end

def size do

```

¹⁵ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache.ex>

¹⁶ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache.ex>


```

    ref = make_ref()
    send(:cache, {:size, self(), ref})
    receive do
      {:ok, ^ref, s} -> s
    end
  end
end

def terminate do
  send(:cache, {:terminate})
end

```

Функции `put()` и `terminate()` просто принимают свои аргументы, упаковывают их в кортеж и отправляют в виде сообщения. Функции `get()` и `size()` чуть более сложные из-за того, что им приходится ждать ответа. Они отправляют уникальные ссылки, используя шаблон, что был продемонстрирован вчера.

Следующий листинг демонстрирует актор в действии:

```

iex(1)> Cache.start_link
#PID<0.47.0>
iex(2)> Cache.put("google.com", "Welcome to Google ...")
{:put, "google.com", "Welcome to Google ..."}
iex(3)> Cache.get("google.com")
"Welcome to Google ..."
iex(4)> Cache.size()
21

```

Пока все выглядит неплохо — мы можем добавить страницу в кэш, извлечь ее из кэша и узнать объем кэша.

Но что случится, если передать актору недопустимый аргумент? Например, если попытаться добавить пустую (`nil`) страницу?

```

iex(5)> Cache.put("paulbutcher.com", nil)
{:put, "paulbutcher.com", nil}
iex(6)>
=ERROR REPORT==== 22-Aug-2013::16:18:41 ===
Error in process <0.47.0> with exit value: {badarg, [{erlang, byte_size, [nil], []} ...

** (EXIT from #PID<0.47.0>) {:badarg, [{:erlang, :byte_size, [nil], []}, ...

```

Ничего удивительного, особенно если учесть, что мы не предусмотрели ни строчки кода для проверки аргументов. В большинстве языков есть только один путь решить эту проблему — добавить код, проверяющий аргументы и сообщающий об ошибке, если обнаруживается какое-либо недопустимое значение. Однако в языке Elixir

имеется другой путь – выделение обработки ошибок в отдельный процесс-диспетчер. Этот, несомненно, простой шаг может приобретать самые разные формы и позволяет писать ясный, легко сопровождаемый и надежный код.

Чтобы написать такой процесс-диспетчер, нужно детальнее разобраться с организацией связи между процессами.

Определение момента отказа

В разделе «Связывание процессов» выше мы использовали функцию `spawn_link()` для организации связи между двумя процессами, чтобы иметь возможность определить момент завершения одного из процессов. Связь процессов – это одна из важнейших концепций в программировании на Elixir. Рассмотрим ее подробнее.

Связи позволяют определить момент аварийного завершения

Установить связь между двумя процессами можно в любой момент, вызвав функцию `Process.link()`. Следующий актор поможет нам исследовать поведение связей:

Actors/links/links.ex¹⁷

```
defmodule LinkTest do
  def loop do
    receive do
      {:exit_because, reason} -> exit(reason)
      {:link_to, pid} -> Process.link(pid)
      {:EXIT, pid, reason} -> IO.puts("#{inspect(pid)} exited because
#{reason}")
    end
  loop
  end
end
```

Создадим пару экземпляров этого актора, свяжем их между собой и посмотрим, что получится, когда один из них неожиданно завершится:

```
iex(1)> pid1 = spawn(&LinkTest.loop/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
```

¹⁷ <http://media.pragprog.com/titles/pb7con/code/Actors/links/links.ex>

```
iex(3)> send(pid1, { :link_to, pid2 })
{:link_to, #PID<0.49.0>}
iex(4)> send(pid2, { :exit_because, :bad_thing_happened })
{:exit_because, :bad_thing_happened}
```

Здесь сначала создаются два экземпляра актора, а соответствующие им идентификаторы процессов сохраняются в переменных `pid1` и `pid2`. Затем устанавливается связь из `pid1` в `pid2`. Наконец, процессу `pid2` сообщается, что он должен завершиться.

Обратите внимание, что процесс `pid1` не вывел сообщение, описывающее причину завершения процесса `pid2`. Это объясняется тем, что мы не установили флаг `:trap_exit`. Однако связь между процессами устанавливается и без этого флага, в чем можно убедиться с помощью функции `Process.info()`:

```
iex(5)> Process.info(pid2, :status)
nil
iex(6)> Process.info(pid1, :status)
nil
```

Как видите, завершились *оба* процесса, а не только `pid2`. Как исправить эту проблему, я расскажу чуть ниже, но сначала проведем еще один эксперимент.

Связи являются двунаправленными

Если повторить предыдущий эксперимент, но на этот раз послать сообщение с требованием завершиться процессу `pid1`, можно будет наблюдать в точности то же поведение – оба процесса завершатся:

```
iex(1)> pid1 = spawn(&LinkTest.loop/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, { :link_to, pid2 })
{:link_to, #PID<0.49.0>}
iex(4)> send(pid1, { :exit_because, :another_bad_thing_happened })
{:exit_because, :another_bad_thing_happened}
iex(5)> Process.info(pid1, :status)
nil
iex(6)> Process.info(pid2, :status)
nil
```

Это объясняется тем, что связи между процессами носят *двунаправленный* характер. Создание связи из процесса `pid1` с процессом `pid2` приводит также к созданию связи в обратном направлении – если один из двух процессов завершится, завершится и второй.

Нормальное завершение

Наконец, посмотрим, что получится, если один из связанных процессов завершится нормальным образом (определяется специальной причиной `:normal`):

```
iex(1)> pid1 = spawn(&LinkTest.loop/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, {:link_to, pid2})
{:link_to, #PID<0.49.0>}
iex(4)> send(pid2, {:exit_because, :normal})
{:exit_because, :normal}
iex(5)> Process.info(pid2, :status)
nil
iex(6)> Process.info(pid1, :status)
{:status, :waiting}
```

Итак, нормальное завершение одного из связанных процессов не приводит к завершению другого.

Системные процессы

Мы можем позволить одному процессу перехватывать событие завершения другого процесса установкой флага `:trap_exit`. На жаргоне это называется превращением в *системный процесс*:

Actors/links/links.ex¹⁸

```
def loop_system do
  Process.flag(:trap_exit, true)
  loop
end
```

Посмотрим, что получилось:

```
iex(1)> pid1 = spawn(&LinkTest.loop_system/0)
#PID<0.47.0>
iex(2)> pid2 = spawn(&LinkTest.loop/0)
#PID<0.49.0>
iex(3)> send(pid1, {:link_to, pid2})
{:link_to, #PID<0.49.0>}
iex(4)> send(pid2, {:exit_because, :yet_another_bad_thing_happened})
{:exit_because, :yet_another_bad_thing_happened}
#PID<0.49.0> exited because yet_another_bad_thing_happened
iex(5)> Process.info(pid2, :status)
```

¹⁸ <http://media.pragprog.com/titles/pb7con/code/Actors/links/links.ex>

```
nil
iex(6)> Process.info(pid1, :status)
{:status, :waiting}
```

На этот раз процесс `pid1` был запущен вызовом функции `loop_system()`. Теперь он не только получает уведомление о завершении процесса `pid2` (и выводит сообщение), но и продолжает работу.

Слежение за работой процессов

Теперь мы знаем достаточно, чтобы реализовать процесс-диспетчер — системный процесс, который следит за работой одного или более обычных процессов и выполняет соответствующие действия, когда они терпят неудачу.

Ниже приводится реализация процесса-диспетчера для актора кэша, созданного выше, которая просто перезапускает подопечный актор кэша, если (и когда) он неожиданно завершается:

Actors/cache/cache.ex¹⁹

```
defmodule CacheSupervisor do
  def start do
    spawn(__MODULE__, :loop_system, [])
  end
  def loop do
    pid = Cache.start_link
    receive do
      {:EXIT, ^pid, :normal} ->
        IO.puts("Cache exited normally")
        :ok
      {:EXIT, ^pid, reason} ->
        IO.puts("Cache failed with reason #{inspect reason} - restarting it")
        loop
    end
  end
  def loop_system do
    Process.flag(:trap_exit, true)
    loop
  end
end
```

На запуске этот актор отмечает себя как системный процесс и затем входит в функцию `loop()`, которая вызывает `Cache.loop()` и блокируется до момента, пока порожденный процесс не завершится. Если прекращение работы произойдет обычным путем, процесс-диспетчер

¹⁹ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache.ex>

тоже завершится (вернув `:ok`), но если работа наблюдаемого процесса прекратится по какой-либо другой причине, `loop()` рекурсивно вызовет себя и вновь породит процесс кэша.

Теперь вместо экземпляра `Cache` мы должны создать экземпляра `CacheSupervisor`, который от нашего имени сам создаст экземпляра `Cache`:

```
iex(1)> CacheSupervisor.start
#PID<0.47.0>
iex(2)> Cache.put("google.com", "Welcome to Google ...")
{:put, "google.com", "Welcome to Google ..."}
iex(3)> Cache.size
21
```

Если произойдет аварийное завершение актора `Cache`, он автоматически будет перезапущен:

```
iex(4)> Cache.put("paulbutcher.com", nil)
{:put, "paulbutcher.com", nil}
Cache failed with reason {:badarg, [{:erlang, :byte_size, [nil],
[]}, ...
iex(5)>
=ERROR REPORT==== 22-Aug-2013::17:49:24 ===
Error in process <0.48.0> with exit value: {badarg, [{erlang,byte_size,
[nil],[]}, ...
iex(5)> Cache.size
0
iex(6)> Cache.put("google.com", "Welcome to Google ...")
{:put, "google.com", "Welcome to Google ..."}
iex(7)> Cache.get("google.com")
"Welcome to Google ..."
```

Разумеется, в случае аварии теряется все, что хранилось в кэше, но теперь мы хотя бы можем продолжать использовать кэш.

Тайм-ауты

Автоматический перезапуск кэша — большое достижение, но этот прием не является универсальной панацеей. Если два процесса одновременно пошлют сообщения кэшу, мы можем увидеть, к примеру, следующую последовательность действий:

1. Процесс 1 отправляет сообщение `:put` кэшу.
2. Процесс 2 отправляет сообщение `:get` кэшу.
3. Кэш завершается аварийно при попытке обработать сообщение от процесса 1.

4. Процесс-диспетчер перезапускает кэш, но сообщение от процесса 2 теряется.
5. Процесс 2 оказывается заблокированным в блоке `receive`, ожидая ответа, который никогда не придет.

Мы можем решить эту проблему, определив тайм-аут для `receive`, добавив предложение `after`. Ниже приводится измененная версия функции `get()` (то же изменение необходимо внести в функцию `size()`):

Actors/cache/cache2.ex²⁰

```
def get(url) do
  ref = make_ref()
  send(:cache, {:get, self(), ref, url})
  receive do
    {ok, ^ref, page} -> page
  > after 1000 -> nil
  end
end
```



Вопрос Джо: гарантируется ли доставка сообщений?

Проблема потери клиентского сообщения, которую мы только что наблюдали, является лишь частным случаем более общей проблемы – какие гарантии, касающиеся доставки сообщений, может дать Elixir?

В общем случае:

- сообщения гарантированно доставляются, когда все работает штатно;
- если произойдет что-то необычное, у вас есть возможность узнать об этом (предполагается, что главная программа осуществляет мониторинг интересующего ее процесса).

Именно вторая гарантия является основой поддержки отказоустойчивого кода в Elixir.

Ядро ошибки

Тони Хоар (Tony Hoare) очень метко сказал:²¹

Существует два пути проектирования программного обеспечения: первый – сделать его настолько простым, чтобы в нем

²⁰ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache2.ex>

²¹ <http://zoo.cs.yale.edu/classes/cs422/2011/bib/hoare81emperor.pdf>

явно отсутствовали любые недостатки, и второй – сделать его настолько сложным, чтобы в нем не было видно явных недостатков.

Акторы естественным образом поддерживают создание отказоустойчивого кода, подчеркивая данное наблюдение, за счет использования шаблона проектирования «ядро ошибки» (error-kernel pattern).

Ядром ошибки в программной системе называется та ее часть, которую следует подкорректировать, чтобы обеспечить нормальное функционирование всей системы. Следует стремиться сделать эту область настолько маленькой и простой, «чтобы в ней явно отсутствовали любые недостатки».

Ядром ошибки в программе на основе акторов является процесс-диспетчер верхнего уровня. Он управляет подчиненными ему процессами – запускает, останавливает и перезапускает их по мере необходимости.

Каждый модуль программы имеет свое ядро ошибки – часть модуля, которая должна функционировать безупречно, чтобы обеспечить правильную работу всего модуля. Подмодули также имеют свои ядра ошибок и т. д. В результате получается своеобразная иерархия, в которой число рискованных операций ширится вниз, в направлении низкоуровневых акторов, как показано на рис. 17.

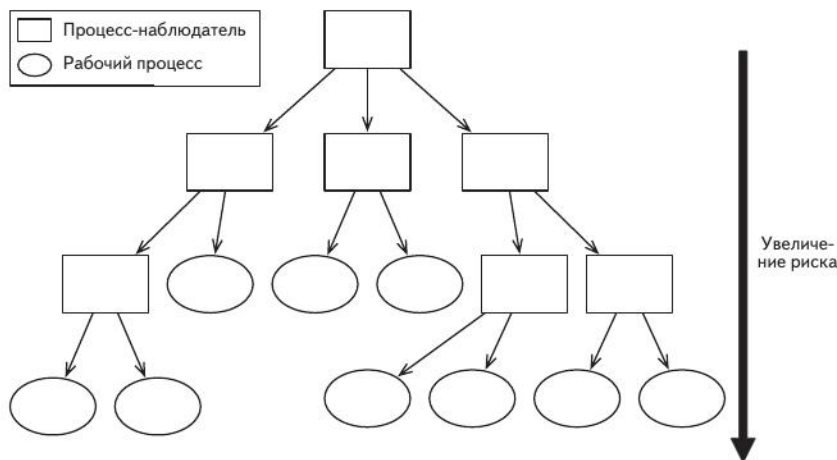


Рис. 17. Иерархия ядер ошибок

С шаблоном проектирования «ядро ошибки» тесно связана весьма неоднозначная тема защитного программирования.

И пусть падает!

Защитное программирование – это один из способов достижения отказоустойчивости за счет попытки предусмотреть все возможные ошибки. Представьте, к примеру, что вы пишете метод, принимающий строку и возвращающий `true`, если она содержит только символы верхнего регистра, и `false` в противном случае. Ниже приводится одна из возможных реализаций:

```
def all_upper?(s) do
  String.upcase(s) == s
end
```

Это решение выглядит вполне разумным, но если по какой-то причине методу будет передано значение `nil`, он потерпит крах (то есть «упадет»). Обнаружив это, некоторые разработчики могли бы изменить метод, как показано ниже:

```
defmodule Upper do
  def all_upper?(s) do
    cond do
      nil?(s) -> false
      true -> String.upcase(s) == s
    end
  end
end
```

Теперь метод перестанет «падать», получая значение `nil`, но что, если передать ему что-то другое, бессмысленное в данном контексте (например, ключевое слово)? И вообще, что может означать попытка вызвать эту функцию со значением `nil`? Исправив метод, мы замаскируем код, являющийся источником этой ошибки, то есть мы будем оставаться в неведении до тех пор, пока не столкнемся с этой ошибкой в более отдаленном будущем.

Акторы помогают избегать приемов защитного программирования и проводить в жизнь философию «и пусть падает» (*let it crash*), согласно которой ответственность за решение всех проблем перекладывается на плечи процесса-диспетчера. Это дает массу преимуществ, включая следующие:

- код становится проще и понятнее, в нем четко видна граница между «рабочим кодом» и кодом, обеспечивающим отказоустойчивость;
- акторы отделены друг от друга и не имеют общего состояния, поэтому вероятность отказа из-за взаимовлияния акторов све-

дена к минимуму. В частности, актер-диспетчер не может вызывать аварию в подчиненном ему акторе, потому что он всего лишь наблюдает за его работоспособностью.

- помимо восстановления после аварии, процесс-диспетчер может также зафиксировать ошибку в файле журнала, чтобы позднее можно было разобраться с ней и исправить.

Несмотря на всю необычность философии «и пусть падает», она неоднократно доказывала свою состоятельность, особенно в комплексе с шаблоном проектирования «Ядро ошибки». Некоторые системы, построенные на основе этой философии, обеспечивают высокую доступность в 99.9999999% случаев (то есть девять девяток после запятой – см. «Programming Erlang: Software for a Concurrent World» [Arm13]).

В завершение второго дня

В первый день мы познакомились с основами модели акторов, а во второй увидели, как она способствует повышению отказоустойчивости. Завтра мы посмотрим, как модель акторов помогает в создании распределенных приложений.

Что мы узнали во второй день

В языке Elixir поддерживается возможность обнаружения отказов за счет связывания процессов, что позволяет создавать процессы-диспетчеры:

- связи между процессами являются двунаправленными – если процесс *A* связать с процессом *B*, процесс *B* тоже окажется связанным с процессом *A*;
- ошибки распространяются по связям – если один из двух связанных процессов завершится аварийно, это также приведет к завершению второго процесса;
- если процесс отмечен как системный, в случае аварии второго процесса он не завершится, а получит сообщение `:EXIT`.

Упражнения ко второму дню

Найдите:

- документацию с описанием `Process.monitor()` – чем механизм мониторинга процессов отличается от механизма связей? Когда предпочтительнее использовать мониторинг, а когда связи?

- как действуют исключения в Elixir? Когда предпочтительнее обрабатывать исключения вместо использования процесса-диспетчера и следования философии «и пусть падает»?

Задания:

- сообщения, не соответствующие ни одному шаблону в блоке `receive`, остаются в почтовом ящике процесса. Используя этот факт вместе с возможностью установки тайм-аута, реализуйте *почтовый ящик с приоритетами*, в котором высокоприоритетные сообщения будут обрабатываться раньше низкоприоритетных, даже если последние будут отправлены раньше;
- создайте версию кэша, опираясь на реализацию, представленную в разделе «Актор кэширования», которая будет распределять кэш по нескольким акторам с помощью функции хэширования. Создайте процесс-диспетчер, который будет запускать несколько акторов кэширования и распределять входящие сообщения между ними. Какие шаги должен предпринять процесс-диспетчер при аварийном завершении одного из рабочих акторов?

День 3: распределенные приложения

Все программные компоненты, что мы создавали до настоящего момента, выполнялись на единственном компьютере, однако одним из основных преимуществ модели акторов является поддержка распределенных приложений – отправить сообщение удаленному актору, выполняющемуся на другом компьютере, так же просто, как локальному.

Но, прежде чем перейти к обсуждению приемов создания распределенных приложений, поговорим немного о наиболее существенной причине, почему стоит использовать язык Elixir, – о библиотеке ОТР.

ОТР

Вчера и позавчера мы писали весь код вручную, на «низкоуровневом» языке Elixir. Это прекрасный способ понять, что творится за кулисами, однако создавать все процессы – рабочие и диспетчеры – вручную слишком утомительно и чревато ошибками. Едва ли мне удастся удивить вас, если скажу, что существует библиотека, помо-

гающая автоматизировать все эти операции. Эта библиотека называется OTP.



Вопрос Джо: что означает название OTP?

Аббревиатуры часто начинают жить своей жизнью. Так, название IBM является аббревиатурой от «International Business Machines», но для большинства название IBM – это просто IBM: аббревиатура превратилась в имя собственное. Аналогично никто больше не вспоминает, что название BBC происходит от «British Broadcasting Corporation». Название OTP тоже давно уже никто не расшифровывает как «Open Telecom Platform».

Язык Erlang (и, соответственно, Elixir) зародился в телекоммуникационной среде, и многие алгоритмы на этом языке были включены в OTP. Но лишь малая часть из них имеет хоть какое-то отношение к телекоммуникации, поэтому название OTP – это просто OTP.

Прежде чем мы начнем знакомиться с примерами использования OTP, сделаем небольшое отступление и посмотрим, как в языке Elixir взаимодействуют функции и операция сопоставления с шаблоном.

Функции и сопоставление с шаблоном

До сих пор мы видели лишь примеры сопоставления с шаблоном в блоке `receive`, но в языке Elixir эта операция используется повсюду. В частности, каждый раз, вызывая функцию, вы фактически выполняете операцию сопоставления с шаблоном. Следующая простая функция демонстрирует эту особенность языка:

Actors/patterns/patterns.ex²²

```
defmodule Patterns do
  def foo({x, y}) do
    IO.puts("Got a pair, first element #{x}, second #{y}")
  end
end
```

Здесь определяется функция, которая принимает единственный аргумент и сопоставляет его с шаблоном `{x, y}`. Если передать функции соответствующую шаблону пару значений, первое из них будет связано с переменной `x`, а второе – с переменной `y`:

²² <http://media.pragprog.com/titles/pb7con/code/Actors/patterns/patterns.ex>

```
iex(1)> Patterns.foo({:a, 42})
Got a pair, first element a, second 42
:ok
```

Если передать функции аргумент, не соответствующий шаблону, мы получим ошибку:

```
iex(2)> Patterns.foo("something else")
** (FunctionClauseError) no function clause matching in Patterns.foo/1
    patterns.ex:3: Patterns.foo("something else")
    erl_eval.erl:569: :erl_eval.do_apply/6
    src/elixir.erl:147: :elixir.eval_forms/3
```

В функцию можно включить любое число разных определений (или *предложений*):

Actors/patterns/patterns.ex²³

```
def foo({x, y, z}) do
  IO.puts("Got a triple: #{x}, #{y}, #{z}")
end
```

Когда происходит вызов функции, она выполняет совпавшее предложение:

```
iex(2)> Patterns.foo({:a, 42, "yahoo"})
Got a triple: a, 42, yahoo
:ok
iex(3)> Patterns.foo({:x, :y})
Got a pair, first element x, second y
:ok
```

А теперь посмотрим, как эту особенность можно использовать для реализации сервера на основе библиотеки OTP.

Повторная реализация кэша с использованием GenServer

Первое, с чем мы познакомимся в библиотеке OTP, – это *GenServer*, программный компонент, в терминологии Erlang называемый *поведением* (behaviour), позволяющий автоматизировать создание актора с сохранением состояния. Мы будем использовать его для повторной реализации кэша, созданного вчера.

Для тех, кому правописание *behaviour* покажется странным²⁴, поясню, что поведения (behaviours) переключались в Elixir из Erlang, а

²³ <http://media.pragprog.com/titles/pb7con/code/Actors/patterns/patterns.ex>

²⁴ В американском английском это слово пишется как *behavior*. – Прим. перев.

Erlang создавался в Великобритании, поэтому в Elixir и Erlang используются правила правописания, действующие в британском английском.

Поведения очень похожи на интерфейсы в Java – они определяют комплекс функций. Модуль, реализующий некоторое поведение, должен указывать его имя с помощью `use`:

Actors/cache/cache3.ex²⁵

```
defmodule Cache do
  > use GenServer.Behaviour

  def handle_cast({:put, url, page}, {pages, size}) do
    new_pages = Dict.put(pages, url, page)
    new_size = size + byte_size(page)
    {:noreply, {new_pages, new_size}}
  end

  def handle_call({:get, url}, _from, {pages, size}) do
    {:reply, pages[url], {pages, size}}
  end

  def handle_call({:size}, _from, {pages, size}) do
    {:reply, size, {pages, size}}
  end
end
```

Эта версия `Cache` явно указывает, что реализует `GenServer.Behaviour`, и предоставляет собственную реализацию двух функций, `handle_cast()` и `handle_call()`.

Первая из них, `handle_cast()`, обрабатывает сообщения, не требующие ответа. Она принимает два аргумента: в первом передается сообщение, а во втором – текущее состояние актора. Возвращаемое значение представляет собой пару в форме `{:noreply, new_state}`. В нашем случае функция `handle_cast()` имеет единственное предложение, обрабатывающее сообщения `:put`.

Вторая функция, `handle_call()`, обрабатывает сообщения, требующие ответа. Она принимает три аргумента: сообщение, ссылку на отправителя и текущее состояние. Возвращаемое значение является кортежем с тремя элементами `{:reply, reply_value, new_state}`. В нашем случае функция `handle_call()` имеет два предложения, одно обрабатывает сообщения `:get`, а второе – сообщения `:size`. Обратите внимание, что в Elixir, так же как в Clojure, имена неиспользуемых переменных начинаются с символа подчеркивания (`_`) – здесь объявлена одна такая переменная `_from`.

²⁵ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache3.ex>

Как и в предыдущей реализации, мы скрываем тонкости работы с кэшем за фасадом API, чтобы пользователям не приходилось запоминать, как выполнять инициализацию или как отправлять сообщения:

Actors/cache/cache3.ex²⁶

```
def start_link do
  :gen_server.start_link({:local, :cache}, __MODULE__, {HashDict.new, 0},
    [])
end

def put(url, page) do
  :gen_server.cast(:cache, {:put, url, page})
end

def get(url) do
  :gen_server.call(:cache, {:get, url})
end

def size do
  :gen_server.call(:cache, {:size})
end
```

Вместо `spawn_link()` мы используем `:gen_server.start_link()`. Отправка сообщений, не требующих ответа, осуществляется с помощью `:gen_server.cast()`, а отправка сообщений, требующих ответа, осуществляется с помощью `:gen_server.call()`.

Как действует этот кэш, будет показано ниже, но перед этим давайте посмотрим, как с помощью OTP создать процесс-диспетчер.

Реализация процесса-диспетчера на основе OTP

Ниже приводится исходный код процесса-диспетчера, реализующего поведение `Supervisor` из библиотеки OTP:

Actors/cache/cache3.ex²⁷

```
defmodule CacheSupervisor do
  def init(_args) do
    workers = [worker(Cache, [])]
    supervise(workers, strategy: :one_for_one)
  end
end
```

²⁶ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache3.ex>

²⁷ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache3.ex>

Как можно догадаться по имени функции `init()`, она вызывается на этапе запуска. Эта функция принимает единственный аргумент (неиспользуемый в данном случае), создает несколько рабочих процессов и объявляет их подчиненными (то есть контролируруемыми). В нашем случае мы создаем единственный рабочий процесс `Cache` и устанавливаем связь с ним с указанием стратегии перезапуска «один из-за одного».



Вопрос Джо: что такое «стратегия перезапуска»?

Поведение `Supervisor` в библиотеке OTP поддерживает несколько разных стратегий перезапуска, из которых наиболее часто используются «один из-за одного» и «все из-за одного».

Стратегии перезапуска определяют, как процесс-диспетчер с несколькими подчиненными процессами будет перезапускать аварийные рабочие процессы. Если аварию потерпит единственный рабочий процесс, а для процесса-диспетчера определена стратегия «все из-за одного», тогда диспетчер остановит все рабочие процессы и снова запустит их (даже те, которые работали в штатном режиме). При использовании стратегии «один из-за одного», напротив, перезапущен будет только аварийный процесс.

Поддерживаются и другие стратегии перезапуска, но этих двух достаточно для подавляющего большинства случаев.

Как обычно, определим API для наших пользователей:

```
Actors/cache/cache3.ex28
```

```
def start_link do
  :supervisor.start_link(__MODULE__, [])
end
```

Остановитесь на минутку и опробуйте все это в интерактивной оболочке, как мы делали это вчера (я не буду приводить листинг сеанса, потому что он очень похож на то, что вы уже видели).

Узлы

Всякий раз, создавая экземпляры виртуальной машины Erlang, мы создаем *узел* (node). До сих пор мы создавали единственный узел, а теперь посмотрим, как создать и соединить друг с другом множество узлов.

²⁸ <http://media.pragprog.com/titles/pb7con/code/Actors/cache/cache3.ex>



Вопрос Джо: что еще может предложить OTP?

Как можно видеть в предыдущем примере, библиотека OTP помогает избежать необходимости писать некоторый шаблонный код, но она может предложить еще много чего. В примерах выше это не видно, но серверы и процессы-диспетчеры, реализованные на основе OTP, обладают намного более широкими функциональными возможностями, чем наши простые версии, созданные вчера. Помимо всего прочего, они поддерживают:

Улучшенную логику перезапуска. Простой процесс-диспетчер, который мы написали для своих нужд, реализует чересчур упрощенный подход к перезапуску рабочего процесса – если последний завершится аварийно, он будет перезапущен. Если рабочий процесс начнет терпеть крах немедленно после запуска, наш процесс-диспетчер будет просто перезапускать его снова и снова, до бесконечности. Реализация диспетчера в OTP, напротив, ограничивает частоту перезапуска и в случае ее превышения сам завершается аварийно.

Отладку и журналирование. Сервер OTP можно запустить с разными параметрами, управляющими журналированием и отладкой, что может очень пригодиться на этапе разработки.

«Горячую» замену кода. Сервер OTP можно обновлять динамически, не останавливая всю систему целиком.

Многое, многое другое. Управление версиями, восстановление после отказа, автоматизированное масштабирование...

Мы больше не будем говорить об этих особенностях, но отмечу, что они являются веской причиной предпочесть библиотеку OTP в большинстве случаев.

Соединение узлов

Чтобы соединить один узел с другим, им обоим требуется присвоить имена. Для этого следует запустить виртуальную машину Erlang с параметром `--name` или `--sname`. Например, мой MacBook Pro имеет IP-адрес 10.99.1.50. Если я выполню на нем команду `iex --sname node1@10.99.1.50 --cookie yumyum` (параметр `--cookie` описывается во врезке «Как управлять своим кластером?» ниже), то увижу имя узла в строке приглашения к вводу:

```
iex (node1@10.99.1.50) 1> Node.self
:"node1@10.99.1.50"
```

```
iex(node1@10.99.1.50) 2> Node.list  
[]
```

У узла можно запросить его имя, вызовом `Node.self()`, или список известных ему других узлов, вызовом `Node.list()`. Прямо сейчас этот список пуст – давайте посмотрим, как заполнить его. Если запустить другой экземпляр виртуальной машины Erlang на другом компьютере с IP-адресом 10.99.1.92 командой `iex --sname node2@10.99.1.92 --cookie yumyum`, можно будет соединить его с моим MacBook Pro вызовом `Node.connect()`:

```
iex(node1@10.99.1.50) 3> Node.connect(:"node2@10.99.1.92")  
true  
iex(node1@10.99.1.50) 4> Node.list  
[:"node2@10.99.1.92"]
```

Соединения между узлами имеют двунаправленную природу – мой другой компьютер теперь также знает о существовании моего же MacBook Pro:

```
iex(node2@10.99.1.92) 1> Node.list  
[:"node1@10.99.1.50"]
```



Вопрос Джо: а если у меня всего один компьютер?

Если у вас под рукой только один компьютер и вы желаете поэкспериментировать с распределенными приложениями, у вас на выбор есть несколько вариантов:

- воспользоваться виртуальными машинами;
- воспользоваться услугами Amazon EC2 или подобных облачных веб-служб;
- запустить несколько узлов на одном компьютере. Даже при том, что это не самая реалистичная ситуация, данный вариант является самым простым и позволяет обходить брандмауэры и другие проблемы, связанные с настройками сетей, которые могут мешать при экспериментах со множеством узлов на разных компьютерах.

Удаленное выполнение

Теперь, когда у нас имеются два соединенных узла, один из них может вызывать код другого:

```
iex(node1@10.99.1.50)5> whoami = fn() -> IO.puts(Node.self) end
#Function<20.80484245 in :erl_eval.expr/5>
iex(node1@10.99.1.50)6> Node.spawn(:"node2@10.99.1.92", whoami)
#PID<8242.50.0>
node2@10.99.1.92
```

Эти обманчиво простые строки кода демонстрируют удивительную мощь – не только первый узел может вызывать код второго узла, но и вывод, производимый вторым узлом, появляется в первом. Это объясняется тем, что процесс наследует лидера группы процесса, породившего его и (кроме всего прочего) определяющего, где должен появляться вывод, производимый функцией `IO.puts()`. Вот как много чего находится за кулисами!



Вопрос Джо: как управлять своим кластером?

Системы, позволяющие одному компьютеру выполнять код на другом компьютере, отличаются необычайной вычислительной мощностью. Но всякий мощный инструмент также таит в себе опасность. В частности, размышляя об управлении кластером, необходимо с особым вниманием отнестись к вопросам обеспечения безопасности. Аргумент `--cookie`, который мы указывали при запуске `iex`, как раз и является одним из элементов защиты – узел Erlang будет принимать соединения только от узлов, запущенных с тем же значением параметра `--cookie`. Существуют и другие подходы к обеспечению безопасности кластеров Erlang, такие как туннелирование соединений через SSL.

Однако безопасность – это не единственное, о чем придется задумываться. В предыдущем примере я использовал IP-адреса вместо имен узлов, чтобы гарантировать его работоспособность в большинстве сетей (я же не знаю, как настроена сеть у вас). Но для промышленных приложений возможно (даже не возможно, а наверняка) это будет не лучший выбор.

Архитектура кластеров является весьма сложной темой и далеко выходит за рамки данной книги. Поэтому обязательно почитайте документацию по этой теме, прежде чем передавать в эксплуатацию свой кластер.

Обмен сообщениями с удаленными узлами

Как вы уже наверняка поняли, актер из одного узла способен обмениваться сообщениями с актором из другого узла. Для демонстрации породим экземпляр актора `Counter`, созданного выше (см. раздел «Акторы с сохранением состояния») на одном узле:


```
iex(node2@10.99.1.92)1> pid = spawn(Counter, :loop, [42])
#PID<0.51.0>
iex(node2@10.99.1.92)2> :global.register_name(:counter, pid)
:yes
```

После запуска зарегистрируем его вызовом функции `:global.register_name()`, которая действует подобно `Process.register()`, за исключением того, что она создает глобальное имя кластера.

Благодаря этому мы сможем теперь вызвать `:global.whereis_name()` на другом узле, чтобы получить идентификатор процесса, необходимый для отправки сообщений:

```
iex(node1@10.99.1.50)1> Node.connect(:"node2@10.99.1.92")
true
iex(node1@10.99.1.50)2> pid = :global.whereis_name(:counter)
#PID<7856.51.0>
iex(node1@10.99.1.50)3> send(pid, {:next})
{:next}
iex(node1@10.99.1.50)4> send(pid, {:next})
{:next}
```

В результате выполнения этих операций на первом узле мы увидим:

```
Current count: 42
Current count: 43
```

Обратите внимание, что вывод снова появляется на узле, где был запущен актор, породивший его.

Распределенный счетчик слов

Обсуждение акторов и языка Elixir я предлагаю закончить созданием распределенной версии примера счетчика слов в Википедии, который мы рассматривали в предыдущих главах. Как и предыдущие решения, это также будет способно использовать несколько ядер. Но, в отличие от предыдущих версий, оно сможет выполняться одновременно на нескольких компьютерах и восстанавливаться после сбоев.

На рис. 18 приводится диаграмма архитектуры будущего решения.

Наше решение будет реализовано на основе акторов трех разных типов: `Parser`, несколько акторов `Counter` и один актор `Accumulator`. Актор `Parser` отвечает за парсинг дампа Википедии и разбиение его на страницы, акторы `Counter` будут выполнять подсчет слов в страницах, и актор `Accumulator` будет вести общий подсчет слов для всех страниц.

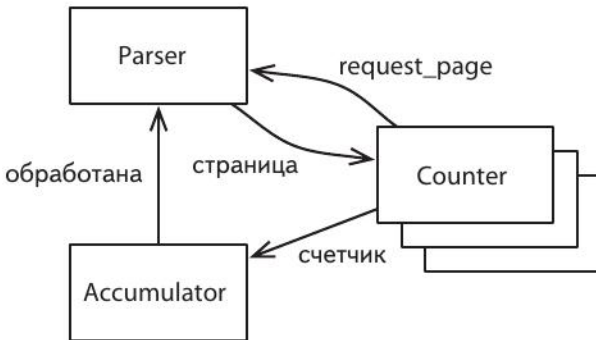


Рис. 18. Архитектура распределенной версии счетчика слов

Обработка начинается с обращения актора `Counter` к актору `Parser` за страницей. Получив страницу, `Counter` подсчитывает слова, содержащиеся в ней, и отправляет результаты актору `Accumulator`. Наконец, актор `Accumulator` сообщает актору `Parser`, что страница обработана.

Чуть ниже я объясню, почему выбрана такая схема, а пока познакомимся с реализациями акторов, начав с `Counter`.

Подсчет слов

Наш модуль `Counter` реализует простой актор без сохранения состояния, который принимает страницы от актора `Parser` и возвращает результаты актору `Accumulator`. Ниже приводится полная его реализация:

Actors/word_count/lib/counter.ex²⁹

```

1 defmodule Counter do
  - use GenServer.Behaviour
  - def start_link do
  -   :gen_server.start_link(__MODULE__, nil, [])
  5 end
  - def deliver_page(pid, ref, page) do
  -   :gen_server.cast(pid, {:deliver_page, ref, page})
  - end
  -
  10 def init(_args) do
  -   Parser.request_page(self())
  -   {:ok, nil}
  - end

```

²⁹ http://media.pragprog.com/titles/pb7con/code/Actors/word_count/lib/counter.ex

```

-
15  def handle_cast({:deliver_page, ref, page}, state) do
-    Parser.request_page(self())
-
-    words = String.split(page)
-    counts = Enum.reduce(words, HashDict.new, fn(word, counts) ->
20    Dict.update(counts, word, 1, &(&1 + 1))
-    end)
-    Accumulator.deliver_counts(ref, counts)
-    {:noreply, state}
-  end
25 end

```

Реализация выполнена по шаблону, обычному для серверов ОТР, – общедоступный API (в данном случае функции `start_link()` и `deliver_page()`), за которым следуют функция инициализации (`init()`) и обработчики событий (`handle_cast()`).

Каждый актор `Counter` начинает работу с вызова `Parser.request_page()` в ходе инициализации (строка 11).

Получив страницу, `Counter` тут же запрашивает следующую (строка 16 – это делается в первую очередь для минимизации задержек). Затем он подсчитывает слова в текущей странице, конструируя словарь с именем `counts` (строки 18–21). Наконец, результаты отправляются актору `Accumulator` вместе со ссылкой (`ref`), полученной со страницей.

Далее приводится реализация `CounterSupervisor`, процесса-диспетчера, который создает и контролирует несколько акторов `Counter`:

Actors/word_count/lib/counter.ex³⁰

```

defmodule CounterSupervisor do
  use Supervisor.Behaviour
  def start_link(num_counters) do
    :supervisor.start_link(__MODULE__, num_counters)
  end
  def init(num_counters) do
    workers = Enum.map(1..num_counters, fn(n) ->
      worker(Counter, [], id: "counter#{n}")
    end)
    supervise(workers, strategy: :one_for_one)
  end
end

```

Функция `CounterSupervisor.init()` принимает аргумент с числом счетчиков, которое используется для создания списка `workers`.

³⁰ http://media.pragprog.com/titles/pb7con/code/Actors/word_count/lib/counter.ex

Отметьте, что каждый рабочий процесс должен иметь уникальный идентификатор `id`, что обеспечивается путем отображения на диапазон `1..num_counters`.

Подсчет общих сумм

Актор `Accumulator` имеет хранимое состояние, включающее два элемента: `totals`, словарь, где накапливаются счетчики слов, и `processed_pages`, множество ссылок всех обработанных страниц.

Actors/word_count/lib/accumulator.ex³¹

```
1 defmodule Accumulator do
-   use GenServer.Behaviour
-
-   def start_link do
5       :gen_server.start_link({:global, :wc_accumulator}, __MODULE__,
        {HashDict.new, HashSet.new}, [])
-   end
-
-   def deliver_counts(ref, counts) do
10      :gen_server.cast({:global, :wc_accumulator}, {:deliver_counts,
ref, counts})
-   end
-
-   def handle_cast({:deliver_counts, ref, counts}, {totals, proce
ssed_pages}) do
-       if Set.member?(processed_pages, ref) do
15         {:noreply, {totals, processed_pages}}
-       else
-         new_totals = Dict.merge(totals, counts, fn(_k, v1, v2) -> v1 + v2 end)
-         new_processed_pages = Set.put(processed_pages, ref)
-         Parser.processed(ref)
20         {:noreply, {new_totals, new_processed_pages}}
-       end
-   end
- end
```

Передав `{:global, wc_accumulator}` функции `:gen_server.start_link()` (строка 5), мы определили глобальное имя для аккумулятора. Это имя можно использовать непосредственно для передачи сообщений вызовом `:gen_server.cast()` (строка 10).

Когда аккумулятор получает множество счетчиков, он сначала проверяет, не была ли данная страница обработана прежде (вскоре я поясню, почему это важно и почему одна страница может быть обработана дважды). Если страница еще не обрабатывалась, получен-

³¹ http://media.pragprog.com/titles/pb7con/code/Actors/word_count/lib/accumulator.ex

ные результаты включаются в словарь `totals` вызовом `Dict.merge()`, ссылка на обработанную страницу добавляется в `processed_pages` вызовом `Set.put()`, и актору `Parser` посылается сообщение, что страница обработана.

Парсинг и устойчивость к ошибкам

Актор `Parser` является самым сложным в нашем приложении, поэтому разберем его реализацию по частям. Сначала рассмотрим общедоступный интерфейс:

Actors/word_count/lib/parser.ex³²

```
defmodule Parser do
  use GenServer.Behaviour

  def start_link(filename) do
    :gen_server.start_link({:global, :wc_parser}, __MODULE__, filename, [])
  end

  def request_page(pid) do
    :gen_server.cast({:global, :wc_parser}, {:request_page, pid})
  end

  def processed(ref) do
    :gen_server.cast({:global, :wc_parser}, {:processed, ref})
  end
end
```

Как и `Accumulator`, актор `Parser` присваивает себе глобальное имя на этапе инициализации. Он поддерживает две операции – функцию `request_page()`, которая вызывается актором `Counter` для получения страницы, и функцию `processed()`, которая вызывается актором `Accumulator`, чтобы сообщить, что страница была успешно обработана.

Ниже приводится реализация обработчиков сообщений для этих двух операций:

Actors/word_count/lib/parser.ex³³

```
def init(filename) do
  xml_parser = Pages.start_link(filename)
  {:ok, {ListDict.new, xml_parser}}
end

def handle_cast({:request_page, pid}, {pending, xml_parser}) do
```

³² http://media.pragprog.com/titles/pb7con/code/Actors/word_count/lib/parser.ex

³³ http://media.pragprog.com/titles/pb7con/code/Actors/word_count/lib/parser.ex

```
new_pending = deliver_page(pid, pending, Pages.next(xml_parser))
{:noreply, {new_pending, xml_parser}}
end

def handle_cast({:processed, ref}, {pending, xml_parser}) do
  new_pending = Dict.delete(pending, ref)
  {:noreply, {new_pending, xml_parser}}
end
```

Parser поддерживает состояние с двумя элементами: `pending` – список типа `ListDict` для хранения ссылок страниц, отправленных акторам `Counter`, но пока необработанных, и `xml_parser` – актор, использующий библиотеку `xmerl` для языка Erlang³⁴ для парсинга дампа Википедии (я не буду приводить реализацию парсинга здесь – полные исходные тексты вы найдете в загружаемом пакете примеров для книги).

Обработка сообщения `:processed` заключается в простом удалении ссылки на обработанную страницу из `pending`. Обработка сообщения `:request_page` заключается в извлечении следующей страницы из дампа и передаче ее функции `deliver_page()`:

Actors/word_count/lib/parser.ex³⁵

```
defp deliver_page(pid, pending, page) when nil?(page) do
  if Enum.empty?(pending) do
    pending # Ничего делать не надо
  else
    {ref, prev_page} = List.last(pending)
    Counter.deliver_page(pid, ref, prev_page)
    Dict.put(Dict.delete(pending, ref), ref, prev_page)
  end
end

defp deliver_page(pid, pending, page) do
  ref = make_ref()
  Counter.deliver_page(pid, ref, page)
  Dict.put(pending, ref, page)
end
```

Функция `deliver_page()` использует одну особенность языка Elixir, которую мы еще не видели, – *предложение-ограничитель* (guard clause), начинающееся с `when` в первом предложении в функции `deliver_page()`. Предложение-ограничитель – это выражение,

³⁴ <http://www.erlang.org/doc/apps/xmerl/>

³⁵ http://media.pragprog.com/titles/pb7con/code/Actors/word_count/lib/parser.ex

возвращающее логическое значение. Предложение в функции, включающее такой ограничитель, совпадет, только если выражение вернет значение `true`.

Рассмотрим сначала ситуацию, когда `page` имеет значение, отличное от `nil`. В этом случае создается новая уникальная ссылка, вызовом `make_ref()`, страница со ссылкой отправляется счетчику, пришедшему запрос, и затем эта страница добавляется в словарь `pending`.

Если `page` имеет значение `nil`, это указывает на то, что парсер XML завершил парсинг дампа Википедии. В этом случае из `pending` удаляется самая старая запись, и соответствующая страница отправляется актору `Counter`, после чего она снова добавляется в `pending`, чтобы сделать ее самой новой записью.

Зачем это нужно? Разумеется, каждая ожидающая страница будет в конечном итоге обработана. Но что мы получаем, отправляя страницу для обработки другому актору `Counter`?

Большой выигрыш

Мы получаем отказоустойчивость. Если актор `Counter` неожиданно завершит свою работу, или соединение будет разорвано из-за неполадок в сети, или компьютер, где выполняется актор `Counter`, будет выключен, мы просто отправим страницу другому актору `Counter`. Так как каждая страница сопровождается уникальной ссылкой, мы точно будем знать, какие страницы уже обработаны, и сможем не допустить двойного подсчета.

Чтобы убедиться в правоте сказанного выше, попробуйте запустить кластер. На одном компьютере запустите `Parser` и `Accumulator`. На одном или нескольких других компьютерах запустите несколько акторов `Counter`. Если теперь отключить сетевой кабель от одного компьютера, где выполняются счетчики, или остановить одну виртуальную машину с ними, остальные счетчики продолжат обработку страниц, включая и те, что обрабатывались на том компьютере.

Это – яркий пример преимуществ конкуренции и распределенной обработки. Эта программа выдержит отказ аппаратных средств, который сделал бы невозможным продолжение работы нормальной, последовательной или многопоточной программы.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения модели акторов.

Что мы узнали в третий день

Язык Elixir позволяет создавать кластеры узлов. Актор из одного узла может отправлять сообщения актерам в других узлах точно так же, как если бы все они выполнялись на одном компьютере. Это обстоятельство помогает нам не только создавать распределенные системы, выполняющиеся на нескольких компьютерах, но и противостоять всевозможным отказам.

Упражнения к третьему дню

Найдите:

- презентацию Джо Армстронга (Joe Armstrong) «Systems That Run Forever Self-Heal and Scale», представленную им на конференции Lambda Jam;
- охарактеризуйте приложение на основе ОТР. Почему его можно было бы более точно описать как компонент?
- во всех наших примерах акторы утрачивали информацию о своем состоянии в случае завершения. Как в языке Elixir можно было бы обеспечить сохранение состояния?

Задания:

- отказоустойчивая программа-счетчик, созданная нами сегодня, способна противостоять отказам на компьютерах, где выполняются акторы `Counter`, но она не переживет отказ компьютера, где выполняется актор `Parser` или `Accumulator`. Напишите версию, которая была бы способна противостоять отказам в любом акторе или узле.

В завершение

Алан Кей (Alan Kay), создатель языка Smalltalk и отец объектно-ориентированного программирования, сказал следующее по поводу объектной ориентированности:³⁶

Я прошу прощения, что давным-давно выдумал термин «объекты» для этой темы, потому что большинство людей сосредоточились на менее важной идее.

Более важная идея – это «обмен сообщениями»... В японском языке есть маленькое слово «мэ», означающее «то, что лежит между». Возможно, что наиболее близкий эквивалент – «про-

³⁶ <http://c2.com/cgi/wiki?AlanKayOnMessaging>

межуточный». Для создания мощной и способной развиваться системы нужно сосредоточиться больше на проектировании взаимодействий между модулями, чем на внутренних для модулей свойствах и поведении.

Это высказывание как нельзя лучше подчеркивает суть программирования с применением акторов – акторы можно рассматривать как логическое продолжение объектно-ориентированного программирования в конкурентном мире. Фактически акторы можно рассматривать скорее не как объекты, а как программные компоненты, ориентированные на объекты, поддерживающие обмен сообщениями и инкапсуляцию.

Сильные стороны

Акторы обладают многими чертами, делающими их идеальными для решения широкого круга задач конкуренции.

Обмен сообщениями и инкапсуляция

Акторы не имеют совместно используемого состояния и хотя выполняются конкурентно с другими акторами, сами они действуют последовательно. Это означает, что нам нужно лишь позаботиться о конкурентной обработке потоков сообщений между акторами.

Это огромное благо для разработчиков. Акторы можно тестировать по отдельности, и если наши тесты достаточно точно будут имитировать типы сообщений и порядок их доставки, мы сможем пребывать в полной уверенности, что тестируемые акторы действуют именно так, как ожидалось. А столкнувшись с какой-нибудь ошибкой, связанной с конкуренцией, мы будем точно знать, где ее искать – в потоках сообщений между акторами.

Отказоустойчивость

Программы, построенные на основе акторов, изначально являются отказоустойчивыми. Это позволяет выпускать не только высоконадежные программы, но и писать более краткий и ясный код (следуя философии «и пусть падает»).

Распределенные приложения

Поддержка акторами обеих архитектур, с разделяемой и распределенной памятью, дает множество важных преимуществ.

Во-первых, это позволяет создавать масштабируемые программы, способные решать задачи практически любого размера. Мы не ограничены рамками одной системы.

Во-вторых, это дает возможность решать задачи, естественной чертой которых является географическая разбросанность. Модель акторов является отличным выбором для программ, разные части которых должны размещаться в разных географических регионах.

И в-третьих, распределенность является ключом к созданию высоконадежных и отказоустойчивых систем.

Слабые стороны

Хотя программы на основе акторов проще в отладке, если сравнивать их с программами, использующими потоки выполнения и блокировки, акторы все еще восприимчивы к таким проблемам, как взаимоблокировки, плюс еще несколько, уникальных для акторов (таких как переполнение почтовых ящиков акторов).

Подобно модели на основе потоков и блокировок, модель акторов не имеет прямой поддержки параллелизма. Параллельные решения приходится конструировать из конкурентных строительных блоков, увеличивая фактор недетерминизма. А так как акторы не имеют разделяемого состояния и могут взаимодействовать друг с другом только посредством сообщений, они плохо подходят для реализации параллелизма, когда требуется более тонкое разделение.

Другие языки

Как и любая хорошая идея, модель акторов не нова – первые ее описания появились в 1970-х, наиболее примечательной из которых является работа Карла Хьюитта (Carl Hewitt). Но наибольшую популярность акторы получили благодаря языку Erlang. Например, создатель Erlang, Джо Армстронг (Joe Armstrong), является родоначальником философии «и пусть падает».

Практически для всех основных языков программирования имеются библиотеки поддержки акторов; в частности, библиотека Akka позволяет использовать акторы в Java и любых других языках, основанных на JVM.³⁷ Если вам интересно узнать больше о библиотеке Akka, обращайтесь к дополнительной главе,³⁸ где описывается применение акторов в языке Scala.

³⁷ <http://akka.io>

³⁸ http://media.pragprog.com/titles/pb7con/Bonus_Chapter.pdf

Напоследок

Модель акторов имеет самую широкую область применения среди других – она не только дает поддержку конкуренции, но также позволяет создавать распределенные приложения, упрощает поиск ошибок и обеспечивает высокую отказоустойчивость. Благодаря этим качествам она отлично подходит для решения самых разных задач программирования в современном распределенном мире.

В следующей главе мы познакомимся с моделью взаимодействия последовательных процессов. Несмотря на то что модель CSP (Communicating Sequential Processes – взаимодействие последовательных процессов) имеет некоторое сходство с акторами, она в большей степени описывает организацию каналов для взаимодействий, а не создание сущностей, между которыми осуществляются взаимодействия, что ставит ее особняком.



ГЛАВА 6.

Взаимодействие последовательных процессов

Если вы так же увлекаетесь автомобилями, как я, значит, вы так же легко концентрируетесь на машине, забывая о дорогах, по которым вы ездите. Это так здорово, обсуждать с кем-то преимущества турбонаддува перед атмосферными карбюраторами или размещения двигателя в середине кузова перед размещением спереди, забывая, что самое важное в автомобиле не имеет никакого отношения ко всему этому. Ответы на вопросы «как проехать» и «как быстрее добраться» лежат в плоскости организации сети дорог, а не устройства автомобиля.

Аналогично особенности и возможности системы обмена сообщениями определяются в первую очередь не программными компонентами, ведущими обмен сообщениями, и не содержанием самих сообщений, а транспортными коммуникациями, по которым эти сообщения перемещаются.

В данной главе мы рассмотрим модель, имеющую поверхностное сходство с акторами, но совершенно иное внутреннее устройство.

Взаимодействия – это все

Как мы видели в предыдущей главе, программы на основе акторов состоят из независимых сущностей (которые называются акторами, или процессами в Elixir), выполняющихся конкурентно и взаимодействующих посредством обмена сообщениями. Каждый актор имеет свой почтовый ящик, где хранятся необработанные сообщения.

Программа, использующая модель взаимодействий последовательных процессов, также состоит из независимых сущностей, выполняющихся конкурентно и взаимодействующих посредством обмена сообщениями. Разница заключается в фокусе – вместо сущ-

ностей, посылающих и принимающих сообщения, в центре модели CSP находятся *каналы*, по которым осуществляется обмен. Канал – это сущность первого порядка, которая не связывает процесс с конкретным почтовым ящиком. Каналы могут создаваться независимо, в них можно записывать сообщения, из них можно читать сообщения, и их можно передавать между процессами.

Подобно акторам и функциональному программированию, CSP является довольно старой идеей, переживающей эпоху возрождения. Нынешняя популярность CSP в значительной степени обусловлена появлением языка Go.¹ Далее мы будем обсуждать различные стороны модели CSP на примерах применения библиотеки `core.async`,² которая привнесла модель конкуренции из языка Go в Clojure.

Библиотека `core.async`

Библиотека `core.async` появилась в языке Clojure относительно недавно и все еще находится в состоянии тестирования (поэтому имейте в виду, что кое-что в ней может поменяться). Чтобы получить возможность использовать библиотеку в программе, ее необходимо включить в зависимости проекта и импортировать. Здесь возникает одна сложность из-за того, что некоторые функции в библиотеке `core.async` имеют имена, совпадающие с именами функций в стандартной библиотеке Clojure. Чтобы вам было проще следовать за экспериментами, вы можете использовать проект `channels` из загружаемого пакета примеров для книги, который импортирует `core.async`, как показано ниже:

```
CSP/channels/src/channels/core.clj*
```

```
(ns channels.core
  (:require [clojure.core.async :as async :refer :all
            :exclude [map into reduce merge partition partition-
                     by take]]))
```

Ключевые слова `:refer :all` обеспечивают возможность непосредственного использования большинства функций из `core.async`, но некоторые (имена которых вступают в конфликт с именами функций из стандартной библиотеки) придется использовать, предваряя их префиксом `async/`.

Чтобы запустить интерактивную оболочку REPL со всеми загруженными определениями, перейдите в каталог проекта `channels` и введите команду `lein repl`.

* <http://media.pragprog.com/titles/pb7con/code/CSP/channels/src/channels/core.clj>

¹ <http://golang.org>

² <http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html>

В первый день мы познакомимся с двумя столпами, на которых основана библиотека `core.async`: каналы и блоки `go`. Во второй день мы сконструируем программу с помощью этой библиотеки. Наконец, в третий день мы посмотрим, как можно использовать библиотеку `core.async` в сценариях на ClojureScript для упрощения разработки клиентских сценариев.

День 1: каналы и блоки go

Библиотека `core.async` предоставляет два основных инструмента – каналы и блоки `go`. Блоки `go` позволяют организовать выполнение многочисленных задач на ограниченном множестве потоков. Но пока отложим их в сторону и познакомимся с каналами.

Каналы

Канал – это очередь с поддержкой использования в многопоточном окружении. Любая задача, имеющая ссылку на канал, может добавлять сообщения с одного конца, и любая задача, имеющая ссылку на канал, может извлекать сообщения с другого конца. В отличие от модели акторов, где сообщения посылаются определенным акторам, в модели CSP отправитель ничего не знает о приемнике, и наоборот.

Новый канал создается с помощью `chan`:

```
channels.core=> (def c (chan))
#'channels.core/c
```

Запись в канал осуществляется с помощью `>!!`, а чтение – с помощью `<!!`:

```
channels.core=> (thread (println "Read:" (<!! c) "from c"))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@78fcc563>
channels.core=> (>!! c "Hello thread")
Read: Hello thread from c
nil
```

Здесь используется удобный вспомогательный макрос `thread` из библиотеки `core.async`, который, как можно понять из имени, выполняет переданный ему код в отдельном потоке. Этот код выводит любое сообщение, извлеченное из канала. Он блокируется до момента, пока мы не запишем что-нибудь в канал с помощью `>!!`, после чего на экране появится сообщение.

Буферизация

По умолчанию каналы действуют *синхронно* (то есть *не буферизуются*) – операция записи в канал блокируется, пока сообщение не будет прочитано:

```
channels.core=> (thread (>!! c "Hello") (println "Write completed"))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@78fcc563>
channels.core=> (<!! c)
Write completed
"Hello"
```

Однако есть возможность создать буферизованный канал, передав размер буфера в `chan`:

```
channels.core=> (def bc (chan 5))
#'channels.core/bc
channels.core=> (>!! bc 0)
nil
channels.core=> (>!! bc 1)
nil
channels.core=> (close! bc)
nil
channels.core=> (<!! bc)
0
channels.core=> (<!! bc)
1
channels.core=> (<!! bc)
nil
```

Здесь создается канал с буфером на пять сообщений. Пока в буфере имеется место, операции записи будут выполняться мгновенно.

Заккрытие каналов

В предыдущем примере была показана еще одна особенность каналов – они могут закрываться с помощью `close!`. Операция чтения из пустого закрытого канала вернет `nil`, а операция записи в закрытый канал просто приведет к потере сообщения. Как можно догадаться, попытка записать `nil` в канал приведет к ошибке:

```
channels.core=> (>!! (chan) nil)
IllegalArgumentException Can't put nil on channel "..."
```

Ниже приводится реализация функции, которая читает сообщения из канала, пока он не будет закрыт, после чего возвращает вектор с прочитанными сообщениями:

CSP/channels/src/channels/core.clj³

```
(defn readall!! [ch]
  (loop [coll []]
    (if-let [x (<!! ch)]
      (recur (conj coll x))
      coll)))
```

Она в цикле заполняет изначально пустой вектор `coll`. В каждой итерации производится чтение значения из `ch`, и если значение не равно `nil`, оно добавляется в `coll`. Если значение равно `nil` (канал был закрыт), возвращает `coll`.

А далее приводится функция `writeall!!`, которая принимает канал и последовательность и записывает элементы последовательности в канал. После записи последнего элемента канал закрывается:

CSP/channels/src/channels/core.clj⁴

```
(defn writeall!! [ch coll]
  (doseq [x coll]
    (>!! ch x))
  (close! ch))
```

Давайте посмотрим, как действуют эти функции:

```
channels.core=> (def ch (chan 10))
#'channels.core/ch
channels.core=> (writeall!! ch (range 0 10))
nil
channels.core=> (readall!! ch)
[0 1 2 3 4 5 6 7 8 9]
```

Думаю, вас не удивит, если я скажу, что в библиотеке `core.async` уже имеются функции, решающие аналогичные задачи:

```
channels.core=> (def ch (chan 10))
#'channels.core/ch
channels.core=> (onto-chan ch (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@6b16d3cf>
channels.core=> (<!! (async/into [] ch))
[0 1 2 3 4 5 6 7 8 9]
```

Функция `onto-chan` записывает содержимое коллекции в канал и закрывает его. А функция `async/into` принимает начальную коллек-

³ <http://media.pragprog.com/titles/pb7con/code/CSP/channels/src/channels/core.clj>

⁴ <http://media.pragprog.com/titles/pb7con/code/CSP/channels/src/channels/core.clj>

цию (пустой вектор в данном примере) и канал, и возвращает канал. Возвращаемый канал будет содержать записанную в него исходную коллекцию и все, что было прочитано из указанного канала.

Эти утилиты мы будем использовать далее для более детального исследования каналов.

Стратегии реакции на переполнение буфера

По умолчанию попытка записи в заполненный канал блокируется, но есть возможность выбрать альтернативную стратегию, передав буфер функции `chan`:

```
channels.core=> (def dc (chan (dropping-buffer 5)))
#'channels.core/dc
channels.core=> (onto-chan dc (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@147c0def>
channels.core=> (<!! (async/into [] dc))
[0 1 2 3 4]
```

Здесь мы создали канал с *отбрасывающим буфером* (`dropping buffer`) на пять сообщений, затем записываем в канал числа от 0 до 9. Операция записи при этом не блокируется, даже несмотря на то что канал не способен вместить так много сообщений. Выполнив операцию чтения из канала, мы обнаруживаем, что получили только пять первых сообщений – все последующие сообщения были отброшены.

В Clojure есть также возможность использовать *скользящий буфер* (`sliding buffer`):

```
channels.core=> (def sc (chan (sliding-buffer 5)))
#'channels.core/sc
channels.core=> (onto-chan sc (range 0 10))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@3071908b>
channels.core=> (<!! (async/into [] sc))
[5 6 7 8 9]
```

Как и в предыдущем примере, мы создали канал с буфером, вмещающим пять сообщений, но на этот раз использовали *скользящий буфер*. Выполнив операцию чтения из канала, мы обнаруживаем, что получили пять последних сообщений – запись в заполненный канал со скользящим буфером приводит к отбрасыванию самых первых сообщений. Мы продолжим знакомство с каналами позже, а сейчас посмотрим на другой основной инструмент из библиотеки `core.async` – блоки `go`.

Блоки go

Потоки выполнения имеют существенные накладные расходы и являются весьма дорогостоящими на запуске, из-за чего во многих современных программах программисты стараются избегать создания потоков непосредственно и используют пулы потоков (см. раздел «Еще раз о создании потоков» в главе 2). Макрос `thread`, который мы использовали выше, тоже за кулисами использует пул потоков `CachedThreadPool`.



Вопрос Джо: неужели нет буфера, растущего автоматически?

Мы познакомились со всеми тремя типами буферов, поддерживаемых библиотекой `core.async`, – блокирующие, отбрасывающие и скользящие. Технически вполне возможно было бы создать тип буферов, растущих автоматически по мере отправки все новых и новых сообщений. Так почему же такие буферы не поддерживаются?

Причина заключается в старой известной проблеме: всякий раз, когда появляется «неисчерпаемый» ресурс, тут же возникает опасность исчерпать его. Это может произойти, например, когда программе приходится выполнять огромные объемы работы или из-за ошибки, приводящей к накоплению сообщений.

Если вы не будете задумываться о том, чтобы своевременно устранять накапливающиеся проблемы, в конечном итоге это приведет к неясным и трудно диагностируемым нарушениям в работе программы. В действительности переполнение почтового ящика процесса является одним из очень немногих способов полностью нарушить работу системы на Erlang.* Лучше думать о том, как обрабатывать ситуацию заполнения буфера до отказа, и уничтожить проблему в зародыше.

* <http://prog21.dadgum.com/43.html>

Однако пулы потоков не всегда удобны в использовании. В частности, они могут приносить проблемы, если код, который нам требуется выполнить, может оказаться заблокированным.

Проблема блокировки

Пулы потоков – отличный способ решения вычислительных задач, которые занимают потоки на короткие промежутки времени и вновь возвращают их в пул. Но как быть, если задачи должны осуществлять взаимодействия друг с другом? Блокирование потока свяжет его на

неопределенное время, сведя на нет все преимущества использования пула.

Существует несколько способов решения этой проблемы, но обычно они сводятся к реструктуризации кода и превращению его в код, *управляемый событиями*, — этот стиль программирования известен всем, кто занимался разработкой пользовательских интерфейсов или работал с любыми современными серверами, управляемыми событиями.

Несмотря на то что эти решения работают, они нарушают естественное течение потока управления и могут сделать код более сложным для чтения и анализа. Хуже того, часто эти решения приводят к расширению глобального состояния из-за необходимости сохранения данных одними обработчиками событий для использования их в других обработчиках событий. А, как вы знаете, состояние и конкуренция несовместимы.

Блоки `go` предоставляют альтернативное решение, совмещающее все лучшее из обоих миров — эффективность кода, управляемого событиями, без необходимости жертвовать его структурой и удобочитаемостью. Достигается это за счет прозрачного превращения последовательного кода в код, управляемый событиями.

Инверсия управления

Подобно другим диалектам Lisp, язык Clojure обладает мощной системой поддержки макросов. Если вам уже приходилось пользоваться макросами в других языках (например, макросами препроцессора C/C++), макросы Lisp могут казаться вам удивительным инструментом, позволяющим существенно изменять код. Но макрос `go` — это уже настоящая магия.

Код внутри блока `go` преобразуется в конечный автомат (state machine). Вместо блокирования потока внутри операции записи в канал или чтения из канала конечный автомат переводится в режим ожидания, а сам поток выполнения сохраняет возможность продолжения работы. Когда в следующий раз он получит возможность поработать, он произведет переход из одного состояния в другое и продолжит выполнение, возможно в другом потоке.

Такая инверсия управления позволяет библиотеке `core.async` эффективно переключаться между множеством блоков `go`, используя ограниченный пул потоков выполнения. Насколько эффективно такое решение, мы увидим потом, а сейчас рассмотрим пример.

Приостановка

Ниже приводится простой пример использования блока `go`:

```
channels.core=> (def ch (chan))
#'channels.core/ch
channels.core=> (go
  #=> (let [x (<! ch)
  #=>      y (<! ch)]
  #=>      (println "Sum:" (+ x y))))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@13ac7b98>
channels.core=> (>!! ch 3)
nil
channels.core=> (>!! ch 4)
nil
Sum: 7
```

В этом примере сначала создается канал `ch`. Далее следует блок `go`, который читает два значения из канала и выводит их сумму. Хотя все выглядит так, как будто блок `go` должен блокироваться в операции чтения из канала, в действительности происходит кое-что более интересное.

Вместо `<!!` для чтения из канала наш блок `go` использует `<!`. Единственный восклицательный знак указывает, что это — *ожидающая* версия операции чтения, а не блокирующая. Как вы наверняка догадались, функция `>!` является ожидающей версией блокирующей функции `>!!`.

Макрос `go` преобразует последовательный код в конечный автомат с тремя состояниями, как показано на рис. 19.



Рис. 19. Конечный автомат с тремя состояниями

1. Начальное состояние сразу после перехода в режим ожидания. Ожидание появления в канале `ch` значения, доступного для чтения. Когда это происходит, автомат переходит в следующее состояние 2.
2. Конечный автомат связывает значение, прочитанное из канала `ch`, с переменной `x` и снова переходит в режим ожидания до появления другого значения, после чего переходит в состояние 3.

3. Конечный автомат связывает значение, прочитанное из канала `ch`, с переменной `y`, выводит сообщение и завершает работу.



Вопрос Джо: что случится, если в блоке `go` использовать блокирующую версию функции?

Если в блоке `go` вызвать блокирующую функцию, такую как `<!!`, поток просто заблокируется. Код в блоке `go` при этом, вероятно, выполнится вполне благополучно (хотя есть риск заблокировать все доступные потоки и оказаться в состоянии взаимоблокировки), но такой подход противоречит цели использования блоков `go`. Если вы допустите подобное по ошибке, то не увидите никаких предупреждений, поэтому имейте в виду, что вся ответственность лежит только на вас.

К счастью, если допустить обратную ошибку, вы увидите примерно такое предупреждение:

```
channels.core=> (<! ch)
AssertionError Assert failed: <! used not in (go ...) block
nil clojure.core.async/<! (async.clj:83)
```

Блоки `go` имеют низкие накладные расходы

Главная цель волшебства, кроющегося под покровом макроса `go`, — эффективность. Так как (в отличие от потоков выполнения) блоки `go` имеют низкие накладные расходы, можно создать множество таких блоков, не опасаясь исчерпать доступные ресурсы. Кажется, что это не особенно большое преимущество, но возможность создавать массу конкурирующих задач без особых проблем может оказаться весьма полезной.

Возможно, вы обратили внимание, что блок `go` (как и макрос `thread`) возвращает канал. Этот канал хранит результат, записанный блоком `go`:

```
channels.core=> (<!! (go (+ 3 4)))
7
```

Данное обстоятельство можно использовать для создания маленькой функции, создающей очень большое число блоков `go`, что позволит вам убедиться, насколько недорогими являются блоки `go`:

CSP/channels/src/channels/core.clj⁵

```
(defn go-add [x y]
  (<!! (nth (iterate #(go (inc (<! %))) (go x)) y)))
```

Эта функция является претендентом на звание «самой неэффективной функции сложения». Она складывает x и y , создавая конвейер из y блоков `go`, каждый такой блок выполняет увеличение своего аргумента на единицу.

Чтобы понять, как она работает, разобьем ее на фрагменты:

1. Анонимная функция `%(go (inc (<! %)))` создает блок `go`, который принимает канал, читает из него единственное значение и возвращает канал со значением, увеличенным на единицу.
2. Эта функция передается в `iterate` с начальным значением `(go x)` (канал, который хранит исходное значение x). Напомним, что `iterate` возвращает ленивую последовательность вида: `(x (f x) (f (f x)) (f (f (f x))) ...)`.
3. С помощью `nth` выполняется чтение y -го элемента этой последовательности, значением которого будет канал с результатом увеличения x на единицу y раз.
4. В заключение выполняется чтение значения из канала с помощью `<!!`.

А теперь посмотрим, как она действует:

```
channels.core=> (time (go-add 10 10))
"Elapsed time: 1.935 msecs"
20
channels.core=> (time (go-add 10 1000))
"Elapsed time: 5.311 msecs"
1010
channels.core=> (time (go-add 10 100000))
"Elapsed time: 734.91 msecs"
100010
```

Как видите, за три четверти секунды было создано и выполнено 100 000 блоков `go`. Результаты показывают, что блоки `go` с успехом могут соперничать с процессами в Elixir – результат выглядит еще более внушительным, если учесть, что код на Clojure выполняется под управлением JVM, а код на Elixir – под управлением виртуальной машины Erlang, специально создававшейся с прицелом на конкурентное выполнение кода.

⁵ <http://media.pragprog.com/titles/pb7con/code/CSP/channels/src/channels/core.clj>

Теперь, когда вы увидели, как работают блоки `go` и каналы, можно переходить к конструированию из них более сложных операций с каналами.

Операции с каналами

Если вам покажется, что каналы похожи на последовательности не только при беглом взгляде, вы будете совершенно правы. Так же как последовательности, каналы представляют упорядоченные множества значений. Как и при использовании последовательностей, мы можем писать функции высшего порядка, оперирующие сразу всем содержимым каналов, такие как `map`, `filter` и др. И точно так же мы сможем объединять такие функции в цепочки для определения сложных операций.

Отображение в канал

Ниже приводится версия функции `map` для работы с каналом:

CSP/channels/src/channels/core.clj⁶

```
(defn map-chan [f from]
  (let [to (chan)]
    (go-loop []
      (when-let [x (<! from)]
        (>! to (f x))
        (recur))
      (close! to))
    to))
```

Она принимает функцию (`f`) и исходный канал (`from`). Отображение начинается с создания выходного канала (`to`), который будет возвращен в конце. Но перед этим создается блок `go` вызовом `go-loop`, вспомогательной функции, эквивалентной конструкции `(go (loop ...))`. В теле цикла `when-let` читает значение из `from` и связывает его с переменной `x`. Если `x` не равно `nil`, выполняется тело `when-let`, записывающее результат `(f x)` в `to`, после чего цикл повторяется. Если `x` имеет значение `nil`, канал `to` закрывается.

Вот как действует эта функция:

```
channels.core=> (def ch (chan 10))
#'channels.core/ch
channels.core=> (def mapped (map-chan (partial * 2) ch))
#'channels.core/mapped
channels.core=> (onto-chan ch (range 0 10))
```

⁶ <http://media.pragprog.com/titles/pb7con/code/CSP/channels/src/channels/core.clj>

```
#<ManyToManyChannel          clojure.core.async.impl.channels.
ManyToManyChannel@9f3d43e>
channels.core=> (<!! (async/into [] mapped))
[0 2 4 6 8 10 12 14 16 18]
```

Как можно было бы предположить, `core.async` предоставляет собственную версию функции `map-chan`, которая называется `map<`. Существует также версия `filter` для работы с каналом, которая называется `filter<`, версия `mapcat` с именем `mapcat<` и т. д. Нетрудно догадаться, что эти функции можно объединять в цепочки:

```
channels.core=> (def ch (to-chan (range 0 10)))
#'channels.core/ch
channels.core=> (<!! (async/into [] (map< (partial * 2) (filter< even? ch))))
[0 4 8 12 16]
```

В этом примере используется `to-chan`, еще одна вспомогательная функция из `core.async`, которая создает и возвращает канал с содержимым указанной последовательности, закрывая канал после исчерпания последовательности.

Мы практически подошли к концу первого дня, но, прежде чем завершить его, давайте немного поиграем.

Конкурентное «Решето Эратосфена»

Ниже представлена конкурентная версия задачи «Решето Эратосфена». Функция `get-primes` возвращает канал с простыми числами, не превышающими указанного предела `limit`:

SP/Sieve/src/sieve/core.clj⁷

```
(defn factor? [x y]
  (zero? (mod y x)))

(defn get-primes [limit]
  (let [primes (chan)
        numbers (to-chan (range 2 limit))]
    (go-loop [ch numbers]
      (when-let [prime (<! ch)]
        (>! primes prime)
        (recur (remove< (partial factor? prime) ch)))
      (close! primes))
    primes))
```

Немного погодя мы подробно рассмотрим особенности работы этой функции (впрочем, будет даже лучше, если вы попытаете разо-

⁷ <http://media.pragprog.com/titles/pb7con/code/CSP/Sieve/src/sieve/core.clj>

браться в ней самостоятельно прямо сейчас — вы уже обладаете всеми знаниями, необходимыми для этого). Но перед этим я предлагаю убедиться, что она действует в соответствии с нашими ожиданиями. Следующая функция `main` вызывает `get-primes` и выводит все, что было записано в возвращаемый ею канал:

```
CSP/Sieve/src/sieve/core.clj8
```

```
(defn -main [limit]
  (let [primes (get-primes (edn/read-string limit))]
    (loop []
      (when-let [prime (<!! primes)]
        (println prime)
        (recur))))))
```

Далее показано, что получится, если запустить ее:

```
$ lein run 100000
2357
11
.
.
.
99971
99989
99991
```

А теперь посмотрим, как работает функция `get-primes`. Сначала она создает канал `primes`, который будет возвращен в конце функции. Затем она входит в цикл, перед началом которого с именем `ch` связывается канал `numbers`, хранящий все числа от 2 до `limit`, которые предварительно записываются в канал функцией `to-chan`.

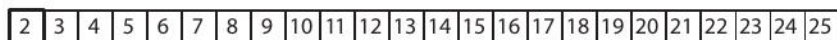
Цикл читает первый элемент из канала `ch`, который является простым числом (чуть ниже вы поймете, почему это утверждение верно), и записывает его в канал `primes`. После этого происходит переход в начало цикла, только на этот раз имя `ch` связано с результатом выражения `(remove< (partial factor? prime) ch)`.

Функция `remove<` действует подобно функции `filter<`, за исключением того, что возвращает канал, содержащий только значения, для которых предикат вернул `false`. В нашем случае это будет канал со всеми значениями, для которых только что найденное простое число является делителем.

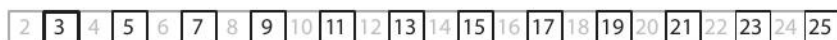
Таким образом, функция `get-primes` создает конвейер из каналов; первый содержит все числа от 2 до `limit`, второй содержит множество

⁸ <http://media.pragprog.com/titles/pb7con/code/CSP/Sieve/src/sieve/core.clj>

всех чисел, из которого удалены числа, делящиеся на 2, затем – множество всех чисел, из которого удалены числа, делящиеся на 3, и т. д., как показано на рис. 20.



(remove< (factor? 2 ...) ...)



(remove< (factor? 3 ...) ...)



(remove< (factor? 5 ...) ...)



Рис. 20. Конкурентное «Решето Эратосфена»

Я бы не хотел, чтобы у вас создалось впечатление, что каналы являются эффективным инструментом параллельного решения задачи «Решето Эратосфена»: каналы – слишком дорогое удовольствие для этого. Но данный пример отлично демонстрирует, что каналы легко можно объединять для реализации различных шаблонов взаимодействий.

В завершение первого дня

На этом заканчивается первый день. Завтра мы посмотрим, как читать данные из более чем одного канала и как создавать программы, интенсивно осуществляющие ввод/вывод, с применением каналов и блоков go.

Что мы узнали в первый день

Каналы и блоки go являются двумя столпами, на которых покоится `core.async`.

- по умолчанию каналы (небуферизованные) действуют синхронно — запись в канал блокируется, пока кто-нибудь не прочитает значение, уже хранящееся в нем;
- каналы могут также быть буферизованными. Поддержка разных стратегий буферизации позволяет нам решать, какие последствия должно иметь заполнение буфера — блокирование, потерю самых старых значений (скользящий буфер) или потерю самых последних значений (отбрасывающий буфер);
- блоки `go` используют прием инверсии управления, чтобы преобразовать код в конечный автомат. Блоки `go` не блокируют потока выполнения, а приостанавливаются сами, позволяя системе использовать поток выполнения для нужд других блоков `go`;
- имена блокирующих версий операций с каналами оканчиваются двумя восклицательными знаками (!!), тогда как имена приостанавливающих версий оканчиваются одним восклицательным знаком (!).

Упражнения к первому дню

Найдите:

- документацию с описанием `core.async`;
- скринкаст Тимоти Балдриджа (Timothy Baldridge) «Core Async Go Macro Internals» или статью Хьюи Петерсена (Huey Petersen) «The State Machines of core.async», где рассказывается, как макрос `go` реализует прием инверсии управления.

Задания:

- наша реализация `map-chan` создает и возвращает синхронный (небуферизованный) канал. Что случится, если в ней создать буферизованный канал? Какое решение выглядит предпочтительнее? При каких условиях предпочтительнее выглядят буферизованные каналы (если это вообще имеет смысл)?
- помимо функции `map<`, в библиотеке `core.async` имеется также функция `map>`. Чем они отличаются? Напишите свою версию `map>`. В каких ситуациях следует использовать одну, а в каких другую?
- напишите параллельную версию `map` на основе каналов (похожую на функцию `rmap` в языке Clojure или параллельную версию `map`, созданную нами на языке Elixir в предыдущей главе).

День 2: множество каналов и ввод/вывод

Сегодня мы посмотрим, как библиотека `core.async` способна упростить реализацию асинхронного ввода/вывода и повысить ее удобочитаемость в исходном программном коде. Но перед этим рассмотрим некоторые приемы, которых пока еще не видели, позволяющие одновременно обслуживать множество каналов.

Обслуживание множества каналов

До сих пор в наших примерах мы обслуживали по одному каналу за раз, однако нет ничего, что препятствовало бы обслуживанию сразу множества каналов. Макрос `alt!` позволяет писать код, который имеет дело сразу с несколькими каналами:

```
channels.core=> (def ch1 (chan))
#'channels.core/ch1
channels.core=> (def ch2 (chan))
#'channels.core/ch2
channels.core=> (go-loop []
  #_=> (alt!
    #_=> ch1 ([x] (println "Read" x "from channel 1")))
    #_=> ch2 ([x] (println "Twice" x "is" (* x 2))))
  #_=> (recur))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@d8fd215>
channels.core=> (>!! ch1 "foo")
Read foo from channel 1
nil
channels.core=> (>!! ch2 21)
Twice 21 is 42
nil
```

Здесь создаются два канала, `ch1` и `ch2`, и затем следует блок `go`, выполняющий бесконечный цикл, в котором производит чтение сразу из обоих каналов с помощью `alt!`. Если в канале `ch1` имеется что-то, доступное для чтения, это «что-то» выводится на экран. Если в канале `ch2` имеется что-то, доступное для чтения, это «что-то» удваивается и также выводится на экран.

Из контекста должно быть очевидно, что здесь происходит: макрос `alt!` принимает пару аргументов, первый из которых определяет канал, а второй — код, который должен быть выполнен, если что-то удастся прочитать из указанного канала. В данном примере код напоминает анонимную функцию — значение, прочитанное из канала, связывается

с именем `x` и затем выводится последующим вызовом `println`. Но это не анонимная функция – код не начинается с формы `fn`.

Это еще один пример волшебства системы макросов в языке Clojure, которое обеспечивает краткость синтаксиса и более высокую эффективность, чем способны дать анонимные функции.



Вопрос Джо: возможна ли запись сразу в несколько каналов?

Мы лишь слегка затронули возможности макроса `alt!` – помимо одновременного чтения из множества каналов, он позволяет также записывать данные сразу в несколько каналов и даже смешивать операции чтения и записи. Мы не будем пользоваться этими возможностями в данной книге, но желающим глубже узнать возможности макроса `alt!` стоит обратиться к соответствующей документации.

Тайм-ауты

Функция `timeout` возвращает канал, автоматически закрывающийся через указанное число миллисекунд:

```
channels.core=> (time (<!! (timeout 10000)))
"Elapsed time: 10001.662 msecs"
nil
```

Ее можно использовать в сочетании с макросом `alt!` для ограничения времени выполнения других операций с каналами, как показано ниже:

```
channels.core=> (def ch (chan))
#'channels.core/ch
channels.core=> (let [t (timeout 10000)]
  #_=> (go (alt!
    #_=> ch ([x] (println "Read" x "from channel")))
    #_=> t (println "Timed out"))))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@28134be9>
channels.core=>
Timed out
```

В тайм-аутах нет ничего нового, но такой подход, когда тайм-ауты относятся к некоторой конкретной задаче, оказывается удивительно мощным, в чем мы убедимся далее.

Овеществленные тайм-ауты

Большинство систем поддерживает тайм-ауты, относящиеся к запросам. Класс `URLConnection` в Java, например, поддерживает метод `setReadTimeout()` – если сервер не отвечает в течение указанного числа миллисекунд, метод `read()` возбуждает исключение `IOException`.

Это отлично подходит для ситуаций, когда выполняется единственный запрос. Но что, если требуется ограничить общее время для выполнения целой последовательности запросов? Определение тайм-аутов для каждого запроса в отдельности мало поможет в этом, а овеществленный тайм-аут – это именно то, что нужно, достаточно определить единственный тайм-аут и использовать его для каждого запроса в последовательности.

Для иллюстрации изменим пример решения задачи «Решето Эратосфена», созданный вчера, только на этот раз ограничим не количество чисел, а сгенерируем так много простых чисел, сколько удастся за определенное число секунд.

Сначала изменим функцию `get-primes` так, чтобы она генерировала простые числа до бесконечности:

CSP/SieveTimeout/src/sieve/core.clj⁹

```
(defn get-primes []
  (let [primes (chan)
        numbers (to-chan (iterate inc 2))]
    > (go-loop [ch numbers]
      (when-let [prime (<! ch)]
        (>! primes prime)
        (recur (remove< (partial factor? prime) ch)))
      (close! primes))
    primes))
```

Вместо выражения `(range 2 limit)` для создания исходного канала в этой версии используется бесконечная последовательность `(iterate inc 2)`.

Ниже показано, как используется эта функция:

CSP/SieveTimeout/src/sieve/core.clj¹⁰

```
(defn -main [seconds]
  (let [primes (get-primes)
        limit (timeout (* (edn/read-string seconds) 1000))]
    >
```

⁹ <http://media.pragprog.com/titles/pb7con/code/CSP/SieveTimeout/src/sieve/core.clj>

¹⁰ <http://media.pragprog.com/titles/pb7con/code/CSP/SieveTimeout/src/sieve/core.clj>


```
(loop []
  > (alt!! :priority true
  >   limit nil
  >   primes ([prime] (println prime) (recur))))))
```

Здесь применяется макрос `alt!!`, который, как вы наверняка поняли, является блокирующей версией макроса `alt!`. Он блокирует выполнение до тех пор, пока либо не появится новое простое число, либо не истечет время тайм-аута, – и в том, и в другом случае он просто возвращает `nil`. Параметр `:priority true` гарантирует, что предложения, переданные макросу `alt!!`, будут вычисляться по порядку (по умолчанию, если есть возможность выполнить два предложения, выбор между производится недетерминированно). Это исключает (по общему мнению, маловероятную) ситуацию, когда простые числа будут генерироваться настолько быстро, что всегда будет доступно хотя бы одно число и предложение, определяющее завершение тайм-аута, никогда не будет выполнено. Данный пример представляет более естественный способ выразить решаемую проблему – намного более естественный, чем позволяют тайм-ауты для отдельных запросов.

В следующем разделе демонстрируется использование тайм-аутов совместно с системой макросов в языке Clojure для создания вспомогательной утилиты, решающей такую общую задачу, как опрос.

Асинхронный опрос

Немного позднее, сегодня же, мы напишем инструмент для чтения RSS. Помимо всего прочего, в нем необходимо будет реализовать опрос лент новостей для выявления новых статей. В этом разделе мы будем использовать тайм-ауты совместно с поддержкой макросов в Clojure для создания утилиты, существенно упрощающей реализацию асинхронного опроса.

Функция опроса

Функция `timeout`, которую мы видели уже сегодня, – именно то, что нам необходимо для реализации опроса. Ниже приводится функция, которая принимает интервал времени в секундах и функцию и вызывает эту функцию по истечении каждого интервала:

```
CSP/Polling/src/polling/core.clj11
```

```
(defn poll-fn [interval action]
```

¹¹ <http://media.pragprog.com/titles/pb7con/code/CSP/Polling/src/polling/core.clj>


```
(let [seconds (* interval 1000)]
  (go (while true
        (action)
        (<! (timeout seconds))))))
```

Это довольно простая функция, и работает она в полном соответствии с нашими ожиданиями:

```
polling.core=> (poll-fn 10 # (println "Polling at:" (System/currentTimeMillis)))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@6e624159>
polling.core=>
Polling at: 1388827086165
Polling at: 1388827096166
Polling at: 1388827106168
.
.
.
```

Но есть одна проблема – может показаться, что из-за того, что `poll-fn` вызывает указанную функцию в блоке `go`, она должна иметь возможность вызывать функции, осуществляющие приостановку. Однако давайте посмотрим, что получится в этом случае:

```
polling.core=> (def ch (to-chan (iterate inc 0)))
#'polling.core/ch
polling.core=> (poll-fn 10 # (println "Read:" (<! ch)))
Exception in thread "async-dispatch-1" java.lang.AssertionError:
  Assert failed: <! used not in (go ...) block
nil
```

Проблема в том, что вызов функции, осуществляющей приостановку, должен производиться непосредственно в блоке `go`, – система макросов в языке Clojure не способна творить свое волшебство в иных случаях.

Макрос опроса

Решение проблемы, обозначенной выше, состоит в том, чтобы реализовать утилиту опроса в виде макроса:

CSP/Polling/src/polling/core.clj¹²

```
(defmacro poll [interval & body]
  `(let [seconds# (* ~interval 1000)]
    (go (while true
```

¹² <http://media.pragprog.com/titles/pb7con/code/CSP/Polling/src/polling/core.clj>

```
(do ~@body)
(<! (timeout seconds#))))))
```

У нас нет возможности заниматься детальным обсуждением макросов Clojure, поэтому вам придется поверить мне на слово. Вскоре я покажу, как происходит разворачивание макроса `poll`, а пока дам несколько подсказок, которые помогут вам понять, как он работает:

- макрос не компилируется непосредственно, а возвращает программный код, который затем будет скомпилирован;
- обратный апостроф (```) — это оператор синтаксической кавычки (syntax quote operator); он принимает исходный код и вместо его выполнения возвращает представление этого кода в форме, пригодной для компиляции;
- внутри этого кода можно использовать операторы `~` (*синтаксического размаскирования*) и `~@` (*размаскирования с объединением*) для ссылки на аргументы, передаваемые в макрос;
- окончание `#` (*автоматический генератор символов*) сообщает, что Clojure должен автоматически сгенерировать уникальное имя (гарантирующее отсутствие конфликтов с любыми именами, которые могут быть переданы в макрос).

Давайте посмотрим на него в действии:

```
polling.core=> (poll 10
                #_=> (println "Polling at:" (System/currentTimeMillis))
                #_=> (println (<! ch)))
#<ManyToManyChannel clojure.core.async.impl.channels.
ManyToManyChannel@1bec079e>
polling.core=>
Polling at: 1388829368011
0
Polling at: 1388829378018
1
.
.
.
```

Так как разворачивание макроса происходит на этапе компиляции, код, переданный в `poll`, встраивается в окружающий код и, соответственно, оказывается непосредственно внутри блока `do`, благодаря чему мы получаем возможность передавать код, содержащий функции приостановки. Но это не единственное преимущество использования макроса — так как мы передаем фрагмент кода, а не функцию, синтаксис выглядит намного более естественно — отпадает необхо-

димось создавать анонимную функцию. Фактически мы только что создали собственную управляющую конструкцию.

Увидеть, как выглядит код, сгенерированный макросом `poll`, можно, выполнив развертывание макроса:

```
polling.core=> (macroexpand-1
  #_=> '(poll 10
  #_=> (println "Polling at:" (System/currentTimeMillis))
  #_=> (println (<! ch)))
(clojure.core/let [seconds__2691__auto__ (clojure.core/* 10 1000)]
  (clojure.core.async/go
    (clojure.core/while true
      (do
        (println "Polling at:" (System/currentTimeMillis))
        (println (<! ch)))
        (clojure.core.async/<! (clojure.core.async/timeout seconds__
2691__auto__))))))
```

Я немного отформатировал вывод `macroexpand-1`, чтобы сделать его более удобочитаемым. Вы можете видеть, как программный код, переданный макросу `poll`, был вставлен (объединен) в код, окружающий сам макрос, и как имя `seconds#` превратилось в уникальное имя (чтобы понять, почему это важно, представьте, что код, переданный в макрос `poll`, уже использует имя `seconds` для обозначения чего-то еще).

В следующем разделе мы посмотрим пример практического применения макроса `poll`.

Асинхронный ввод/вывод

Ввод/вывод – это область, где асинхронный код оказывается как нельзя кстати, в отличие от традиционного подхода, когда для каждого соединения создается отдельный поток выполнения. Поддержка асинхронного ввода/вывода позволяет запускать множество операций и получать извещения, когда данные будут доступны. Однако, несмотря на радужные перспективы, реализовать такой подход не всегда просто – программный код стремится превратиться в мешанину из функций-обработчиков, вызывающих другие функции-обработчики. В этом разделе мы посмотрим, как `core.async` может облегчить нам жизнь.

Опираясь на знания, приобретенные при реализации подсчета слов в предыдущих главах, мы создадим инструмент для чтения RSS, который будет следить за множеством лент новостей, выявлять новые статьи и выполнять подсчет слов, содержащихся в них. Реализация будет иметь вид конвейера блоков `go`, выполняющихся конкурентно:

1. Блок `go` на самом нижнем уровне будет следить за единственной лентой новостей, опрашивая ее каждые 60 секунд. После парсинга полученных данных в формате XML он будет извлекать ссылки на новые статьи и передавать их далее по конвейеру.
2. Следующий блок `go` будет управлять списком всех статей, уже извлеченных из конкретной ленты новостей. Всякий раз, при обнаружении новой статьи, он будет передавать ее URL дальше по конвейеру.
3. Следующий блок `go` будет поочередно извлекать новые статьи, выполнять подсчет слов в ней и передавать значения счетчиков дальше по конвейеру.
4. Значения счетчиков из нескольких лент новостей будут объединяться в единый канал.
5. Блок `go` на самом верхнем уровне будет следить за этим объединенным каналом и выводить новые значения счетчиков.

Структура приложения показана на рис. 21.

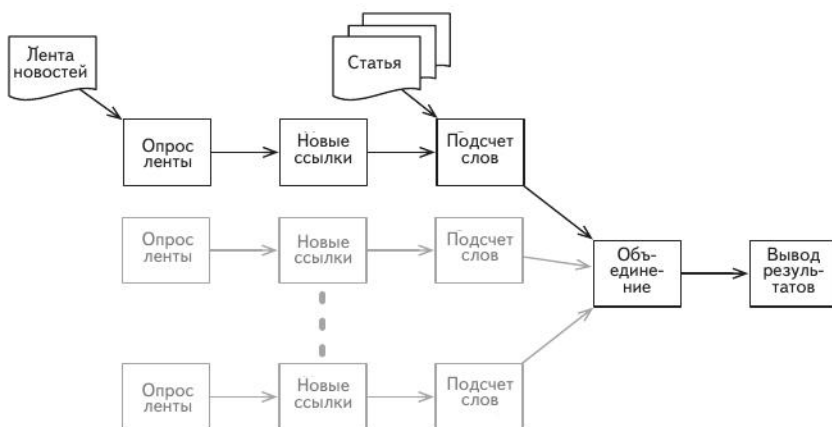


Рис. 21. Структура программы чтения RSS

Сначала посмотрим, как интегрировать существующую библиотеку асинхронного ввода/вывода с `core.async`.

От функций обратного вызова к каналам

В нашей реализации мы будем использовать библиотеку `http-kit`.¹³ Подобно многим библиотекам поддержки асинхронного ввода/выво-

¹³ <http://http-kit.org>

да, http-kit сообщает о завершении операции вызовом функции обратного вызова:

```
wordcount.core=> (require '[org.httpkit.client :as http])
nil
wordcount.core=> (defn handle-response [response]
  #_=> (let [url (get-in response [:opts :url])
            status (:status response)]
        (println "Fetched:" url "with status:" status)))
#'wordcount.core/handle-response
wordcount.core=> (http/get "http://paulbutcher.com/" handle-response)
#<core$promise$reify__6310@3a9280d0: :pending>
wordcount.core=>
Fetched: http://paulbutcher.com/ with status: 200
```

Наша первая цель – объединить http-kit с core.async, создав обертку для http/get. Для этого будет использоваться функция put!, с которой мы еще не встречались. Эта функция не должна вызываться внутри блока go и реализует запись в канал по принципу «вызвал и забыл» (она никогда не блокирует и не приостанавливает задачу, вызвавшую ее):

CSP/WordCount/src/wordcount/http.clj¹⁴

```
(defn http-get [url]
  (let [ch (chan)]
    (http/get url (fn [response]
                    (if (= 200 (:status response))
                        (put! ch response)
                        (do (report-error response) (close! ch)))))
    ch))
```

Функция-обертка сначала создает канал, возвращаемый в конце (этот прием уже должен быть знаком вам), и затем вызывает http/get, которая тут же возвращает управление. В некоторый момент времени, в будущем, когда операция GET завершится, автоматически будет вызвана наша функция-обработчик. Если код состояния равен 200 (запрос выполнен успешно), обработчик просто запишет ответ response в канал. Если код состояния имеет какое-то другое значение, обработчик сообщит об ошибке и закроет канал.

Далее напишем функцию, опрашивающую ленту новостей RSS.

¹⁴ <http://media.pragprog.com/titles/pb7con/code/CSP/WordCount/src/wordcount/http.clj>

Опрос ленты новостей

Как вы наверняка догадались, имея функции `http-get` и `poll`, реализовать опрос ленты новостей RSS не составляет никакого труда:

CSP/WordCount/src/wordcount/feed.clj¹⁵

```
(def poll-interval 60)

; Простейшая функция опроса ленты новостей
; ВНИМАНИЕ: не используйте эту функцию в промышленном коде
;           (вместо нее используйте условный метод GET)

(defn poll-feed [url]
  (let [ch (chan)]
    (poll poll-interval
      (when-let [response (<! (http-get url))]
        (let [feed (parse-feed (:body response))]
          (onto-chan ch (get-links feed) false)))
      ch)))
```

Для парсинга полученных данных в формате XML функции `parse-feed` и `get-links` используют библиотеку `Rome`.¹⁶ Мы не будем рассматривать эти функции, но вы можете исследовать их, заглянув в исходный код примеров.

Список ссылок, возвращаемый функцией `get-links`, записывается в канал `ch` вызовом `onto-chan`. По умолчанию `onto-chan` закрывает канал по исчерпанию переданной ей последовательности; мы запрещаем такое поведение передачей `false` в последнем аргументе.

Ниже показано, как действует `poll-feed`:

```
wordcount.core=> (ns wordcount.feed)
nil
wordcount.feed=> (def feed (poll-feed "http://www.cbsnews.com/feeds/
rss/main.rss"))
#'wordcount.feed/feed
wordcount.feed=> (loop []
  #_=> (when-let [url (<!! feed)]
    #_=> (println url)
    #_=> (recur)))
http://www.cbsnews.com/news/three-year-old-dies-after-visit-to-
dentist-in-hawaii/
http://www.cbsnews.com/news/obama-unemployment-benefits-
expiration-just-plain-cruel/
```

¹⁵ <http://media.pragprog.com/titles/pb7con/code/CSP/WordCount/src/wordcount/feed.clj>

¹⁶ <http://rometools.github.io/rome/>

<http://www.cbsnews.com/news/rand-paul-says-hes-suing-over-nsa-surveillance-programs/>

.
. .
.

Теперь посмотрим, как фильтровать ссылки, возвращаемые функцией `poll-feed`, чтобы избавиться от дубликатов.

Не используйте это решение

Подобная упрощенная стратегия опроса хорошо подходит для демонстрационного примера в книге, но, пожалуйста, не используйте ее в промышленном коде. Извлекать всю ленту новостей в каждом цикле опроса нет никакой необходимости, к тому же это ведет к неоправданному увеличению нагрузки на вашу систему, на сеть, а также на опрашиваемые серверы. Нагрузку можно уменьшить за счет использования условного HTTP-метода `GET`.*

* http://fishbowl.pastiche.org/2002/10/21/http_conditional_get_for_rss_hackers/

Уникальные ссылки

Функция `poll-feed` просто возвращает все ссылки, какие только найдет во время опроса ленты новостей. Среди них может оказаться множество повторяющихся ссылок. Нам же требуется, чтобы в канал попали только новые ссылки. Следующая функция делает в точности то, что нам нужно:

CSP/WordCount/src/wordcount/feed.clj¹⁷

```
(defn new-links [url]
  (let [in (poll-feed url)
        out (chan)]
    (go-loop [links #{}]
      (let [link (<! in)]
        (if (contains? links link)
            (recur links)
            (do
              (>! out link)
              (recur (conj links link))))))
    out))
```

¹⁷ <http://media.pragprog.com/titles/pb7con/code/CSP/WordCount/src/wordcount/feed.clj>

Сначала она создает два канала, `in` и `out`. Первый из них возвращается функцией `poll-feed`; второй – новый, он будет служить хранилищем для новых ссылок. Затем функция входит в цикл внутри блока `go`, наполняющий множество `links` ссылками, встречавшимися к настоящему моменту, которое первоначально инициализируется как пустое множество `#{}.` Прочитав очередную ссылку `link` из канала `in`, функция проверяет – не включена ли она во множество `links`. Если включена, функция ничего не делает; в противном случае новая ссылка записывается в канал `out` и добавляется во множество `links`.

Запустите эту функцию из интерактивной оболочки `REPL`, и вместо всего пакета ссылок, генерируемого каждые 60 секунд, вы должны увидеть только новые.

Теперь, когда у нас есть ссылки на новые статьи, на следующем шаге нужно загрузить каждую статью и подсчитать количество слов в них.

Подсчет слов

После всего, что мы видели, функция `get-counts` пишется почти сама собой:

```
CSP/WordCount/src/wordcount/core.clj18
```

```
(defn get-counts [urls]
  (let [counts (chan)]
    (go (while true
        (let [url (<! urls)]
          (when-let [response (<! (http-get url))]
            (let [c (count (get-words (:body response)))]
              (>! counts [url c]))))))
    counts))
```

Она принимает канал `urls` и для каждого URL читает содержимое соответствующей статьи с помощью `http-get`, подсчитывает слова в ней и записывает в выходной канал массив с двумя элементами – в первом элементе сохраняется URL статьи, а во втором – счетчик слов.

Мы почти закончили – осталось лишь связать все эти функции воедино.

Соединяем все вместе

Ниже приводится функция `main`, реализующая счетчик слов RSS:

¹⁸ <http://media.pragprog.com/titles/pb7con/code/CSP/WordCount/src/wordcount/core.clj>

CSP/WordCount/src/wordcount/core.clj¹⁹

```
1 (defn -main [feeds-file]
2   (with-open [rdr (io/reader feeds-file)]
3     (let [feed-urls (line-seq rdr)
4           article-urls (doall (map new-links feed-urls))
5           article-counts (doall (map get-counts article-urls))
6           counts (async/merge article-counts)]
7       (while true
8         (println (<!! counts))))))
```

Она образует программу, принимающую файл со списком адресов URL лент новостей, по одной в каждой строке. В строке 2 создается объект, осуществляющий чтение из файла (функция `with-open` гарантирует закрытие файла, когда объект чтения выйдет из области видимости). Содержимое файла преобразуется в последовательность адресов URL вызовом `line-seq` (строка 3). Операция отображения `new-links` на эту последовательность (строка 4) возвращает последовательность каналов со ссылками на новые статьи. А операция отображения `get-counts` на эту последовательность (строка 5) возвращает последовательность каналов, участвовавших в подсчете.

Наконец, вызовом `async/merge` (строка 6) эта последовательность каналов объединяется в единый канал, содержащий все, что было запущено в исходные каналы. Затем следует бесконечный цикл, в котором выводится все, что попало в объединенный канал (строка 7). Ниже показано, как работает программа:

```
$ lein run feeds.txt
[http://www.bbc.co.uk/sport/0/football/25611509 10671]
[http://www.wired.co.uk/news/archive/2014-01/04/time-travel 11188]
[http://news.sky.com/story/1190148 3488]
.
.
.
```

Когда будете опробовать этот пример у себя, обратите внимание на потребление процессора. Этот код не только выглядит очень просто и легко читается, но он еще и весьма эффективен, может следить за сотнями лент новостей, почти не потребляя вычислительную мощность CPU.

¹⁹ <http://media.pragprog.com/titles/pb7con/code/CSP/WordCount/src/wordcount/core.clj>



Вопрос Джо: почему в программе не используются буферизованные каналы?

Обратите внимание на каналы, созданные сегодня, – все они являются небуферизованными (синхронными). Начиная осваивать модель CSP (взаимодействие последовательных процессов) обычно считают, что буферизованные каналы используются чаще, чем небуферизованные, но в действительности все обстоит совсем иначе. Буферизованные каналы, безусловно, имеют свою область применения, однако всегда старайтесь тщательно все взвесить, прежде чем использовать их. Убедитесь, что буфер действительно необходим.

В завершение второго дня

Мы подошли к концу второго дня. Завтра мы посмотрим, как можно использовать `core.async` на стороне клиента, в сценариях на ClojureScript.

Что мы узнали во второй день

Каналы и блоки `go` позволяют писать эффективный, асинхронный код, который читается легко и просто и не содержит сложностей, обычно характерных для кода, наполненного функциями обратного вызова.

- Существующие прикладные интерфейсы на основе функций обратного вызова можно перенести в асинхронный мир, реализовав минимальную функцию обратного вызова, которая просто записывает результаты в канал;
- макрос `alt!` дает возможность одновременно читать или записывать данные из/во множество каналов;
- функция `timeout` возвращает канал, который закрывается спустя указанный промежуток времени, позволяя интерпретировать тайм-ауты как самые обычные сущности;
- вызовы функций, осуществляющих приостановку, должны производиться непосредственно в блоке `go`. Макросы в языке Clojure можно использовать для встраивания кода, и с их помощью можно разбивать большие блоки `go`, не нарушая этого ограничения.

Упражнения ко второму дню

Найдите:

- помимо макроса `alt!`, в составе `core.async` имеется также макрос `alts!`. Чем они отличаются? В каких случаях используется один, а в каких – другой?
- помимо `async/merge`, библиотека `core.async` предоставляет еще несколько способов объединения множества каналов. Найдите документацию с описанием `pub`, `sub`, `mult`, `tap`, `mix` и `admix`. В каких ситуациях могут пригодиться эти функции?

Задания:

- поэкспериментируйте с программой чтения RSS и выясните, в каком порядке выполняются операции в ней. Обратите внимание, что благодаря применению небуферизованных каналов результат очень похож на потоковые программы, когда блоки `go`, определенные ближе к началу конвейера, поставляют результаты для использования в последующих.

Что случится, если использовать буферизованные каналы? Дают ли они хоть какие-то преимущества? Какие проблемы могут возникнуть с буферизованными каналами?

- реализуйте собственную версию `async/merge`. Не забудьте обработать ситуацию, когда один или несколько исходных каналов окажутся закрытыми. (Подсказка: возможно, решение покажется проще, если вместо `alt!` использовать `alts!`);
- с помощью средств развертывания макросов в языке Clojure исследуйте макрос `alt!`:

```
channels.core=> (macroexpand-1 `(alt! ch1 ([x] (println x))
ch2 ([y] (println y))))
```

Вероятно, вам будет проще, если предварительно отформатировать полученный код и удалить префиксы `clojure.core`. Смогли ли вы понять, как `alt!` реализует эффект, напоминающий вызов анонимной функции?

День 3: модель CSP на стороне клиента

Язык ClojureScript – это версия Clojure, программный код на котором не компилируется в байт-код Java, а преобразуется в программный код на JavaScript (см. <http://clojurescript.com>). Это означает возмож-

ность писать веб-приложения, обе части которых – серверная и клиентская – на языке Clojure.

Одной из самых веских причин использовать ClojureScript является поддержка `core.async`, дающая массу преимуществ, исследованием которых мы займемся сегодня, не самым маловажным из которых является избавление от проклятия разработчиков на JavaScript – ада функций обратного вызова.

Конкуренция – это образ жизни

Имеющие сколько-нибудь серьезный опыт программирования клиентских сценариев на JavaScript наверняка скептически отнесутся к моим словам – движки JavaScript в браузерах являются однопоточными, о каких выгодах применения `core.async` тогда может идти речь? Разве для конкурентного программирования не требуется иметь многопоточную среду выполнения?

Волшебство инверсии управления, которое способен творить макрос `go`, помогает языку ClojureScript *выглядеть* многопоточной средой выполнения даже в отсутствие истинной многопоточности. Речь идет о своеобразной форме *кооперативной многозадачности* – никакая задача не может быть принудительно прервана другой задачей. Как мы увидим далее, применение принципов кооперативной многозадачности позволяет существенно улучшить структуру и ясность кода.



Вопрос Джо: а как же технология Web Workers?

Последние версии браузеров поддерживают ограниченную форму истинной многопоточности в JavaScript в виде веб-потоков.* Однако технология Web Workers предназначена только для выполнения фоновых задач, не имеющих доступа к дереву DOM.

Веб-потоки могут использоваться в сценариях на ClojureScript посредством, например, библиотеки Servant.**

* <http://www.whatwg.org/specs/web-apps/current-work/multipage/workers.html>

** <https://github.com/MarcoPolo/servant>

Привет, ClojureScript

Язык ClojureScript очень похож на Clojure, но имеет некоторые отличия – я буду упоминать их по мере того, как они будут встречаться.

Компиляция типичного приложения на ClojureScript выполняется в два этапа. На первом этапе клиентский сценарий на ClojureScript компилируется в программный код на JavaScript, а затем компилируется серверный код, который в процессе работы будет создавать страницы, включающие код на JavaScript в виде тегов `<script>`. Для автоматизации сборки всех сегодняшних примеров мы будем использовать расширение `lein-cljsbuild`.²⁰ Серверный код будет располагаться в каталоге `src-clj`, а клиентский код – в каталоге `src-cljs`.

Рассмотрим пример простого проекта, включающего единственную страницу с единственным сценарием. Ниже приводится исходный код самой страницы:

CSP/HelloClojureScript/resources/public/index.html²¹

```
1 <html>
2   <head>
3     <title>Hello ClojureScript</title>
4     <script src="/js/main.js" type="text/javascript"></script>
5   </head>
6   <body>
7     <div id="content">
8   </div>
9 </body>
10 </html>
```

Сгенерированный сценарий на JavaScript подключается в строке 4. Этот сценарий будет заполнять пустой элемент `<div>` в строке 7. Ниже приводится исходный код сценария на ClojureScript:

CSP/HelloClojureScript/src-cljs/hello_clojurescript/core.cljs²²

```
1 (ns hello-clojurescript.core
-   (:require-macros [cljs.core.async.macros :refer [go]])
-   (:require [goog.dom :as dom]
-             [cljs.core.async :refer [<! timeout]])
5
-   (defn output [elem message]
-     (dom/append elem message (dom/createDom "br")))
-   (defn start []
-     (let [content (dom/getElement "content")]
10      (go
```

²⁰ <https://github.com/emezeske/lein-cljsbuild>

²¹ <http://media.pragprog.com/titles/pb7con/code/CSP/HelloClojureScript/resources/public/index.html>

²² http://media.pragprog.com/titles/pb7con/code/CSP/HelloClojureScript/src-cljs/hello_clojurescript/core.cljs

```

-      (while true
-        (<! (timeout 1000))
-        (output content "Hello from task 1")))
-    (go
15    (while true
-      (<! (timeout 1500))
-      (output content "Hello from task 2")))))
-
- (set! (.onload js/window) start)

```

Первое отличие между Clojure и ClojureScript заключается в необходимости предварительного объявления всех используемых макросов с помощью `:require-macros` (строка 2). Функция `output` в строке 6 использует библиотеку Google Closure (обратите внимание на написание: вместо символа *j* используется символ *s*) для добавления сообщения в конец элемента DOM.²³

Эта функция вызывается в строках 13 и 17, причем оба вызова производятся в независимо выполняющихся блоках `go`. Первый из них выводит сообщение каждую секунду, а второй – каждые полторы секунды.

Наконец, в строке 19 определяется, что наша функция `start` должна запускаться после загрузки страницы, путем присваивания ее атрибуту `onload` JavaScript-объекта `window`. Здесь используется *специальная форма «точка»* (`.`), существующая в языке ClojureScript, обеспечивающая взаимодействие с JavaScript. Выражение

```
(set! (.onload js/window) start)
```

преобразуется в код на JavaScript:

```
window.onload = hello_clojurescript.core.start;
```

Мы не будем рассматривать серверный код, поскольку он очень прост (см. исходный код в загружаемом пакете примеров для книги).

Скомпилируйте сценарий командой `lein cljbuild once`, запустите сервер, выполнив `lein run`, и откройте в браузере страницу по адресу `http://localhost:3000`. Вы должны увидеть нечто подобное:

```

Hello from task 1
Hello from task 2
Hello from task 1
Hello from task 1
Hello from task 2
.

```

²³ <https://developers.google.com/closure/library/>

Кто сказал, что для конкуренции нужны потоки выполнения?

Задачи, выполняющиеся параллельно, — само по себе впечатляющее зрелище, но большинство пользовательских интерфейсов должно также взаимодействовать с человеком, а это означает необходимость обработки событий, поэтому далее мы посмотрим, как это делается.

Обработка событий

Мы рассмотрим приемы обработки событий в сценариях на ClojureScript на примере создания простой анимации, реагирующей на щелчки мышью. Для этого создадим веб-страницу, отображающую окружности после щелчка, которые постепенно сжимаются и исчезают, как показано на рис. 22.

Код разметки страницы очень прост, он содержит единственный элемент `<div>`, заполняющий все окно:

```
CSP/Animation/resources/public/index.html24
```

```
<html>
<head>
  <title>Animation</title>
  <script src="/js/main.js" type="text/javascript"></script>
</head>
<body>
  <div id="canvas" width="100%" height="100%"></div>
</body>
</html>
```

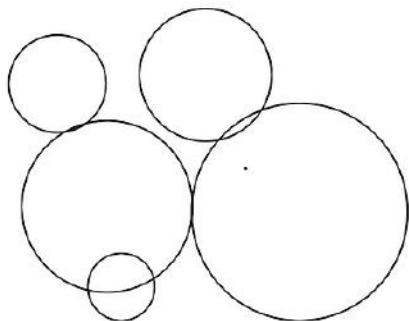


Рис. 22. Сжимающиеся окружности

²⁴ <http://media.pragprog.com/titles/pb7con/code/CSP/Animation/resources/public/index.html>

Чтобы получить возможность рисовать графику на странице, нам потребуется поддержка графики из библиотеки Google Closure, которая эффективно скрывает различия между браузерами. Функция `create-graphics` принимает элемент DOM и возвращает объект, который можно использовать как поверхность для рисования:

CSP/Animation/src-cljs/animation/core.cljs²⁵

```
(defn create-graphics [elem]
  (doto (graphics/createGraphics "100%" "100%")
    (.render elem)))
```

Следующая функция, `shrinking-circle`, принимает такой объект с координатами рисования, и создает блок `go`, управляющий анимацией окружности с центром в указанных координатах:

CSP/Animation/src-cljs/animation/core.cljs²⁶

```
1 (def stroke (graphics/Stroke. 1 "#ff0000"))
-
- (defn shrinking-circle [graphics x y]
-   (go
5     (let [circle (.drawCircle graphics x y 100 stroke nil)]
-       (loop [r 100]
-         (<! (timeout 25))
-         (.setRadius circle r r)
-         (when (> r 0)
10          (recur (dec r))))
-       (.dispose circle))))
```

Сначала она рисует окружность, вызовом функции `drawCircle` из библиотеки Google Closure (строка 5), а затем входит в цикл с таймаутом 25 миллисекунд (строка 7), чтобы обеспечить вызов `setRadius` с частотой 40 раз в секунду. Наконец, когда радиус уменьшается до нуля, окружность удаляется (строка 11).

Теперь нам нужно что-то, что позволит определить момент щелчка мышью на странице. В библиотеке Google Closure имеется функция `listen`, с помощью которой можно регистрировать обработчики событий.

Так же как функция `http/get`, которую мы видели вчера, она принимает функцию обратного вызова, реализующую обработку собы-

²⁵ <http://media.pragprog.com/titles/pb7con/code/CSP/Animation/src-cljs/animation/core.cljs>

²⁶ <http://media.pragprog.com/titles/pb7con/code/CSP/Animation/src-cljs/animation/core.cljs>

тия. Так же как и вчера, наш обработчик будет просто записывать события в канал:

```
CSP/Animation/src-cljs/animation/core.cljs27
```

```
(defn get-events [elem event-type]
  (let [ch (chan)]
    (events/listen elem event-type
      # (put! ch %))
    ch))
```

Теперь у нас есть все необходимое для конструирования сценария:

```
CSP/Animation/src-cljs/animation/core.cljs28
```

```
(defn start []
  (let [canvas (dom/getElement "canvas")
        graphics (create-graphics canvas)
        clicks (get-events canvas "click")]
    (go (while true
        (let [click (<! clicks)
              x (.-offsetX click)
              y (.-offsetY click)]
          (shrinking-circle graphics x y))))))

(set! (.-onload js/window) start)
```

Сначала наш сценарий находит элемент `<div>`, который будет использоваться в качестве холста, создает объект для отображения графики и получает канал с событиями щелчка мышью. Затем он входит в цикл, ожидающий появления событий, извлекает координаты указателя в момент щелчка, обращением к `offsetX` и `offsetY`, и создает анимированную окружность в этой позиции.

Реализация выглядит удивительно простой (впрочем, она таковой и является) — перейдя из мира функций-обработчиков JavaScript в мир `core.async`, мы избавились от кромешного ада функций обратного вызова.

Усмирение функций обратного вызова

Термин «ад функций обратного вызова» был придуман, чтобы описать программный код, который получается из-за повсеместного

²⁷ <http://media.pragprog.com/titles/pb7con/code/CSP/Animation/src-cljs/animation/core.cljs>

²⁸ <http://media.pragprog.com/titles/pb7con/code/CSP/Animation/src-cljs/animation/core.cljs>

использования функций обратного вызова в JavaScript, – одни функции-обработчики вызывают другие функции-обработчики, используя при этом различные элементы состояния для взаимодействия друг с другом.

Переход к асинхронной модели программирования дает возможность уйти от этого, как показано далее.

Отправляемся в путь, чтобы увидеть Мастера

Мастер – это распространенный шаблон организации пользовательского интерфейса, когда для достижения цели пользователь выполняет некоторую последовательность шагов. Последнее, что мы сделаем сегодня, – создадим мастера (см. рис. 23) без применения функций обратного вызова, воспользовавшись всеми имеющимися у нас знаниями.

Рис. 23. Пользовательский интерфейс мастера

Наш мастер состоит из формы, включающей несколько наборов полей:

CSP/Wizard/resources/public/index.html²⁹

```
<form id="wizard" action="/wizard" method="post">
  <fieldset class="step" id="step1">
    <legend>Step 1</legend>
    <label>First Name:</label><input type="text" name="firstname" />
    <label>Last Name:</label><input type="text" name="lastname" />
  </fieldset>

  <fieldset class="step" id="step2">
```

²⁹ <http://media.pragprog.com/titles/pb7con/code/CSP/Wizard/resources/public/index.html>

```

<legend>Step 2</legend>
<label>Date of Birth:</label><input type="date" name="dob" />
<label>Homepage:</label><input type="url" name="url" />
</fieldset>

<fieldset class="step" id="step3">
  <legend>Step 3</legend>
  <label>Password:</label><input type="password" name="pass1" />
  <label>Confirm Password:</label><input type="password" name="pass2" />
</fieldset>
<input type="button" id="next" value="Next" />
</form>

```

Каждый элемент `<fieldset>` представляет один шаг. Сначала объявим всех их скрытыми:

CSP/Wizard/resources/public/styles.css³⁰

```

label { display:block; width:8em; clear:left; float:left;
        text-align:right; margin-right: 3pt; }
input { display:block; }
> .step { display:none; }

```

В нашем сценарии используются следующие вспомогательные функции, скрывающие и отображающие указанные наборы полей:

CSP/Wizard/src-cljs/wizard/core.cljs³¹

```

(defn show [elem]
  (set! (. elem -style -display) "block"))

(defn hide [elem]
  (set! (. elem -style -display) "none"))

(defn set-value [elem value]
  (set! (. elem -value) value))

```

Все они используют специальную форму «точка» (`.`), позволяющую обращаться к атрибутам элементов. Например, выражение

```
(set! (. elem -style -display) "block")
```

будет преобразовано в следующий код на JavaScript:

```
elem.style.display = "block";
```

³⁰ <http://media.pragprog.com/titles/pb7con/code/CSP/Wizard/resources/public/styles.css>

³¹ <http://media.pragprog.com/titles/pb7con/code/CSP/Wizard/src-cljs/wizard/core.cljs>

Следующий фрагмент реализует поток управления самого мастера:

CSP/Wizard/src-cljs/wizard/core.cljs³²

```
1 (defn start []
-   (go
-     (let [wizard (dom/getElement "wizard")
-           step1 (dom/getElement "step1")
-           step2 (dom/getElement "step2")
-           step3 (dom/getElement "step3")
-           next-button (dom/getElement "next")
-           next-clicks (get-events next-button "click")]
-       (show step1)
10      (<! next-clicks)
-       (hide step1)
-       (show step2)
-       (<! next-clicks)
-       (set-value next-button "Finish")
15      (hide step2)
-       (show step3)
-       (<! next-clicks)
-       (.submit wizard))))
-
20 (set! (.onload js/window) start)
```

Здесь сначала извлекаются ссылки на все элементы формы, и вызовом функции `get-events`, написанной нами ранее, приобретается канал для событий щелчка мышью на кнопке **Next** (Далее) (строка 8). Затем этот код просто выводит набор полей, соответствующий первому шагу, и ожидает, пока пользователь щелкнет на кнопке **Next** (Далее) (строка 10). После щелчка код скрывает набор полей для шага 1, отображает набор полей для шага 2 и снова ожидает щелчка на кнопке **Next** (Далее). Так продолжается, пока не будут пройдены все шаги, после чего осуществляется отправка формы на сервер (строка 18).

Самое примечательное в этом коде — то, что он не отличается ничем особенным — простая, линейная последовательность шагов, которая и читается, собственно, как простая, линейная последовательность шагов. Конечно, зная магическую сущность макроса `go`, мы понимаем, что в действительности это не так, — мы фактически создали конечный автомат, который действует, когда в состоянии сделать это, и приостанавливается, когда ждет команды выполнить переход к следующему состоянию. Но мы легко можем игнорировать этот факт

³² <http://media.pragprog.com/titles/pb7con/code/CSP/Wizard/src-cljs/wizard/core.cljs>

и интерпретировать данный фрагмент как линейно выполняющийся программный код.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения модели взаимодействий последовательных процессов.

Что мы узнали в третий день

Язык ClojureScript – это версия языка Clojure, исходный код на котором компилируется в исходный код JavaScript и позволяет использовать мощь `core.async` в разработке клиентских сценариев. ClojureScript не только дает возможность привнести поддержку кооперативной многозадачности в однопоточный мир JavaScript, но также позволяет уйти от проблем, связанных с переплетением функций обратного вызова.

Упражнения к третьему дню

Найдите:

- как известно, реализация `core.async` для ClojureScript поддерживает операции, осуществляющие приостановку, такие как `<!` и `>!`, но не их блокирующие версии `<!!` или `>!!`. Почему?
- документацию с описанием `take!` – как можно было бы с помощью этой функции преобразовать API на основе каналов в API на основе функций обратного вызова? В каких случаях это может пригодиться? (Подсказка: это может быть связано с предыдущим вопросом).

Задания:

- используя `core.async`, напишите простую игру для браузера, такую как «Питон», «Теннис» или «Арконоид»;
- реализуйте JavaScript-версию мастера, созданного нами выше. Какая из версий кажется вам предпочтительнее – на JavaScript или на ClojureScript?

В завершение

Внешне программы, реализованные на основе моделей акторов и CSP, выглядят похожими – они состоят из множества независимых задач, выполняющихся конкурентно и взаимодействующих друг с другом посредством передачи сообщений. Но, как мы увидели в этой

главе, в центре внимания этих моделей находятся совершенно разные аспекты.

Сильные стороны

Основным достоинством модели CSP в сравнении с акторами является гибкость. В программах, использующих акторы, передача сообщений тесно связана с единицами выполнения – каждый актор имеет собственный почтовый ящик. В программах, реализующих модель CSP, напротив, каналы являются самыми обычными сущностями: они могут создаваться независимо от других компонентов программы, свободно передаваться между задачами, и любая задача может записывать в них данные и читать их оттуда.

Рич Хикки (Rich Hickey), создатель языка Clojure, объясняет причины отказа от модели акторов в пользу CSP так:³³

Я не испытываю симпатии к акторам. Они слишком тесно связывают производителя с потребителем. Да, с помощью акторов можно реализовать или имитировать некоторые виды очередей (что многие часто и делают), но, так как любая реализация акторов уже включает очереди, создается ощущение, что очереди являются наиболее простым механизмом.

Если смотреть с более прагматичной точки зрения, современные реализации CSP, такие как `core.async`, использующие прием инверсии управления для поддержки асинхронных задач, не только помогают повысить эффективность, но и существенно упрощают модель программирования в прикладных областях, традиционно опиравшихся на функции обратного вызова. Мы видели две такие области – асинхронный ввод/вывод и пользовательские интерфейсы, но существует масса других.

Слабые стороны

Если сравнить эту главу с предыдущей, посвященной акторам, можно заметить, что в ней отсутствует упоминание двух важнейших тем – распределенного программирования и отказоустойчивости. Теоретически нет ничего, что мешало бы реализовать и то, и другое, однако так сложилось, что вопросы распределенного программирования и отказоустойчивости в модели CSP не получили такого же внимания, поскольку любой желающий мог воспользоваться языками с

³³ <http://clojure.com/blog/2013/06/28/clojure-core-async-channels.html>

поддержкой акторов – в мире не существует эквивалента OTP для поддержки CSP.

Так же как потоки выполнения с блокировками и акторы, программы на основе модели CSP подвержены опасности взаимоблокировок и не имеют прямой поддержки параллелизма. Параллельные решения приходится строить из конкурентных строительных блоков, увеличивая фактор недетерминизма.

Другие языки

Подобно акторам, модель CSP появилась где-то в начале 1970-х, когда впервые была представлена Тони Хоаром (Tony Hoare). Эти две модели развивались практически параллельно, многое позаимствовав друг у друга за прошедшие годы.

В 1980-х модель CSP легла в основу языка оссам (на котором ваш покорный слуга съел собаку, занимаясь параллельным программированием),³⁴ но, вне всяких сомнений, наибольший вклад в популяризацию этой модели внес не так давно появившийся язык Go.

Применение приема инверсии управления для реализации асинхронных задач, используемого в библиотеке `core.async` и в языке Go, получило еще более широкое распространение благодаря его заимствованию другими языками программирования, такими как F#,³⁵ C#,³⁶ Nemerle,³⁷ и Scala.³⁸

Напоследок

Различия между моделями акторов и CSP в основном объясняются разными взглядами их разработчиков. Разработчики модели акторов сосредоточились на обеспечении высокой отказоустойчивости и распределенном программировании, а разработчики модели CSP – на эффективности и выразительности. Соответственно, выбор между этими моделями в значительной степени зависит от того, какие аспекты являются наиболее важными для вас.

CSP была последней универсальной моделью программирования в этой книге. В следующей главе мы займемся исследованием первой специализированной модели.

³⁴ [https://ru.wikipedia.org/wiki/Оккам_\(язык_программирования\)](https://ru.wikipedia.org/wiki/Оккам_(язык_программирования))

³⁵ <http://blogs.msdn.com/b/dsyme/archive/2007/10/11/introducing-f-asynchronous-workflows.aspx>

³⁶ <http://msdn.microsoft.com/ru-ru/library/hh191443.aspx>

³⁷ <https://github.com/rsdn/nemerle/wiki/Computation-Expression-macro#async>

³⁸ <http://docs.scala-lang.org/sips/pending/async.html>



ГЛАВА 7.

Параллелизм данных

Параллелизм данных можно сравнить с восьмиполосным шоссе. Даже при том, что каждый конкретный автомобиль едет с относительно невысокой скоростью, число машин, пересекающих определенную точку, огромно, потому что автомобили могут ехать бок о бок.

Все подходы, обсуждавшиеся до сих пор, могут использоваться для самых разных проблем программирования. Но модель параллелизма данных, напротив, имеет гораздо более узкую область применения. Как следует из названия модели, она определяет приемы не конкурентного, а параллельного программирования (не забывайте, что конкуренция и параллелизм, несмотря на некоторое сходство, суть есть разные вещи – см. раздел «Конкуренция или параллелизм» в главе 1).

В недрах вашего ноутбука спрятан суперкомпьютер

В этой главе я покажу, как использовать суперкомпьютер, скрытый в недрах вашего ноутбука, – графический процессор, или GPU. Современные GPU являются очень мощными процессорами, поддерживающими параллельную обработку данных, и способны затмить обычные процессоры (CPU) в вычислительных задачах. Свод приемов организации вычислений на GPU часто называют *программированием универсальных вычислений на GPU* (general-purpose computing on the GPU, GPGPU).

За последние годы появилось множество технологий, служащих фасадом, скрывающим за собой тонкости реализации GPU. Для разработки кода GPGPU мы будем пользоваться одной из таких технологий – Open Computing Language, или OpenCL.¹

В первый день мы познакомимся с основами устройства ядра OpenCL вместе с программой, компилирующей и выполняющей его.

¹ <http://www.khronos.org/opencv/>

Во второй день мы подробнее остановимся на том, как это ядро отображается на аппаратную часть. Наконец, в третий день мы увидим, как OpenCL взаимодействует с графическим кодом, написанным с применением библиотеки Open Graphics Library, или OpenGL.²

День 1: программирование GPGPU

Сегодня мы увидим, как создать простенькую программу GPGPU, параллельно перемножающую элементы двух массивов, и проверим, насколько быстрее действует GPU в сравнении с CPU. Но перед этим мы потратим немного времени, чтобы понять, что делает GPU таким мощным инструментом для решения вычислительных задач.

Обработка данных и параллелизм данных

Работа с компьютерной графикой практически всегда связана с обработкой данных – огромных объемов данных. И эта обработка должна выполняться быстро. Сцена в трехмерной игре конструируется из миллионов маленьких треугольников, для каждого из которых требуется вычислить его местоположение на экране с учетом перспективы относительно точки зрения, отсечения границ, освещения и наложения текстур, 25 или более раз в секунду.

Самое интересное в том, что, несмотря на огромный объем обрабатываемых данных, над ними выполняются относительно простые векторные и матричные операции. Благодаря этому появляется возможность распараллеливания обработки, то есть выполнения одних и тех же операций с разными фрагментами данных параллельно.

Современные графические процессоры – это исключительно сложные и мощные параллельные процессоры, способные отображать миллиарды треугольников в секунду. Но даже при том, что они изначально создавались для обработки графики, со временем они развились до такого состояния, что стало возможным использовать их для решения более широкого круга задач.

Параллелизм данных можно реализовать множеством разных способов. Чуть ниже мы рассмотрим два из них: конвейерную обработку и применение множества ALU (Arithmetic Logic Unit – арифметико-логических устройств).

² <http://www.opengl.org>

Конвейерная обработка

Мы привыкли считать, что умножение двух чисел является единственной и атомарной операцией. Но если опуститься на уровень кристалла микропроцессора, можно обнаружить, что в действительности она выполняется в несколько этапов. Обычно эти этапы выстраиваются в виде конвейера, как показано на рис. 24.

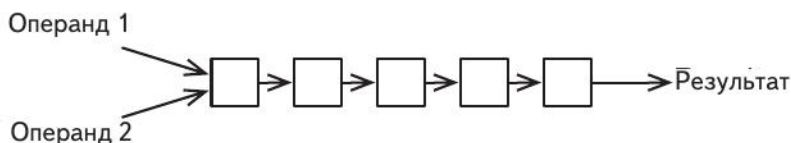


Рис. 24. Обычно этапы выстраиваются в виде конвейера

Если предположить, что для выполнения каждого из пяти этапов конвейера, изображенного на рис. 24, требуется один такт процессорного времени, тогда умножение пары чисел будет выполняться в течение пяти тактов. Но для большого количества чисел производительность может оказаться немного выше, потому что (если допустить, что чтение данных из памяти выполняется достаточно быстро) есть возможность своевременно пополнять конвейер новыми данными, как показано на рис. 25.

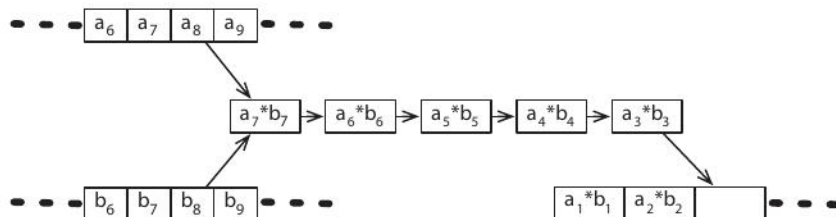


Рис. 25. При наличии большого количества чисел можно своевременно пополнять конвейер новыми данными

То есть умножение тысячи пар чисел займет, конечно же, больше тысячи тактов, но уже не пять тысяч, как можно было бы подумать, исходя из факта, что умножение одной пары выполняется за пять тактов.

Применение множества ALU

Часть процессора, отвечающая за выполнение различных операций, таких как умножение, называется *арифметико-логическим устройством* (Arithmetic Logic Unit, ALU).

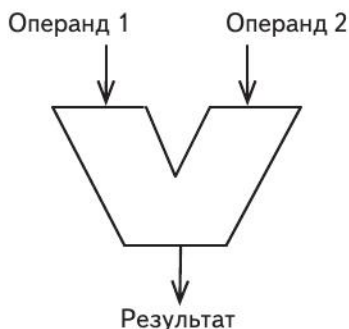


Рис. 26. Арифметико-логическое устройство

Достаточно связать несколько ALU с памятью широкой шиной данных, позволяющей одновременно извлекать несколько операндов, и операции на больших объемах данных можно распараллелить, как показано на рис. 27.

Графические процессоры обычно связаны с памятью 256-разрядной или даже более широкой шиной, что дает возможность (например) одновременно извлекать восемь или более 32-разрядных вещественных чисел.

Запутанная картина

Для достижения высокой производительности в современных GPU используются и конвейерная обработка, и множество ALU, и еще целое множество других приемов, которые мы не будем упоминать здесь. Мы могли бы разобраться во всех тонкостях единственного GPU. Но, к сожалению, разные GPU (даже выпущенные одним и тем же производителем) имеют слишком мало общего. Если бы нам приходилось писать код для какой-то конкретной архитектуры, идея программирования GPGPU потерпела бы сокрушительное фиаско.

OpenCL поддерживает множество архитектур, определяя C-подобный язык, позволяющий абстрактно выражать параллельные алгоритмы. Для каждой отдельной модели GPU производитель предоставляет собственные компиляторы и драйверы, с помощью которых программы на этом языке можно компилировать и выполнять на аппаратном обеспечении данного производителя.

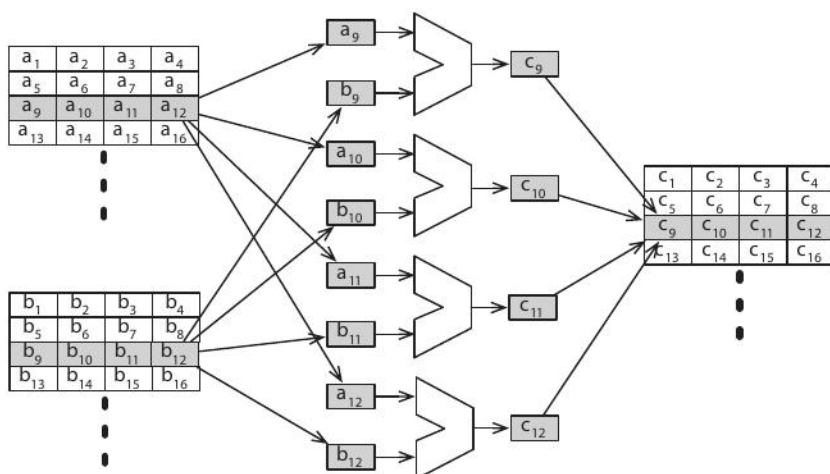


Рис. 27. Параллельная обработка больших объемов данных с использованием нескольких ALU

Наша первая программа для OpenCL

Чтобы распараллелить нашу задачу перемножения элементов двух массивов с помощью OpenCL, ее необходимо разделить на блоки, которые затем будут выполняться параллельно.

Рабочие элементы

Если вам уже приходилось писать параллельный код, значит, вы знакомы с проблемой дробления задач на слишком мелкие подзадачи. Как правило, слишком мелкое дробление ухудшает производительность, потому что приводит к увеличению накладных расходов на создание потоков выполнения и взаимодействия между ними.

В OpenCL, напротив, задачи обычно делятся на очень мелкие подзадачи, или *рабочие элементы* (work-items). Например, чтобы перемножить два массива по 1024 элемента в каждом, можно создать 1024 рабочих элемента (см. рис. 28).

Ваша задача как программиста состоит в том, чтобы разделить задачу на как можно более мелкие рабочие элементы, а компилятор и среда выполнения OpenCL сами позаботятся о том, как лучше спланировать их выполнение на имеющемся аппаратном обеспечении, чтобы максимально эффективно использовать его возможности.



Рис. 28. Деление задачи перемножения элементов двух массивов на рабочие элементы

Оптимизация OpenCL

Полагаю, вы не удивитесь, если я скажу, что в действительности дела обстоят намного сложнее. Для оптимизации приложений на OpenCL часто приходится задумываться о доступных ресурсах и подсказывать компилятору и среде выполнения, как лучше спланировать выполнение рабочих элементов. Иногда для большей эффективности требуется даже ограничивать параллелизм.

Однако, как это обычно бывает, преждевременная оптимизация и в данном случае является корнем проблем. В большинстве случаев предпочтительнее реализовать максимальное распараллеливание, разбив задачу на самые мелкие рабочие элементы, какие только возможны, и только потом задумываться об оптимизации.

Ядра

Порядок выполнения каждого рабочего элемента определяется *ядром* (kernel) OpenCL. Ниже приводится объявление ядра, реализующего рабочий элемент:

```
DataParallelism/MultiplyArrays/multiply_arrays.cl3
```

```
__kernel void multiply_arrays(__global const float* inputA,
```

³ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.cl

```
__global const float* inputB,  
__global float* output) {  
  
    int i = get_global_id(0);  
    output[i] = inputA[i] * inputB[i];  
}
```

Это ядро принимает указатели на два исходных массива, `inputA` и `inputB`, и на массив результатов `output`. Вызывает `get_global_id()`, чтобы определить порядковый номер рабочего элемента, и затем просто записывает результат умножения элементов массивов `inputA` и `inputB` в соответствующий элемент массива `output`.

Чтобы получить законченную программу, нужно внедрить наше ядро в основную программу, которая должна выполнять следующие действия:

1. Создать контекст, в котором будет выполняться ядро вместе с очередью команд.
2. Скомпилировать ядро.
3. Создать буферы для входных и выходных данных.
4. Добавить в очередь команду, которая выполнит ядро один раз для каждого рабочего элемента.
5. Извлечь результаты.

Стандарт OpenCL определяет прикладной интерфейс реализации для языков C и C++. Однако для многих основных языков существуют неофициальные реализации, благодаря чему вы сможете писать программы на любом языке. Мы пока будем придерживаться языка C, потому что он поддерживается стандартом OpenCL и позволяет лучше понять внутреннее устройство. Но в третий день я покажу вам программу, написанную на Java.

В следующих нескольких разделах мы сконструируем полноценную программу. Чтобы сделать структуру этой программы максимально прозрачной, я исключил из нее обработку ошибок, но это не причина для волнений – мы еще вернемся к этому вопросу. Обратите также внимание, что во многих случаях в качестве аргументов функциям передаются значения `NULL`. Это обстоятельство также не должно вызывать беспокойство – позднее мы займемся более детальными исследованиями API, когда будем иметь более полное представление.

Создание контекста

Под контекстом OpenCL понимается окружение, в котором будут выполняться ядра OpenCL. Чтобы создать контекст, сначала нужно определить платформу и устройства на этой платформе, на которых

будет выполняться ядро (подробнее о платформах и устройствах мы поговорим ниже):

```
DataParallelism/MultiplyArrays/multiply_arrays.c4
```

```
cl_platform_id platform;  
clGetPlatformIDs(1, &platform, NULL);  
  
cl_device_id device;  
clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, NULL);  
  
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
```

Нам нужен простой контекст, включающий единственный GPU, поэтому после определения платформы вызовом `clGetPlatformIDs()` мы передаем `CL_DEVICE_TYPE_GPU` в вызов `clGetDeviceIDs()`, чтобы получить идентификатор GPU. После этого создается контекст, передачей идентификатора устройства в вызов `clCreateContext()`.

Создание очереди команд

Теперь у нас есть контекст, и мы можем использовать его для создания очереди команд:

```
DataParallelism/MultiplyArrays/multiply_arrays.c5
```

```
cl_command_queue queue = clCreateCommandQueue(context, device, 0, NULL);
```

Функция `clCreateCommandQueue()` принимает контекст и устройство и возвращает очередь, позволяющую передавать команды этому устройству.

Компиляция ядра

Далее нужно скомпилировать ядро в двоичный код, который будет выполняться на устройстве:

```
DataParallelism/MultiplyArrays/multiply_arrays.c6
```

```
char* source = read_source("multiply_arrays.cl");  
cl_program program = clCreateProgramWithSource(context, 1,
```

⁴ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

⁵ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

⁶ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

```
(const char**) &source, NULL, NULL);
free(source);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
cl_kernel kernel = clCreateKernel(program, "multiply_arrays", NULL);
```

Здесь сначала исходный код ядра загружается из файла `multiply_arrays.cl` в строку `source` (реализацию функции `read_source()` можно найти в загружаемом пакете примеров для книги) и затем передается в `clCreateProgramWithSource()`. Далее вызовом `clBuildProgram()` выполняется сборка программы, которая затем с помощью `clCreateKernel()` преобразуется в ядро.

Создание буферов

Ядра работают с данными, размещенными в буферах:

`DataParallelism/MultiplyArrays/multiply_arrays.c7`

```
#define NUM_ELEMENTS 1024

cl_float a[NUM_ELEMENTS], b[NUM_ELEMENTS];
random_fill(a, NUM_ELEMENTS);
random_fill(b, NUM_ELEMENTS);
cl_mem inputA = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float) * NUM_ELEMENTS, a, NULL);
cl_mem inputB = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
    sizeof(cl_float) * NUM_ELEMENTS, b, NULL);
cl_mem output = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
    sizeof(cl_float) * NUM_ELEMENTS, NULL, NULL);
```

Здесь создаются два массива, `a` и `b`, каждый из которых заполняется случайными значениями с помощью `random_fill()`:

`DataParallelism/MultiplyArrays/multiply_arrays.c8`

```
void random_fill(cl_float array[], size_t size) {
    for (int i = 0; i < size; ++i)
        array[i] = (cl_float)rand() / RAND_MAX;
}
```

Наши два входных буфера, `inputA` и `inputB`, с точки зрения ядра, доступны только для чтения (`CL_MEM_READ_ONLY`) и инициализируются

⁷ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

⁸ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

путем копирования исходных массивов (`CL_MEM_COPY_HOST_PTR`). Выходной буфер `output` доступен лишь для записи (`CL_MEM_WRITE_ONLY`).

Выполнение рабочих элементов

Теперь мы, наконец, имеем возможность выполнить рабочие элементы, составляющие задачу перемножения элементов массивов:

```
DataParallelism/MultiplyArrays/multiply_arrays.c9
```

```
clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &inputB);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &output);

size_t work_units = NUM_ELEMENTS;
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &work_units, NULL, 0,
NULL, NULL);
```

Здесь сначала выполняется подготовка аргументов ядра, вызовом `clSetKernelArg()`, а затем вызывается функция `clEnqueueNDRangeKernel()`, которая включает в очередь *N-мерный массив* (`NDRange`) рабочих элементов. В данном случае *N* равно 1 (третий аргумент функции `clEnqueueNDRangeKernel()` – пример, когда *N* > 1, мы увидим немного позже), а число рабочих элементов равно 1024.

Извлечение результатов

После того как ядро закончит работу, нужно извлечь полученные результаты:

```
DataParallelism/MultiplyArrays/multiply_arrays.c10
```

```
cl_float results[NUM_ELEMENTS];
clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float) * NUM_ELEMENTS,
results, 0, NULL, NULL);
```

С этой целью создается массив `results`, и вызовом функции `clEnqueueReadBuffer()` в него копируется содержимое буфера `output`.

Заключительные операции

Последний этап, который необходимо выполнить в основной программе, – освободить занятые ресурсы:

⁹ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

¹⁰ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

DataParallelism/MultiplyArrays/multiply_arrays.c¹¹

```

clReleaseMemObject(inputA);
clReleaseMemObject(inputB);
clReleaseMemObject(output);
clReleaseKernel(kernel);
clReleaseProgram(program);
clReleaseCommandQueue(queue);
clReleaseContext(context);

```

Профилирование

Теперь, получив действующее ядро, посмотрим, какой прирост производительности оно нам дает. Для этого можно воспользоваться прикладным интерфейсом OpenCL для профилирования:

DataParallelism/MultiplyArraysProfiled/multiply_arrays.c¹²

```

1 cl_event timing_event;
- size_t work_units = NUM_ELEMENTS;
- clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &work_units,
-   NULL, 0, NULL, &timing_event);
5
- cl_float results[NUM_ELEMENTS];
- clEnqueueReadBuffer(queue, output, CL_TRUE, 0, sizeof(cl_float)
* NUM_ELEMENTS,
-   results, 0, NULL, NULL);
- cl_ulong starttime;
10 clGetEventProfilingInfo(timing_event, CL_PROFILING_COMMAND_START,
-   sizeof(cl_ulong), &starttime, NULL);
- cl_ulong endtime;
- clGetEventProfilingInfo(timing_event, CL_PROFILING_COMMAND_END,
-   sizeof(cl_ulong), &endtime, NULL);
15 printf("Elapsed (GPU): %lu ns\n\n", (unsigned long)(endtime -
starttime));
- clReleaseEvent(timing_event);

```

Сначала, в строке 3, мы передаем функции `clEnqueueNDRangeKernel()` событие `timing_event`. После того как она вернет управление, можно послать событие запроса результатов хронометража вызовом `clGetEventProfilingInfo()` (строки 10 и 13).

Если определить константу `NUM_ELEMENTS` со значением 100 000, GPU на моем MacBook Pro справится с задачей примерно за 43 000

¹¹ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArrays/multiply_arrays.c

¹² http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArraysProfiled/multiply_arrays.c

наносекунд. Для сравнения попробуем выполнить те же вычисления на обычном процессоре CPU:

DataParallelism/MultiplyArraysProfiled/multiply_arrays.c¹³

```
for (int i = 0; i < NUM_ELEMENTS; ++i)
    results[i] = a[i] * b[i];
```

Перемножение тех же самых 100 000 пар элементов займет что-то около 400 000 наносекунд, то есть GPU решает поставленную задачу более чем в девять раз быстрее, чем CPU с одним ядром.

Предостережение

Профилеринг команды, выполняющей умножение двух массивов, дает не совсем корректные результаты. Дело в том, что перед выполнением команды мы скопировали исходные данные в буферы `inputA` и `inputB`, а после выполнения – извлекли результаты, скопировав их из буфера `output`.

Эти операции копирования являются достаточно дорогостоящими – для такой простой задачи, как попарное перемножение элементов массивов, они являются слишком дорогостоящими, чтобы можно было оправдать применение GPU. На практике вычисления с привлечением графического процессора применяются, только когда требуется выполнить большее число операций с операндами либо когда данные уже находятся в устройстве GPU.

Ради справедливости следует сказать, что мой пример перемножения элементов массивов не особенно осторожен в некоторых аспектах, связанных с использованием OpenCL API. Поэтому давайте исправим этот недостаток.

Множество возвращаемых значений

Многие функции OpenCL способны возвращать множество значений. Например, платформа может поддерживать несколько устройств, соответственно, `clGetDeviceIDs()` может возвращать более одного устройства. Ниже приводится прототип этой функции:

```
cl_int clGetDeviceIDs(cl_platform_id platform,
```

¹³ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArraysProfiled/multiply_arrays.c


```
cl_device_type device_type,  
cl_uint num_entries,  
cl_device_id* devices,  
cl_uint* num_devices);
```

Параметр `devices` – это указатель на массив длиной `num_entries`, а параметр `num_devices` – это указатель на единственное целое число. Один из способов заключается в вызове `clGetDeviceIDs()` с массивом фиксированной длины:

```
cl_device_id devices[8];  
cl_uint num_devices;  
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 8, devices, &num_devices);
```

После того как `clGetDeviceIDs()` вернет управление, в `num_devices` будет записано число доступных устройств, а в первые `num_devices` элементов массива `devices` будут записаны идентификаторы этих устройств.

Этот прием прекрасно работает, но как быть, если в действительности имеется более восьми устройств? Теоретически можно было бы просто передать массив «достаточно большого» размера, однако, как показывает опыт, всякий раз, когда в код «зашивается» какой-то предел, рано или поздно он будет превышен. К счастью, все функции OpenCL, возвращающие массив, дают возможность узнать размер этого массива, стоит только вызвать их дважды:

```
cl_uint num_devices;  
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);  
  
cl_device_id* devices = (cl_device_id*)malloc(sizeof(cl_device_id)  
* num_devices);  
clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
```

В первый вызов `clGetDeviceIDs()` мы передаем `NULL` в аргументе `devices`, и в `num_devices` получаем число доступных устройств. После этого мы динамически выделяем нужный объем памяти для массива и снова вызываем `getDeviceIDs()`.

Обработка ошибок

Функции OpenCL сообщают об ошибках, возвращая их коды. Код `CL_SUCCESS` свидетельствует о том, что функция выполнялась успешно; любое другое значение говорит о возникшей ошибке. Так, обработка ошибок, которые могут произойти при вызове `clGetDeviceIDs()`, могла бы выглядеть примерно так:

```
cl_int status;

cl_uint num_devices;
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL,
&num_devices);
if (status != CL_SUCCESS) {
    fprintf(stderr, "Error: unable to determine num_devices (%d)\n",
status);
    exit(1);
}

cl_device_id* devices = (cl_device_id*)malloc(sizeof(cl_device_id)
* num_devices);
status = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices,
devices, NULL);
if (status != CL_SUCCESS) {
    fprintf(stderr, "Error: unable to retrieve devices (%d)\n",
status);
    exit(1);
}
```

Неудивительно, что в большинстве программ, использующих OpenCL, применяются вспомогательные функции или макросы, помогающие избавиться от этого шаблонного кода, например:

DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c¹⁴

```
#define CHECK_STATUS(s) do { \
    cl_int ss = (s); \
    if (ss != CL_SUCCESS) { \
        fprintf(stderr, "Error %d at line %d\n", ss, __LINE__); \
        exit(1); \
    } \
} while (0)
```

Этот макрос позволяет писать такой код:

DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c¹⁵

```
CHECK_STATUS(clSetKernelArg(kernel, 0, sizeof(cl_mem), &inputA));
```

Некоторые функции OpenCL не возвращают код ошибки непосредственно, но принимают параметр `error_ret`. Например, функция `clCreateContext()` имеет следующий прототип:

¹⁴ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c

¹⁵ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c

```
cl_context clCreateContext(  
    const cl_context_properties* properties,  
    cl_uint num_devices,  
    const cl_device_id* devices,  
    void (CL_CALLBACK* pfn_notify)(const char* errinfo,  
        const void* private_info,  
        size_t cb,  
        void* user_data),  
    void* user_data,  
    cl_int* errcode_ret);
```

Ниже показано, как реализовать обработку ошибок, которые могут возникнуть при обращении к этой функции:

DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c¹⁶

```
cl_int status;  
cl_context context = clCreateContext(NULL, 1, &device, NULL, NULL, &status);  
CHECK_STATUS(status);
```

Часто в программах на основе OpenCL можно встретить и другие приемы обработки ошибок. Выбор того или иного стиля зависит только от ваших личных предпочтений.

В завершение первого дня

На этом заканчивается первый день. Завтра мы поближе познакомимся с платформами OpenCL, особенностями выполнения и моделями памяти.

Что мы узнали в первый день

OpenCL позволяет использовать мощные возможности параллельной обработки данных, которыми обладают GPU. Понимание следующих особенностей поможет получить значительный прирост производительности:

- технология OpenCL основана на разделении большой задачи на множество мелких рабочих элементов, выполняемых параллельно;
- мы определяем число рабочих элементов, которое должно быть выполнено ядром;
- чтобы запустить ядро, основная программа должна сделать следующее:

¹⁶ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MultiplyArraysWithErrorHandling/multiply_arrays.c

- 1) создать контекст, в котором будет выполняться ядро, и очередь команд;
- 2) скомпилировать ядро;
- 3) создать буферы для входных и выходных данных;
- 4) добавить в очередь команду, которая запустит ядро по одному разу для каждого рабочего элемента;
- 5) извлечь результаты.

Упражнения к первому дню

Найдите:

- спецификацию OpenCL;
- справочник по OpenCL API;
- ядро OpenCL описывается на C-подобном языке. Какие отличия от C он имеет?

Задания:

- измените ядро умножения массивов для работы с массивами других типов и измерьте полученную производительность. Как она зависит от типов обрабатываемых данных? Влияет ли на производительность размер (в байтах) типа данных?
- мы создавали и инициализировали буферы, передавая флаг `CL_MEM_COPY_HOST_PTR` в вызов функции `clCreateBuffer()`. Перепишите основную программу так, чтобы она использовала флаг `CL_MEM_USE_HOST_PTR` или `CL_MEM_ALLOC_HOST_PTR` (при этом придется изменить не только имя флага в вызове функции), и измерьте производительность. Как она меняется в зависимости от стратегии выделения буферов?
- перепишите программу так, чтобы вместо `clCreateBuffer()` она использовала `clEnqueueMapBuffer()`, и измерьте производительность. В каких ситуациях такой подход является оптимальным, а в каких – нет?
- в языке OpenCL поддерживаются типы данных, отсутствующие в стандарте C, в частности имеются векторные типы данных, такие как `float4` и `ulong3`. Перепишите ядро так, чтобы оно выполняло умножение двух буферов с векторами. Как векторные типы можно представить в основной программе?

День 2: многомерность и рабочие группы

Вчера мы видели, как с помощью `clEnqueueNDRangeKernel()` реализовать выполнение множества рабочих элементов для обработки одномерного массива. Сегодня мы увидим аналогичный подход к обработке многомерных массивов и воспользуемся преимуществами группировки рабочих элементов для увеличения объемов обрабатываемых данных.

Многомерные массивы рабочих элементов

Когда основная программа вызывает `clEnqueueNDRangeKernel()` для выполнения ядра, она определяет пространство индексов. Каждая точка в этом индексном пространстве идентифицируется уникальным глобальным идентификатором, представляющим один рабочий элемент.

Ядро может получить глобальный идентификатор рабочего элемента с вызовом `get_global_id()`. Вчера в нашем примере использовалось одномерное пространство индексов, поэтому ядру требовалось вызвать `get_global_id()` только один раз. Сегодня мы создадим ядро, перемножающее две двухмерные матрицы и, как следствие, вызывающее `get_global_id()` дважды.

Умножение матриц

Сначала я предлагаю совершить краткий экскурс в линейную алгебру и вспомнить, как выполняется умножение матриц.

Матрица – это двухмерный массив чисел. Мы можем умножить матрицу $w \times n$ на матрицу $m \times w$ (обратите внимание, что ширина первой матрицы должна совпадать с высотой второй матрицы) и получить матрицу $m \times n$. Например, результатом умножения матрицы 2×4 на матрицу 3×2 является матрица 3×4 .

Чтобы получить значение (i, j) в матрице результата, нужно найти сумму произведений чисел j -й строки первой матрицы на соответствующие числа i -го столбца второй матрицы.

$$\begin{pmatrix} a & b \\ \tilde{n} & d \end{pmatrix} \begin{pmatrix} w & x \\ y & z \end{pmatrix} = \begin{pmatrix} aw+by & ax+bz \\ cw+dy & cx+dz \end{pmatrix}$$

Следующий код реализует эту последовательность операций:

```
#define WIDTH_OUTPUT WIDTH_B
#define HEIGHT_OUTPUT HEIGHT_A

float a[HEIGHT_A][WIDTH_A] = "инициализировать a";
float b[HEIGHT_B][WIDTH_B] = "инициализировать b";
float r[HEIGHT_OUTPUT][WIDTH_OUTPUT];

for (int j = 0; j < HEIGHT_OUTPUT; ++j) {
    for (int i = 0; i < WIDTH_OUTPUT; ++i) {
        float sum = 0.0;
        for (int k = 0; k < WIDTH_A; ++k) {
            sum += a[j][k] * b[k][i];
        }
        r[j][i] = sum;
    }
}
```

Как видите, увеличение размеров массивов приводит к резкому увеличению объемов вычислений, что превращает умножение больших матриц в задачу, потребляющую огромный объем вычислительных ресурсов CPU.

Параллельное умножение матриц

Ниже приводится ядро, с помощью которого можно выполнять умножение двумерных матриц:

DataParallelism/MatrixMultiplication/matrix_multiplication.cl¹⁷

```
1 __kernel void matrix_multiplication(uint widthA,
-                                     __global const float* inputA,
-                                     __global const float* inputB,
-                                     __global float* output) {
5
-     int i = get_global_id(0);
-     int j = get_global_id(1);
-
-     int outputWidth = get_global_size(0);
10    int outputHeight = get_global_size(1);
-     int widthB = outputWidth;
-
-     float total = 0.0;
-     for (int k = 0; k < widthA; ++k) {
15        total += inputA[j * widthA + k] * inputB[k * widthB + i];
```

¹⁷ http://media.pragprog.com/titles/pb7con/code/DataParallelism/MatrixMultiplication/matrix_multiplication.cl

```

-     }
-     output[j * outputWidth + i] = total;
- }

```

Это ядро выполняется в двухмерном пространстве индексов, каждая точка в котором идентифицирует местоположение в выходном массиве. Оно получает координаты этой точки, дважды вызывая `get_global_id()` (строки 6 и 7).

Размеры пространства индексов можно узнать с помощью функции `get_global_size()`, которую данное ядро использует для определения размеров выходной матрицы (строки 9 и 10). Тут же определяется ширина второй матрицы `widthB`, которая совпадает с шириной выходной матрицы `outputWidth`, ширину первой матрицы `widthA` мы вынуждены передавать в виде аргумента.

Цикл в строке 14 – это всего лишь внутренний цикл из последовательной версии, представленной ранее, – единственное отличие вызвано тем, что буферы в OpenCL могут быть только одномерными, то есть мы не можем написать такой код:

```
output[j][i] = total;
```

Вместо этого приходится использовать арифметику, чтобы определить правильное смещение:

```
output[j * outputWidth + i] = total;
```

Основная программа, использующая это ядро, очень похожа на программу, что мы видели вчера. Единственное существенное отличие заключено в аргументах функции `clEnqueueNDRangeKernel()`:

DataParallelism/MatrixMultiplication/matrix_multiplication.c¹⁸

```

size_t work_units[] = {WIDTH_OUTPUT, HEIGHT_OUTPUT};
CHECK_STATUS(clEnqueueNDRangeKernel(queue, kernel, 2, NULL,
    work_units, NULL, 0, NULL, NULL));

```

Передачей значения 2 в аргументе `work_dim` (третий аргумент функции) создается двухмерное пространство индексов, а передачей в аргументе `global_work_size` (пятый аргумент) массива с двумя элементами определяется размерность каждого из измерений.

Это ядро демонстрирует еще более существенный прирост производительности, чем мы видели вчера. На моем MacBook Pro умноже-

¹⁸ media.pragprog.com/titles/pb7con/code/DataParallelism/MatrixMultiplication/matrix_multiplication.c

ние матрицы 200×400 на матрицу 300×200 выполняется примерно за 3 мсек. Для сравнения: умножение этих же матриц на CPU выполняется за 66 мсек. То есть умножение на GPU было выполнено быстрее более чем в 20 раз.

Так как это ядро выполняет намного больше работы, чтобы получить каждый элемент выходной матрицы, мы будем наблюдать увеличение производительности, даже если примем во внимание накладные расходы на копирование данных между CPU и GPU. На моем MacBook Pro эти операции копирования выполняются примерно за 2 мсек. То есть общее время равно 5 мсек, что составляет 13-кратный прирост производительности.

Во всех примерах, демонстрировавшихся до сих пор, просто предполагалось наличие OpenCL-совместимого GPU. Очевидно, что такое предположение не всегда верно, поэтому далее мы посмотрим, как узнать, какие платформы и устройства OpenCL доступны на данном компьютере.

Получение информации об устройстве

Стандарт OpenCL включает несколько функций, позволяющих определять параметры платформ, устройств и большинства других объектов API. Ниже, к примеру, демонстрируется функция, использующая `clGetDeviceInfo()` для определения и вывода параметра устройства, имя которого задается в виде строки:

```
DataParallelism/FindDevices/find_devices.c19
```

```
void print_device_param_string(cl_device_id device,
                              cl_device_info param_id,
                              const char* param_name) {
    char value[1024] ;
    CHECK_STATUS(clGetDeviceInfo(device, param_id, sizeof(value), value,
    NULL));
    printf("%s: %s\n", param_name, value);
}
```

Разные параметры имеют значения разных типов (строки, целые числа, массивы элементов типа `size_t` и т. д.). Определив разные функции, подобные предыдущей, мы сможем запрашивать разные параметры устройств, как показано ниже:

¹⁹ http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindDevices/find_devices.c

DataParallelism/FindDevices/find_devices.c²⁰

```
void print_device_info(cl_device_id device) {
    print_device_param_string(device, CL_DEVICE_NAME, "Name");
    print_device_param_string(device, CL_DEVICE_VENDOR, "Vendor");
    print_device_param_uint(device, CL_DEVICE_MAX_COMPUTE_UNITS,
"Compute Units");
    print_device_param_ulong(device, CL_DEVICE_GLOBAL_MEM_SIZE,
"Global Memory");
    print_device_param_ulong(device, CL_DEVICE_LOCAL_MEM_SIZE,
"Local Memory");
    print_device_param_sizet(device, CL_DEVICE_MAX_WORK_GROUP_SIZE,
"Workgroup size");
}
```

В загружаемом пакете примеров для этой книги вы найдете исходный код программы с именем `find_devices`, в которой подобный подход используется для определения доступных платформ и устройств. На моем MacBook Pro эта программа выводит следующее:

```
Found 1 OpenCL platform(s)

Platform 0
Name: Apple
Vendor: Apple

Found 2 device(s)

Device 0
Name: Intel(R) Core(TM) i7-3720QM CPU @ 2.60GHz
Vendor: Intel
Compute Units: 8
Global Memory: 17179869184
Local Memory: 32768
Workgroup size: 1024

Device 1
Name: GeForce GT 650M
Vendor: NVIDIA
Compute Units: 2
Global Memory: 1073741824
Local Memory: 49152
Workgroup size: 1024
```

То есть в данном случае доступна одна платформа, с реализацией по умолчанию Apple OpenCL. Внутри платформы имеются два устройства, CPU и GPU.

²⁰ http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindDevices/find_devices.c

Из полученных результатов можно сделать несколько интересных выводов:

- ядра OpenCL способны работать не только на GPU (помимо CPU, они также могут работать на выделенных акселераторах OpenCL);
- GPU на моем MacBook Pro имеет два вычислительных модуля (что такое «вычислительный модуль», вы узнаете чуть ниже);
- GPU имеет 1 Гибайт²¹ глобальной памяти;
- каждый вычислительный модуль имеет 48 Кибайт локальной памяти и поддерживает рабочие группы с максимальным размером 1024.

В следующих разделах мы поближе познакомимся с платформой и моделью памяти OpenCL, а также узнаем, какое значение они имеют для нашего программного кода.



Вопрос Джо: зачем OpenCL поддерживать CPU?

Для многих это будет неожиданностью, но современные CPU уже довольно давно поддерживают параллельные инструкции. Процессоры компании Intel, например, поддерживают потоковые расширения SIMD системы команд (Streaming SIMD Extensions, SSE) и более новые векторные расширения системы команд (Advanced Vector Extensions, AVX). OpenCL с успехом может использовать эти команды, а также множество ядер, которыми обладает подавляющее большинство современных процессоров.

Модель платформы

Платформа OpenCL включает хост, соединенный с одним или более устройствами. Каждое устройство имеет один или более *вычислительных модулей* (compute units), каждый из которых, в свою очередь, имеет один или более *обрабатывающих элементов* (processing elements), как показано на рис. 29.

Рабочие элементы выполняются обрабатывающими элементами. Коллекция рабочих элементов, выполняющихся на одном вычислительном модуле, называется *рабочей группой* (work-group). Рабочие элементы, принадлежащие одной рабочей группе, совместно исполь-

²¹ 1 гигабайт = 1 073 741 824 байта. – Прим. перев.

зуют общую локальную память, что влечет необходимость обсудить модель памяти в OpenCL.

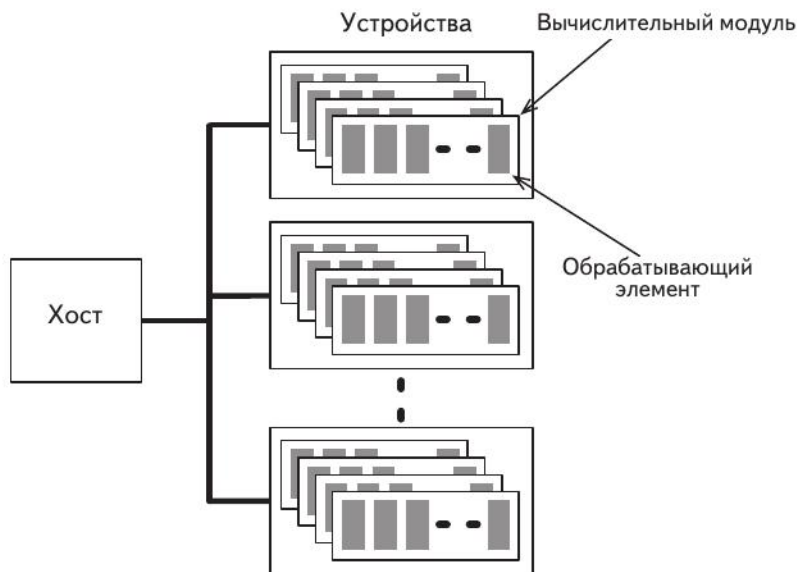


Рис. 29. Модель платформы в OpenCL

Модель памяти

Рабочий элемент, выполняемый ядром, имеет доступ к четырем разным областям памяти:

Глобальная память: память, доступная всем рабочим элементам, выполняющимся на устройстве.

Постоянная память: область глобальной памяти, содержимое которой не изменяется в процессе выполнения ядра.

Локальная память: память, локальная для рабочей группы; может использоваться для взаимодействий между рабочими элементами в этой рабочей группе (пример таких взаимодействий будет показан далее).

Частная (private) память: память, доступная только одному рабочему элементу.

Как мы видели в предыдущих главах, операция свертки коллекций является довольно эффективным решением широкого круга задач. В следующем разделе мы посмотрим, как реализовать параллельную свертку.

Параллельная свертка

В этом разделе мы создадим ядро, которое находит минимальное значение в коллекции, применяя к этой коллекции операцию свертки с оператором `min()`.



Вопрос Джо: как в действительности работают устройства OpenCL?

Модели платформы и памяти в OpenCL не определяют порядок работы аппаратного обеспечения. Фактически они являются абстракциями этого аппаратного обеспечения – разные устройства OpenCL могут иметь разную физическую организацию.

Например, одно устройство OpenCL может действительно иметь локальную память в вычислительных модулях, другое может отображать локальную память на область в глобальной памяти. Или одно устройство может иметь собственную глобальную память, а другое – использовать память хоста.

Все эти архитектурные различия могут иметь существенное значение для оптимизации кода на языке OpenCL, но мы не будем затрагивать эту тему, так как она далеко выходит за рамки данной главы.

Последовательная реализация этой задачи выглядит просто:

```
DataParallelism/FindMinimumOneWorkGroup/find_minimum.c22
```

```
cl_float acc = FLT_MAX;
for (int i = 0; i < NUM_VALUES; ++i)
    acc = fmin(acc, values[i]);
```

Параллельная версия выглядит немного сложнее. Далее мы реализуем параллельную обработку в два этапа – сначала напишем версию с одной рабочей группой, а затем с большим их количеством.

Свертка с одной рабочей группой

Чтобы упростить задачу (чем это поможет, вы поймете чуть ниже), я буду предполагать, что число элементов в массиве, который нужно свернуть, кратно степени двойки и не слишком велико, чтобы можно было ограничиться единственной рабочей группой. Следующее ядро реализует операцию свертки с учетом всех этих допущений:

²² http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindMinimumOneWorkGroup/find_minimum.c

DataParallelism/FindMinimumOneWorkGroup/find_minimum.cl²³

```

1  __kernel void find_minimum(__global const float* values,
-                               __global float* result,
-                               __local float* scratch) {
-      int i = get_global_id(0);
5  int n = get_global_size(0);
-      scratch[i] = values[i];
-      barrier(CLK_LOCAL_MEM_FENCE);
-      for (int j = n / 2; j > 0; j /= 2) {
-          if (i < j)
10         scratch[i] = min(scratch[i], scratch[i + j]);
-          barrier(CLK_LOCAL_MEM_FENCE);
-      }
-      if (i == 0)
-          *result = scratch[0];
15 }

```

Алгоритм предусматривает обработку в три этапа:

1. Копирование массива из глобальной в локальную (*scratch*) память (строка 6).
2. Выполнение свертки (строки 8–12).
3. Копирование результата в глобальную память (строка 14).

В процессе выполнения операции свертки создается дерево свертки (*reduce tree*, см. рис. 30), очень похожее на то, что мы видели, когда знакомились с редуцентами (*reducers*) в языке Clojure (см. раздел «Разделяй и властвуй» в главе 3).

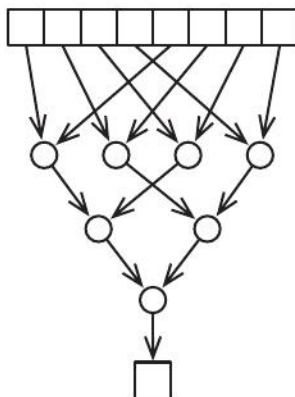


Рис. 30. Дерево свертки

²³ http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindMinimumOneWorkGroup/find_minimum.cl

После каждой итерации половина рабочих элементов становятся неактивными – какую-либо работу продолжают выполнять только элементы, для которых выполняется условие $i < j$ (именно поэтому я ввел ограничение на число рабочих элементов, которое должно быть кратно степени двойки, – так мы можем повторять деление массива пополам). Цикл завершается, только когда остается активным лишь один рабочий элемент. Каждый рабочий элемент выполняет `min()` для своего значения и соответствующего значения в оставшейся половине массива. К моменту завершения цикла первый элемент в массиве `scratch` будет хранить окончательный результат операции свертки, а первый рабочий элемент в рабочей группе скопирует это значение в `result`.

Другой интересный момент, касающийся данного ядра, заключается в использовании барьеров (строки 7 и 11) для синхронизации доступа к локальной памяти.

Барьеры

Барьер (`barrier`) – это механизм синхронизации, позволяющий рабочим элементам координировать обращения к локальной памяти. Если один рабочий элемент в рабочей группе вызывает `barrier()`, все остальные рабочие элементы в этой же рабочей группе также должны вызвать `barrier()`, прежде чем любой из них сможет продолжить работу (эта разновидность синхронизации известна под названием *точка встречи* (`rendezvous`)). В нашем ядре данный прием синхронизации преследует две цели:

- гарантировать, что ни один рабочий элемент не приступит к свертке, пока все рабочие элементы не скопируют свои значения из глобальной памяти в локальную, и ни один рабочий элемент не перейдет к итерации с порядковым номером $n + 1$, пока все рабочие элементы не завершат итерацию с порядковым номером n ;
- спецификация OpenCL определяет требования к целостности данных в памяти, напоминающие модель памяти в Java, с которой мы познакомились в разделе «Видимость памяти» в главе 2, – изменения, выполненные одним рабочим элементом в локальной памяти, не обязательно должны быть видимы другим рабочим элементам, за исключением специальных точек синхронизации, таких как барьеры. Соответственно, использование барьера в конце каждой итерации цикла гарантирует,

что результаты итерации n будут доступны рабочим элементам, перед началом итерации $n + 1$.

Запуск ядра

Запуск этого ядра напоминает то, что мы уже видели выше. Единственное новшество, на котором следует остановиться, – создание локального буфера:

```
DataParallelism/FindMinimumOneWorkGroup/find_minimum.c24
```

```
CHECK_STATUS(clSetKernelArg(kernel, 2, sizeof(cl_float) * NUM_VALUES, NULL));
```

Выделение локального буфера осуществляется вызовом функции `clSetKernelArg()`, которой в аргументе `arg_size` передается размер буфера, и в аргументе `arg_value` – значение `NULL`.

Мы справились с реализацией свертки в виде одной рабочей группы, но, как мы знаем, рабочие группы имеют ограниченные размеры (например, на моем MacBook Pro рабочая группа может включать не более 1024 рабочих элементов). Далее мы посмотрим, как реализовать параллельное решение с применением нескольких рабочих групп.

Свертка с несколькими рабочими группами

Чтобы задействовать несколько рабочих групп, достаточно просто поделить исходный массив между имеющимися рабочими группами и выполнить свертку в каждой из них независимо, как показано на рис. 31.

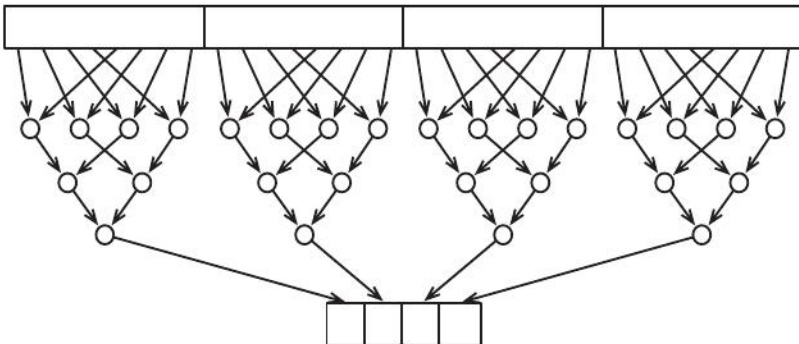


Рис. 31. Свертка массива с использованием нескольких рабочих групп

²⁴ http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindMinimumOneWorkGroup/find_minimum.c

Если, к примеру, каждая рабочая группа способна одновременно оперировать 64 значениями, тогда исходный массив из N элементов окажется свернут до размера $N/64$ элементов. Затем для этого массива меньшего размера можно снова выполнить операцию свертки и т. д., пока не останется единственного результата.

Чтобы наше ядро могло обрабатывать большую задачу частями, в него требуется внести несколько небольших изменений. С этой целью OpenCL поддерживает получение в рабочих элементах локальных идентификаторов, которые являются идентификаторами внутри групп, как показано на рис. 32.



Рис. 32. Локальные идентификаторы рабочих групп

Ниже приводится ядро, использующее локальные идентификаторы:

DataParallelism/FindMinimumMultipleWorkGroups/find_minimum.cl²⁵

```
_kernel void find_minimum(_global const float* values,
                          _global float* results,
                          _local float* scratch) {
> int i = get_local_id(0);
> int n = get_local_size(0);
> scratch[i] = values[get_global_id(0)];
  barrier(CLK_LOCAL_MEM_FENCE);
  for (int j = n / 2; j > 0; j /= 2) {
    if (i < j)
      scratch[i] = min(scratch[i], scratch[i + j]);
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (i == 0)
> results[get_group_id(0)] = scratch[0];
}
```

Вместо функций `get_global_id()` и `get_global_size()` теперь вызываются функции `get_local_id()` и `get_local_size()`, возвращаю-

²⁵ http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindMinimumMultipleWorkGroups/find_minimum.cl

щие идентификатор вызывающего рабочего элемента внутри группы и размер рабочей группы соответственно. Нам все еще требуется вызвать `get_global_id()`, чтобы скопировать правильное значение из глобальной памяти в локальную, и `results` теперь является массивом, индексируемым с помощью `get_group_id()`.

Заключительным элементом мозаики являются изменения в основной программе, реализующие создание рабочих групп соответствующего размера:

```
DataParallelism/FindMinimumMultipleWorkGroups/find_minimum.c26
```

```
size_t work_units[] = {NUM_VALUES};
size_t workgroup_size[] = {WORKGROUP_SIZE};
CHECK_STATUS(clEnqueueNDRangeKernel(queue, kernel, 1, NULL, work_units,
    workgroup_size, 0, NULL, NULL));
```

Если в аргументе `local_work_size` передать значение `NULL`, как мы делали это до сих пор, платформа OpenCL автоматически создаст рабочие группы подходящего размера. Явно устанавливая значение `local_work_size`, мы добиваемся того, что рабочие группы будут иметь размер, требуемый нашим ядром (разумеется, эта величина не должна превышать максимально возможного размера группы – см. раздел «Получение информации об устройстве» выше, где описывается, как узнать это значение).

В завершение второго дня

Мы подошли к концу второго дня. Завтра мы рассмотрим пример приложения, имитирующего физическое явление с помощью OpenCL и использующего OpenGL для отображения результатов.

Что мы узнали во второй день

Стандарт OpenCL определяет модели платформы, выполнения и памяти, представляющие абстракцию аппаратного обеспечения:

- рабочие элементы выполняются на обрабатывающих элементах;
- обрабатывающие элементы группируются в вычислительные модули;
- группа рабочих элементов, которая выполняется на одном вычислительном модуле, называется рабочей группой;

²⁶ http://media.pragprog.com/titles/pb7con/code/DataParallelism/FindMinimumMultipleWorkGroups/find_minimum.c

- рабочие элементы внутри рабочей группы могут взаимодействовать друг с другом посредством локальной памяти, используя барьеры для синхронизации.

Упражнения ко второму дню

Найдите:

- по умолчанию очередь команд обрабатывает команды в порядке их поступления. Как бы вы реализовали внеочередное выполнение команд?
- что такое «очередь событий»? Как можно было бы использовать такие очереди, чтобы определить, когда должны выполняться команды из неупорядоченной очереди команд?
- что делает функция `clEnqueueBarrier()`? В каких случаях можно было бы использовать барьеры, а в каких очереди?

Задания:

- измените пример реализации свертки так, чтобы программа могла обрабатывать массивы с любым числом элементов, а не только кратным степени двойки;
- измените пример реализации свертки так, чтобы он мог использовать несколько устройств. Если в наличии имеется только одно OpenCL-совместимое устройство, можете также использовать CPU или фрагментировать GPU с помощью `clCreateSubDevices()`. Вам придется создать очередь команд для каждого устройства, разбить задачу так, чтобы часть рабочих элементов выполнялась на одном устройстве, а часть на другом, и синхронизировать очереди команд;
- алгоритм свертки, реализованный в сегодняшнем примере, очень прост. Поислав в Интернете, можно найти множество более эффективных алгоритмов. Как быстро вы сможете реализовать их для своего конкретного устройства? Проверьте, являются ли оптимизации для вашего GPU такими же эффективными для CPU?

День 3: OpenCL и OpenGL – храните данные в GPU

Сегодня мы соединим вместе полноценную программу на основе OpenCL, которая имитирует и визуализирует простое физическое

явление. В ходе обсуждения вы не только увидите ядро, выполняющее параллельные вычисления, но также узнаете, как объединить OpenCL с OpenGL и избежать накладных расходов на копирование данных в GPU.

Водная рябь

Сегодня мы реализуем имитацию ряби на воде. Конечно, это будет не самая точная имитация, но достаточно правдоподобная, чтобы ее с успехом можно было использовать в игровых программах, например для изображения поверхности воды в водоеме во время дождя.

LWJGL

Сегодня мы уйдем от языка C и реализуем пример на языке Java с использованием библиотеки Lightweight Java Graphics Library (LWJGL),²⁷ чтобы упростить кросс-платформенную реализацию графического пользовательского интерфейса.

В библиотеке LWJGL реализованы Java-обертки для поддержки OpenCL и OpenGL, что позволяет писать на Java программы, использующие прикладные интерфейсы OpenGL и OpenCL. Как можно догадаться по названию OpenGL, этот стандарт тесно связан с OpenCL. В частности, как будет показано далее, ядро OpenCL, выполняющееся на GPU, может непосредственно оперировать буферами OpenGL.

Программный код, который с помощью библиотеки LWJGL обращается к OpenCL API, очень напоминает аналогичный программный код на C, который мы видели прежде. Например, ниже приводится код инициализации контекста, очереди и ядра OpenCL:

```
DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java28
```

```
CL.create();
CLPlatform platform = CLPlatform.getPlatforms().get(0);
List<CLDevice> devices = platform.getDevices(CL_DEVICE_TYPE_GPU);
CLContext context = CLContext.create(platform, devices, null, drawable,
null);
CLCommandQueue queue = clCreateCommandQueue(context, devices.get(0),
0, null);

CLProgram program =
    clCreateProgramWithSource(context, loadSource("zoom.cl"), null);
```

²⁷ <http://www.lwjgl.org>

²⁸ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java>


```
Util.checkCLError(clBuildProgram(program, devices.get(0), "", null));
CLKernel kernel = clCreateKernel(program, "zoom", null);
```

Как видите, имена методов и аргументов практически идентичны именам в языке C. Да, есть некоторые мелкие отличия, обусловленные различиями между языками, такими как отсутствие указателей в Java, но в общем и целом благодаря библиотеке LWJGL программу на C, использующую функции OpenGL, можно перенести на язык Java почти без изменений.

Отображение сетки в OpenGL

Мы не будем тратить слишком много времени на обсуждение данного примера в части, касающейся OpenGL. Но нам определенно следует разобраться с тем, как пример отображает на экране сетку, представляющую поверхность воды, чтобы понять, какая задача стоит перед нашим кодом OpenCL.

В OpenGL трехмерные сцены конструируются из треугольников. В нашем случае мы будем создавать сетку из треугольников, как показано на рис. 33.

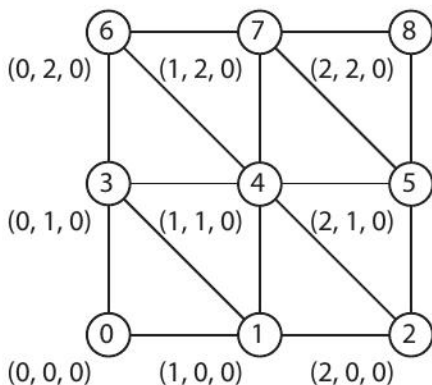


Рис. 33. Сетка из треугольников, имитирующая поверхность воды

Позиция каждого треугольника определяется в два этапа – в *вершинном буфере* (vertex buffer), где определяется множество вершин (точек в трехмерном пространстве), и в *индексном буфере* (index buffer), где определяется, какие вершины используются для рисования каждого треугольника.

Так, в предыдущем примере вершина 0 имеет координаты (0, 0, 0), вершина 1 – координаты (1, 0, 0), вершина 2 – координаты (2, 0, 0)

и т. д. Поэтому вершинный буфер содержит массив значений [0, 0, 0, 1, 0, 0, 2, 0, 0, 0, 1, 0, 1, 1, 0, ...].

Как определено в индексном буфере, первый треугольник задается вершинами 0, 1 и 3; второй треугольник – вершинами 1, 3 и 4; третий – вершинами 1, 2 и 4 и т. д. Созданный нами индексный буфер определяет *полосу из треугольников* (triangle strip), в которой только первый треугольник задается тремя вершинами, а каждый последующий – одной дополнительной вершиной, как показано на рис. 34.

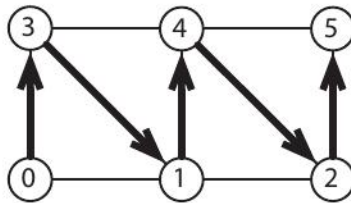


Рис. 34. Полоса из треугольников, где только первый треугольник задается тремя вершинами

Поэтому индексный буфер содержит массив [0, 3, 1, 4, 2, 5, ...].

Исходный код, входящий в состав загружаемого пакета примеров для книги, содержит определение класса `Mesh`, который генерирует начальные значения для вершинного и индексного буферов. Наш пример использует его для создания сетки размером 64×64 с координатами X и Y , изменяющимися в диапазоне от -1.0 до 1.0 :

```
DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java29
```

```
Mesh mesh = new Mesh(2.0f, 2.0f, 64, 64);
```

Координаты Z всех вершин инициализируются нулевым значением – они будут изменяться в процессе анимации, воспроизводящей эффект ряби на воде.

Затем исходные данные копируются в буферы OpenGL, как показано ниже:

```
DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java30
```

```
int vertexBuffer = glGenBuffers();
glBindBuffer(GL_ARRAY_BUFFER, vertexBuffer);
```

²⁹ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java>

³⁰ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java>

```
glBufferData(GL_ARRAY_BUFFER, mesh.vertices, GL_DYNAMIC_DRAW);

int indexBuffer = glGenBuffers();
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, mesh.indices, GL_STATIC_DRAW);
```

Каждый буфер получает собственный числовой идентификатор при размещении вызовом `glGenBuffers()`, затем связывается с типом буфера вызовом `glBindBuffer()` и инициализируется начальными значениями вызовом `glBufferData()`. При инициализации индексного буфера передается флаг `GL_STATIC_DRAW`, указывающий, что он не будет изменяться (то есть является статичным). При инициализации вершинного буфера, напротив, передается флаг `GL_DYNAMIC_DRAW`, потому что содержимое этого буфера будет изменяться в ходе анимации.

Перед тем как переходить к реализации эффекта ряби, решим более простую задачу – напомним ядро, увеличивающее размер ячеек сетки с течением времени.

Доступ к буферу OpenGL из ядра OpenCL

Ниже приводится ядро, реализующее эффект увеличения размеров ячеек:

DataParallelism/Zoom/src/main/resources/zoom.cl³¹

```
_kernel void zoom(_global float* vertices) {

    unsigned int id = get_global_id(0);
    vertices[id] *= 1.01;
}
```

Оно принимает вершинный буфер и умножает каждый элемент этого буфера на 1.01, увеличивая тем самым размер каждой ячейки на 1% при каждом вызове.

Но, прежде чем передать вершинный буфер в наше ядро, нам сначала нужно создать буфер OpenCL, ссылающийся на вершинный буфер:

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java³²

```
CLMem vertexBufferCL =
    clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, vertexBuffer, null);
```

³¹ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Zoom/src/main/resources/zoom.cl>

³² <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java>

Этот объект буфера теперь можно использовать в нашем главном цикле отображения:

DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java³³

```
while (!Display.isCloseRequested()) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, planeDistance);
    glDrawElements(GL_TRIANGLE_STRIP, mesh.indexCount, GL_UNSIGNED_
SHORT, 0);

    Display.update();

> Util.checkCLError(clEnqueueAcquireGLObjects(queue, vertexBufferCL,
null, null));
> kernel.setArg(0, vertexBufferCL);
> clEnqueueNDRangeKernel(queue, kernel, 1, null, workSize, null, null,
null);
> Util.checkCLError(clEnqueueReleaseGLObjects(queue, vertexBufferCL,
null, null));
> clFinish(queue);
}
```

Прежде чем ядро OpenCL сможет пользоваться буфером OpenGL, нужно захватить его вызовом `clEnqueueAcquireGLObjects()`. После этого буфер можно передать ядру в виде аргумента и вызвать `clEnqueueNDRangeKernel()` как обычно. По окончании следует освободить буфер с помощью `clEnqueueReleaseGLObjects()` и ждать команды на завершение, которую мы запросили вызовом `clFinish()`.

Запустив этот код, вы должны увидеть сетку с мелкими ячейками, которые постепенно увеличиваются, пока единственный треугольник не заполнит весь экран.

Теперь, реализовав действующую анимацию, которая воспроизводится средствами OpenGL и OpenCL, перейдем к более сложному ядру, реализующему эффект ряби на воде.

Имитация ряби

Наша цель – реализовать имитацию ряби от дождя на поверхности воды в виде расходящихся кругов. Каждый круг определяется двухмерной точкой на сетке (центр расходящегося круга), а также временем (прошедшим с момента падения капли). Помимо указателя на

³³ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Zoom/src/main/java/com/paulbutcher/Zoom.java>

вершинный буфер OpenGL, наше ядро будет принимать массив точек падения капель и соответствующий ему массив времен (время измеряется в миллисекундах):

DataParallelism/Ripple/src/main/resources/ripple.cl³⁴

```

1 #define AMPLITUDE 0.1
- #define FREQUENCY 10.0
- #define SPEED 0.5
- #define WAVE_PACKET 50.0
5 #define DECAY_RATE 2.0
- __kernel void ripple(__global float* vertices,
-                     __global float* centers,
-                     __global long* times,
-                     int num_centers,
10                     long now) {
-     unsigned int id = get_global_id(0);
-     unsigned int offset = id * 3;
-     float x = vertices[offset];
-     float y = vertices[offset + 1];
15     float z = 0.0;
-
-     for (int i = 0; i < num_centers; ++i) {
-         if (times[i] != 0) {
-             float dx = x - centers[i * 2];
20             float dy = y - centers[i * 2 + 1];
-             float d = sqrt(dx * dx + dy * dy);
-             float elapsed = (now - times[i]) / 1000.0;
-             float r = elapsed * SPEED;
-             float delta = r - d;
25             z += AMPLITUDE *
-                 exp(-DECAY_RATE * r * r) *
-                 exp(-WAVE_PACKET * delta * delta) *
-                 cos(FREQUENCY * M_PI_F * delta);
-         }
30     }
-     vertices[offset + 2] = z;
- }

```

Сначала определяются координаты X и Y вершины, обрабатываемой текущим рабочим элементом (строки 13 и 14). В цикле (строки 17–30) вычисляется новое значение координаты Z, которое записывается обратно в вершинный буфер в строке 31.

Внутри цикла проверяется, отличается ли от нуля время падения каждой капли. Если отличается, определяем расстояние d от рассма-

³⁴ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Ripple/src/main/resources/ripple.cl>

триваемой точки до центра круга (строка 21). Далее вычисляем радиус r расширяющегося круга (строка 23) и δ – расстояние от нашей точки до гребня круга (строка 24), как показано на рис. 35.

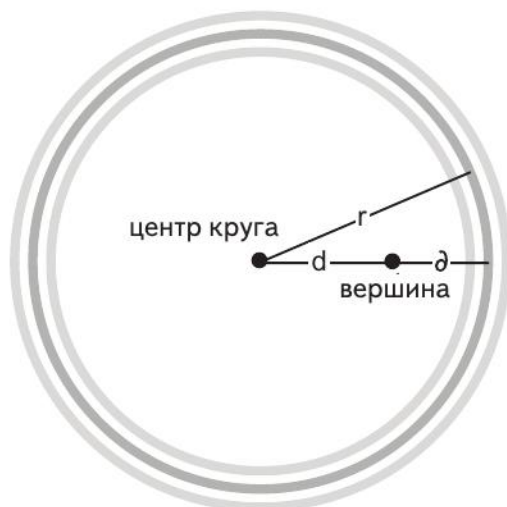


Рис. 35. Вычисляемые параметры расходящегося круга

Наконец, объединяем d и r , чтобы получить координату Z :

$$Z = Ae^{-Dr^2} e^{-W\delta^2} \cos(F\pi\delta).$$

Здесь A , D , W и F – это константы, определяющие амплитуду пакета волны, скорость падения амплитуды с увеличением радиуса, ширину пакета волны и частоту соответственно.

Последний элемент мозаики – добавление в основную программу поддержки создания точек падения капель:

DataParallelism/Ripple/src/main/java/com/paulbutcher/Ripple.java³⁵

```
int numCenters = 16;
int currentCenter = 0;
FloatBuffer centers = BufferUtils.createFloatBuffer(numCenters * 2);
centers.put(new float[numCenters * 2]);
centers.flip();
LongBuffer times = BufferUtils.createLongBuffer(numCenters);
times.put(new long[numCenters]);
```

³⁵ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Ripple/src/main/java/com/paulbutcher/Ripple.java>


```
times.flip();

CLMem centersBuffer =
    clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, centers,
null);
CLMem timesBuffer =
    clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR, times,
null);
```

По щелчку мышью:

DataParallelism/Ripple/src/main/java/com/paulbutcher/Ripple.java³⁶

```
while (Mouse.next()) {
    if (Mouse.getEventButtonState()) {
        float x = ((float)Mouse.getEventX() / Display.getWidth()) * 2 - 1;
        float y = ((float)Mouse.getEventY() / Display.getHeight()) * 2 - 1;

        FloatBuffer center = BufferUtils.createFloatBuffer(2);
        center.put(new float[] {x, y});
        center.flip();
        clEnqueueWriteBuffer(queue, centersBuffer, 0,
            currentCenter * 2 * FLOAT_SIZE, center, null, null);
        LongBuffer time = BufferUtils.createLongBuffer(1);
        time.put(System.currentTimeMillis());
        time.flip();

        clEnqueueWriteBuffer(queue, timesBuffer, 0,
            currentCenter * LONG_SIZE, time, null, null);
        currentCenter = (currentCenter + 1) % numCenters;
    }
}
```

Скомпилируйте и запустите этот код, щелкните на сетке несколько раз, и вы должны увидеть картину, похожую на ту, что изображена на рис. 36.

Итак, у нас получилось создать имитацию ряби на воде, расчеты и визуализация которой производятся параллельно, графическим процессором. Все данные, необходимые для вычислений и визуализации, постоянно хранятся в GPU, и нам не требуется ничего копировать.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения модели параллельной обработки данных на GPU с помощью OpenCL.

³⁶ <http://media.pragprog.com/titles/pb7con/code/DataParallelism/Ripple/src/main/java/com/paulbutcher/Ripple.java>

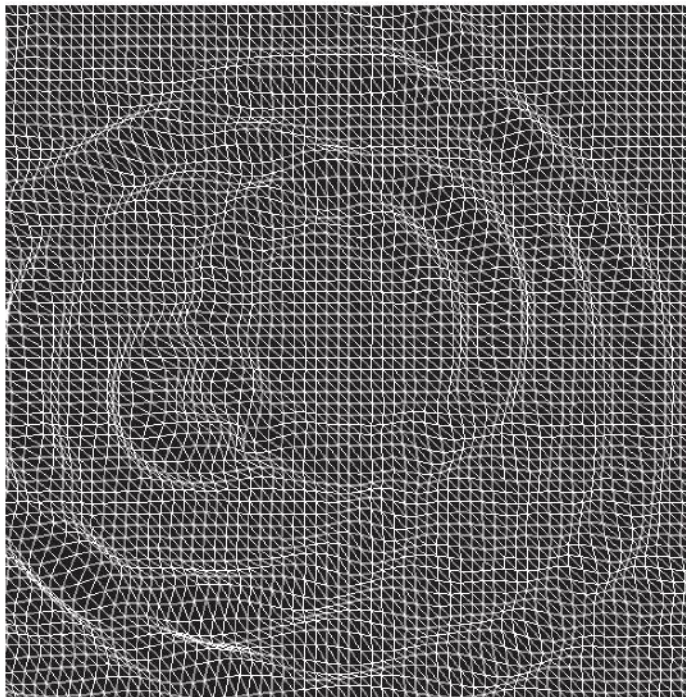


Рис. 36. Рябь

Что мы узнали в третий день

Ядро OpenCL, выполняющееся на GPU, имеет прямой доступ к буферам OpenGL на том же GPU. Мы узнали сегодня, как:

- создать OpenCL-представление буфера OpenGL с помощью `clCreateFromGLBuffer()`;
- захватить буфер OpenGL перед передачей в ядро с помощью `clEnqueueAcquireGLObjects()`;
- освободить буфер после вычислений в ядре с помощью `clEnqueueReleaseGLObjects()`.

Упражнения к третьему дню

Найдите:

- что такое объект изображения? Чем он отличается от обычного буфера OpenCL? Могут ли пригодиться объекты изображений в ядре, никак не взаимодействующем с OpenGL?

- что такое семплер? Какие задачи помогает решать этот объект?
- что такое атомарная функция? В каких случаях атомарные функции оказываются предпочтительнее барьеров?

Задания:

- создайте ядро (без использования атомарных функций), принимающее буфер с целыми числами от 0 до 32 и реализующее вывод гистограммы частоты встречаемости каждого значения. Сможете ли вы расширить свое решение так, чтобы оно было в состоянии обрабатывать тот же буфер, но содержащий числа от 0 до 1024?
- напишите ядро, решающее ту же задачу, что и в предыдущем задании, но с использованием атомарных функций. Как бы вы охарактеризовали отличия между этими двумя решениями?

В завершение

По непонятным причинам модель параллелизма данных игнорируется во многих обсуждениях темы параллелизма. Однако, как мы убедились, эта модель позволяет существенно повысить производительность и определенно должна входить в арсенал любого программиста.

Сильные стороны

Прием параллельной обработки данных идеально подходит в ситуациях, когда требуется обрабатывать огромные объемы числовых данных. Он, в частности, прекрасно подходит для реализации научных и инженерных вычислений и имитации процессов. В качестве примеров можно привести решение задач газогидродинамики, расчеты методом конечных элементов, построение n -мерных моделей, математический аннилинг, оптимизацию подражанием муравьиной колонии, нейросети и т. д.

Графические процессоры не только обладают мощными возможностями параллельной обработки данных; они также чрезвычайно эффективны с точки зрения потребления электроэнергии, имея намного более лучшее отношение Гфлопс/Ватт, чем традиционные процессоры. Это является одной из основных причин, почему самые быстрые в мире суперкомпьютеры широко используют GPU или специализированные процессоры для параллельной обработки данных.³⁷

³⁷ <http://www.top500.org/lists/2013/06/>

Слабые стороны

В своей нише параллельное программирование в целом и GPGPU в частности находятся вне конкуренции. Но этот подход пригоден не для всех задач. Например, хотя параллелизм данных можно применить для создания решений задач, не связанных с вычислениями, таких как обработка текстов на естественных языках, подобные решения получатся слишком сложными — имеющийся набор инструментов слишком сильно специализирован именно для числовых вычислений.

Оптимизация ядра OpenCL может оказаться очень непростым делом, а эффективность оптимизации нередко зависит от понимания архитектурных особенностей аппаратуры. Особые сложности могут возникнуть, если потребуется написать высокопроизводительный кросс-платформенный код. В некоторых ситуациях копирование данных из основной памяти в устройство может занимать основную часть времени вычислений, уменьшая тем самым выгоды, получаемые от параллельных вычислений.

Другие языки

В качестве примеров других фреймворков GPGPU можно привести: CUDA,³⁸ DirectCompute,³⁹ и RenderScript Computation.⁴⁰

Напоследок

Программирование GPGPU является ярким примером использования модели параллельной обработки данных на одном компьютере. В следующей главе мы рассмотрим лямбда-архитектуру, позволяющую распространить модель параллельной обработки на множество компьютеров.

³⁸ http://www.nvidia.com/object/cuda_home_new.html

³⁹ <http://msdn.com/directx>

⁴⁰ <http://developer.android.com/guide/topics/renderscript/compute.html>



ГЛАВА 8.

Лямбда-архитектура

Если вам потребуется доставить большой объем груза из одного конца страны в другой, едва ли найдется что-то лучше 18-колесного грузовика. Но для доставки одного пакета этот вариант совершенно не годится, поэтому компании, занимающиеся доставкой грузов, имеют в своем распоряжении машины поменьше, выполняющие перевозку небольших грузов.

Лямбда-архитектура (Lambda Architecture) объединяет в себе мощь технологии MapReduce и приемов потоковой обработки для создания платформы, на основе которой можно конструировать масштабируемые, отзывчивые и отказоустойчивые решения задач обработки гигантских объемов данных.

Параллелизм позволяет обрабатывать гигантские объемы данных

Возникновение гигантских объемов данных привело к появлению великого множества изменений в технологиях обработки данных за последние годы. Обработка гигантских объемов отличается от традиционной обработки данных широким использованием параллелизма — только пуская в ход многочисленные вычислительные ресурсы, можно рассчитывать на успех в обработке терабайтов информации. Лямбда-архитектура является одним из подходов к обработке гигантских объемов данных, придуманным Натаном Марцем (Nathan Marz) в бытность его работы в BackType и затем в Twitter.

Подобно программированию GPGPU — модели, рассматривавшейся на прошлой неделе, — лямбда-архитектура опирается на параллельную обработку данных. Разница лишь в том, что она предполагает обширное масштабирование, то есть распределение данных и

вычислений по кластерам из десятков и сотен компьютеров. Это не только позволяет привлечь достаточный объем вычислительных ресурсов для решения ранее непосильных задач, но также дает возможность создавать системы, устойчивые к отказам аппаратной части или ошибкам человека.

Лямбда-архитектура имеет множество аспектов. В этой главе мы сосредоточимся только на аспектах параллельной и распределенной обработки (более полное обсуждение лямбда-архитектуры можно найти в книге Натана «Big Data» [MW14]). В частности, мы рассмотрим два основных элемента, *пакетный уровень* (batch layer) и *уровень ускорения* (speed layer), как показано на рис. 37.

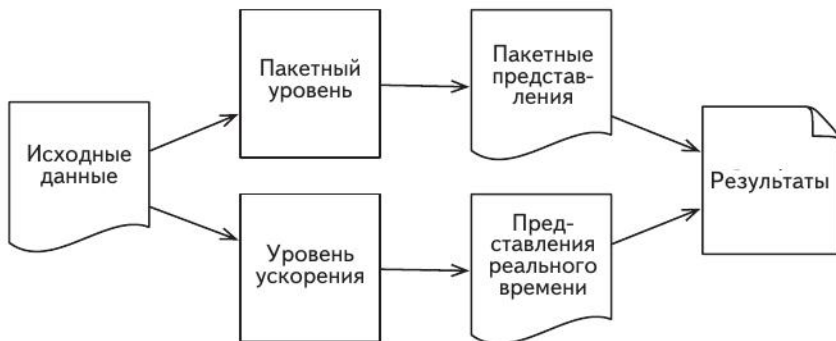


Рис. 37. Пакетный уровень и уровень ускорения

На пакетном уровне используются пакетные технологии, такие как MapReduce, для создания предвычисленных пакетных представлений из ранее полученных данных. Это достаточно эффективное решение, но оно имеет большую задержку, поэтому к пакетному уровню добавляется уровень ускорения, в котором используются технологии с низкой задержкой, такие как потоковая обработка, для создания представлений реального времени из вновь поступающих данных. Затем представления двух типов объединяются и возвращаются в ответ на запросы.

Лямбда-архитектура является самой сложной темой из охватываемых в этой книге. Она опирается на несколько низкоуровневых технологий, наиболее важной из которых является MapReduce. Поэтому в первый день мы сосредоточимся исключительно на MapReduce, не заботясь о ее месте в общей картине. Во второй день мы рассмотрим проблемы традиционных систем обработки данных и узнаем, как технология MapReduce помогает решать их в пакетном уровне лямбда-

архитектуры. Наконец, в третий день мы дополним картину лямбда-архитектуры знакомством с потоковой обработкой, и я покажу, как этот прием используется в уровне ускорения.

День 1: MapReduce

MapReduce – широкий термин. Иногда он используется для описания распространенного шаблона разбиения алгоритма на два этапа: отображения данных с последующей сверткой. Наша функциональная реализация подсчета слов (см. раздел «Функциональный подсчет слов» в главе 3) как раз является примером использования этого шаблона (если вы помните, подсчет слов в том примере был реализован с применением функции `reduce`). Как мы видели в разделе «День 2: функциональный параллелизм» в главе 3, одно из преимуществ такого разбиения алгоритма как раз и заключается в возможности распараллелить обработку данных.

Но термин *MapReduce* может также обозначать нечто более узкоспециализированное – систему, принимающую алгоритм, реализованный как операция отображения с последующей операцией свертки, и эффективно распределяющую его по компьютерам в кластере. Такая система способна не только автоматически распределить данные и их обработку между компьютерами в кластере, но также продолжить работу, если один или несколько из этих компьютеров выйдут из строя.



Вопрос Джо: почему было выбрано такое странное название?

Существует масса предположений, откуда взялось такое название, поэтому самое лучшее, что я могу сделать в ответ на этот вопрос, – привести цитату создателя модели лямбда-архитектуры, Натана Маца (Nathan Marz):*

Такое название обусловлено наличием большого числа общих черт между архитектурой и функциональным программированием. На самом фундаментальном уровне лямбда-архитектура является обобщенной технологией вычисления функций на всех данных сразу.

* <http://www.manning-sandbox.com/message.jspa?messageID=126599>

Технология MapReduce в этом, более узком смысле была впервые разработана в Google.¹ Наиболее популярной реализацией MapReduce, созданной за пределами Google, является фреймворк Hadoop.²

Сегодня мы будем использовать Hadoop для создания параллельной MapReduce-версии примера подсчета слов в Википедии. Hadoop поддерживает множество языков программирования. Мы же будем использовать язык Java.

Практические аспекты

Установка и настройка Hadoop для использования на локальном компьютере является достаточно простой задачей и обычной отправной точкой для желающих заняться разработкой и отладкой программ, использующих технологию MapReduce. Однако использование этого фреймворка на кластере может вызвать массу сложностей – далеко не каждый из нас имеет груды запасных компьютеров, только и ждущих, когда их объединят в кластер. Но даже если у кого-то они и есть, создание, настройка и обслуживание кластера Hadoop требуют массу времени и обширных познаний.

К счастью, появление технологий облачных вычислений существенно улучшило положение дел – они позволяют получить доступ к виртуальным серверам по требованию и по времени. Более того, многие провайдеры теперь предлагают услуги кластеров Hadoop, принимая на себя все хлопоты, связанные с настройкой и обслуживанием.

Для опробования примеров в этой главе мы будем пользоваться услугами веб-службы Amazon Elastic MapReduce, или EMR.³ Приемы, с помощью которых мы будем запускать и останавливать кластеры, копировать данные туда и обратно, являются характерными для EMR, но общие принципы применимы к любым кластерам Hadoop.

Чтобы самостоятельно опробовать примеры, вам потребуется создать учетную запись Amazon AWS и установить инструменты командной строки AWS⁴ и EMR⁵.

¹ <http://research.google.com/archive/mapreduce.html>

² <http://hadoop.apache.org>

³ <http://aws.amazon.com/elasticmapreduce/>

⁴ <http://aws.amazon.com/cli/>

⁵ <http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-cli-reference.html>



Вопрос Джо: какие версии Hadoop существуют?

Hadoop имеет весьма запутанную схему нумерации версий, все версии – 0.20.x, 1.x, 0.22.x, 0.23.x, 2.0.x, 2.1.x и 2.2.x – продолжали активно разрабатываться на момент написания этих строк. Данные версии поддерживают два разных API, которые обычно называют «старый» (пакет `org.apache.hadoop.mapred`) и «новый» (пакет `org.apache.hadoop.mapreduce`).

Кроме того, в состав дистрибутивов с разными версиями Hadoop включены разные наборы дополнительных компонентов, созданных сторонними разработчиками.*

Во всех примерах в этой главе мы будем использовать новый API и опробовать их на Amazon 3.0.2 AMI, где используется версия Hadoop 2.2.0.**

* <http://hortonworks.com>, <http://www.cloudera.com>, <http://www.mapr.com>

** <http://docs.aws.amazon.com/ElasticMapReduce/latest/DeveloperGuide/emr-plan-hadoop-version.html>

Основы Hadoop

Цель создания фреймворка Hadoop – обеспечить возможность обработки гигантских объемов данных. Если объемы ваших данных не измеряются гигабайтами или более крупными единицами измерения, можете уверенно считать, что Hadoop – не ваш инструмент. Его мощь обусловлена тем, что он разбивает данные на фрагменты и обрабатывает эти фрагменты независимо, на отдельных компьютерах.

Как можно догадаться, решение с применением технологии MapReduce состоит из компонентов двух основных типов: средств отображения и средств свертки. Средства отображения принимают исходные данные в некотором формате (по умолчанию обычные текстовые строки) и отображают их во множество пар ключ/значение. Затем средства свертки преобразуют эти пары ключ/значение в окончательный выходной формат (часто также множество пар ключ/значение). Средства отображения и средства свертки распределяются среди множества компьютеров (нет никаких требований, которые предписывали бы, чтобы число компьютеров, осуществляющих отображение, совпадало с числом компьютеров, осуществляющих свертку), как показано на рис. 38.

Исходные данные обычно заключены в один или более больших текстовых файлов. Hadoop разбивает эти файлы на фрагменты (раз-

меры фрагментов зависят от настроек фреймворка, но чаще всего используются фрагменты по 64 Мбайт в каждом) и распределяет их между средствами отображения. Компьютер, осуществляющий отображение, производит множество пар ключ/значение, которые затем передаются фреймворком на компьютеры, осуществляющие свертку.

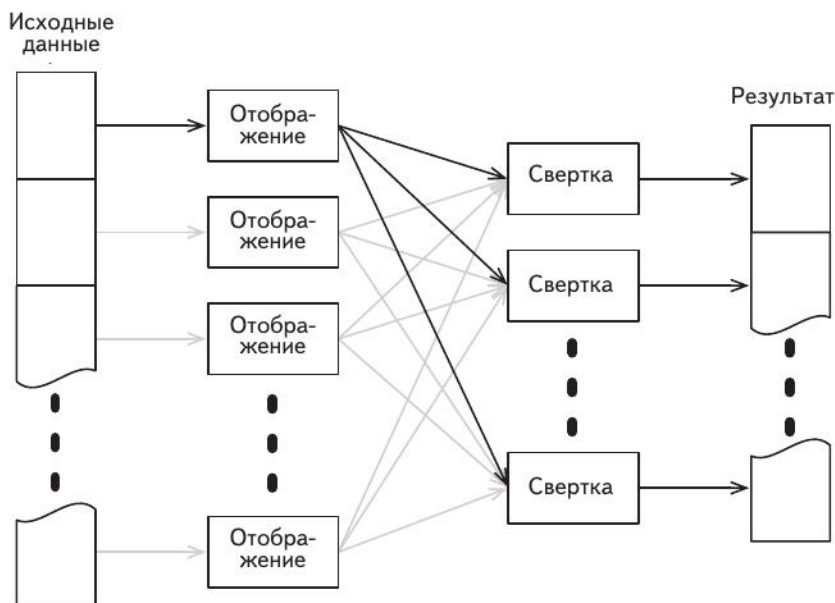


Рис. 38. Общая схема работы фреймворка Nadoop

Пары ключ/значение от одного компьютера, выполняющего отображение, передаются множеству компьютеров, выполняющих свертку. Выбор компьютера, кому какую пару послать, определяется ключом – Nadoop гарантирует, что все пары с одним и тем же ключом будут обработаны на одном и том же компьютере, независимо от того, на каком компьютере, выполняющем отображение, они будут произведены. Совершенно понятно, почему эта фаза называется фазой перемешивания.

Nadoop обращается к компьютеру, выполняющему свертку, один раз для каждого ключа, передавая сразу весь список значений для этого ключа. Компьютер, выполняющий свертку, объединяет эти значения и генерирует окончательный результат (обычно, но не обязательно, также является множеством пар ключ/значение).

Но достаточно теории – перейдем к практике и создадим Hadoop-версию примера подсчета слов в Википедии.

Подсчет слов с помощью Hadoop

Сначала мы решим задачу, несколько упростив ее, – подсчитаем количество слов в коллекции простых текстовых файлов (как распространить найденное решение на дампы Википедии в формате XML, мы увидим потом).

Наш инструмент отображения будет обрабатывать текст построчно, разбивая каждую строку на слова и генерируя пару ключ/значение для каждого слова. Ключом будет служить само слово, а значением – число 1. Инструмент свертки будет принимать все пары ключ/значение для данного слова и суммировать значения, генерируя для каждого слова единственную пару ключ/значение, где значением будет число – частота встречаемости слова в исходном тексте, как показано на рис. 39.

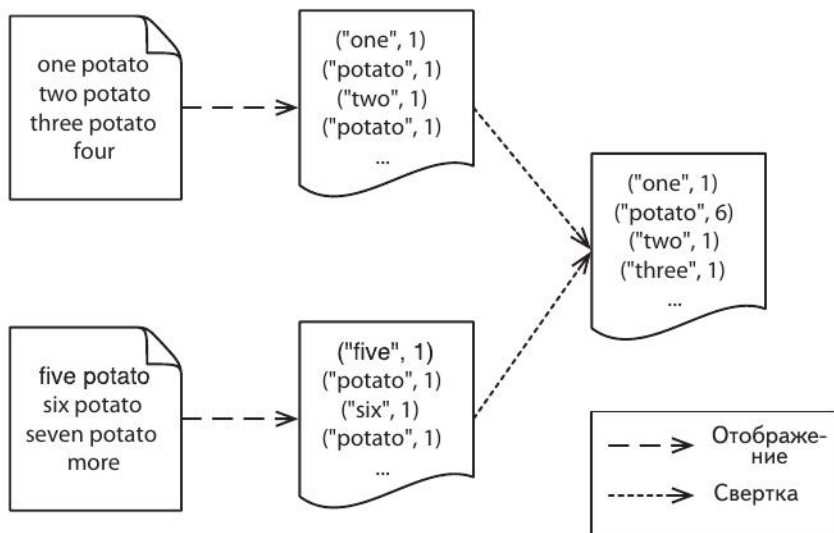


Рис. 39. Подсчет слов с помощью Hadoop

Отображение

Отображение в нашей программе будет выполнять класс `Map`. Он наследует класс `Mapper` из фреймворка Hadoop, параметризованный четырьмя типами – типом входного ключа, типом входного значения, типом выходного ключа и типом выходного значения:

LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/
WordCount.java⁶

```
1 public static class Map extends Mapper<Object, Text, Text, IntWritable> {
-   private final static IntWritable one = new IntWritable(1);
-
-   public void map(Object key, Text value, Context context)
5       throws IOException, InterruptedException {
-
-       String line = value.toString();
-       Iterable<String> words = new Words(line);
-       for (String word: words)
10          context.write(new Text(word), one);
-   }
- }
```

Фреймворк Hadoop использует собственные типы для представления входных и выходных данных (мы не можем использовать простые типы `String` и `Integer`). Наш класс, осуществляющий отображение, будет обрабатывать простые текстовые данные, а не пары ключ/значение, поэтому тип входного ключа не используется (мы передали тип `Object`), а в качестве типа входного значения указан тип `Text`. Выходной ключ также будет иметь тип `Text`, а выходное значение — тип `IntWritable`.

Метод `map()` будет вызываться по одному разу для каждой исходной строки. Он сначала преобразует текстовую строку в объект типа `String` (строка 7) и затем разбивает строку на слова (строка 8). Наконец, выполняет итерации по словам и для каждого из них генерирует пару ключ/значение, где ключом служит само слово, а значением — целое число 1 (строка 10).

Свертка

Класс `Reduce`, реализующий свертку, наследует класс `Reducer` из фреймворка Hadoop. Подобно классу, он также параметризован четырьмя типами — типами входных и выходных ключей и значений (в нашем случае оба ключа имеют тип `Text` и оба значения имеют тип `IntWritable`):

LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/
WordCount.java⁷

```
public static class Reduce extends Reducer<Text, IntWritable, Text,
IntWritable> {
```

⁶ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/WordCount.java>

⁷ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/WordCount.java>

```

public void reduce(Text key, Iterable<IntWritable> values, Context
context)
    throws IOException, InterruptedException {
    int sum = 0;
    for (IntWritable val: values)
        sum += val.get();
    context.write(key, new IntWritable(sum));
}
}

```

Метод `reduce()` вызывается по одному разу для каждого ключа, со значением-коллекцией, содержащей все значения, соответствующие этому ключу. Наша реализация свертки просто суммирует значения из коллекции и генерирует единственную пару ключ/значение, описывающую число вхождений слова.

Теперь, когда созданы инструменты отображения и свертки, нам осталось реализовать драйвер, который сообщит фреймворку Hadoop, как пользоваться этими инструментами.

Драйвер

Драйвер реализует метод `run()` интерфейса `Tool`:

LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/
WordCount.java⁸

```

1 public class WordCount extends Configured implements Tool {
-
-     public int run(String[] args) throws Exception {
-         Configuration conf = getConf();
5         Job job = Job.getInstance(conf, "wordcount");
-         job.setJarByClass(WordCount.class);
-         job.setMapperClass(Map.class);
-         job.setReducerClass(Reduce.class);
-         job.setOutputKeyClass(Text.class);
10        job.setOutputValueClass(IntWritable.class);
-         FileInputFormat.addInputPath(job, new Path(args[0]));
-         FileOutputFormat.setOutputPath(job, new Path(args[1]));
-         boolean success = job.waitForCompletion(true);
-         return success ? 0 : 1;
15    }
-
-     public static void main(String[] args) throws Exception {
-         int res = ToolRunner.run(new Configuration(), new WordCount(),
args);
-         System.exit(res);

```

⁸ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCount/src/main/java/com/paulbutcher/WordCount.java>

```
20    }
    - }
```

В большинстве своем это шаблонный код, который просто информирует Hadoop о том, что он должен делать. В строках 7 и 8 определяются классы, реализующие отображение и свертку, а в строках 9 и 10 определяются типы выходных ключей и значений. Определять типы входных ключей и значений не требуется, потому что Hadoop предполагает по умолчанию, что исходные данные поставляются в виде простых текстовых файлов. И нам не требуется отдельно определять типы выходных ключей/значений для класса `Map` и типы входных ключей/значений для класса `Reduce`, потому что по умолчанию Hadoop предполагает, что они совпадают с типами выходных ключей/значений.

Далее, в строках 11 и 12, фреймворку Hadoop сообщается, где искать исходные данные и куда записывать результаты. После этого в строке 13 осуществляется запуск задания, и выполнение программы приостанавливается до его завершения.

Теперь нам осталось опробовать пример на некоторых данных.

Опробование на локальном компьютере

Сначала попробуем обработать данные на локальном компьютере. При этом мы не получим никаких выгод от параллелизма или повышенной отказоустойчивости, но это даст нам возможность убедиться, что программный код работает, прежде чем прилагать дополнительные усилия для запуска примера на кластере.

Прежде всего попробуем обработать небольшой объем текстовых данных. В каталоге `input` имеются два текстовых файла с литературным шедевром, который мы будем анализировать:

```
LambdaArchitecture/WordCount/input/file1.tx9t
```

```
one potato two potato three potato four
```

```
LambdaArchitecture/WordCount/input/file2.tx10t
```

```
five potato six potato seven potato more
```

⁹ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCount/input/file1.txt>

¹⁰ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCount/input/file2.txt>

Конечно, это не гигабайты данных, но и этого вполне достаточно, чтобы убедиться в работоспособности кода. Мы можем подсчитать слова в этих файлах, выполнив сборку командой `mvn package` и затем запустив локальный экземпляр Hadoop:

```
$ hadoop jar target/wordcount-1.0-jar-with-dependencies.jar input output
```

После того как фреймворк завершит работу, должен появиться новый каталог `output` с двумя файлами в нем — `_SUCCESS` и `part-r-00000`. Первый — пустой файл, он просто служит признаком успешного выполнения задания. Второй должен содержать следующее:

```
five 1
four 1
more 1
one 1
potato 6
seven 1
six 1
three 1
two 1
```

Теперь, когда мы убедились, что наш код успешно справляется с небольшими локальными файлами, можно попробовать запустить его на кластере и обработать гораздо больший объем данных.



Вопрос Джо: результаты всегда сортируются?

Возможно, вы заметили, что результаты были сохранены в порядке (алфавитном) следования ключей. Фреймворк Hadoop гарантирует, что ключи будут отсортированы перед передачей механизму свертки. Это обстоятельство упрощает решение многих задач.

Но будьте внимательны, несмотря на то что ключи сортируются перед сверткой, сами инструменты свертки необязательно будут вызываться в том же порядке, в чем вы убедитесь далее. Порядок вызова инструментов свертки определяется настройками планировщика, однако мы не будем углубляться в эту тему здесь.

Опробование на Amazon EMR

Чтобы запустить задание для Hadoop на Amazon Elastic MapReduce, требуется выполнить несколько дополнительных шагов. Мы не бу-

дем глубоко погружаться в исследование EMR. Но мне хотелось бы рассказать об упомянутых шагах хотя бы самый минимум, чтобы вы смогли проделать их самостоятельно.

Входные и выходные данные

По умолчанию EMR принимает входные данные и выводит выходные посредством веб-службы Amazon S3.¹¹ Веб-служба S3 также служит хранилищем JAR-файла с выполняемым кодом и файлов журналов.

Соответственно, чтобы запустить задание, нам нужно сохранить в S3 некоторые текстовые файлы. Дамп Википедии в исходном виде не подходит для этого, потому что он хранится в формате XML. В составе загружаемого пакета с примерами для этой книги имеется проект программы `ExtractWikiText`, извлекающей текст из дампа Википедии, который можно выгрузить в хранилище S3. Вам также потребуется выгрузить JAR-файл в другое хранилище S3.

Выгрузка файлов в S3

Если ваше «широко»полосное соединение с Интернетом, как и у меня, начинает «задыхаться» при попытке выгрузить большой файл, у вас может появиться мысль о создании кратковременного экземпляра Amazon EC2, с помощью которого можно будет загрузить файл Википедии, извлечь текст из него и выгрузить его в S3. Едва ли кого-то удивит, если я скажу, что службы EC2 и S3 связаны сетью с превосходной пропускной способностью, что поможет вам сохранить румянец на щеках вашего широкополосного соединения.

Создание кластера

Создать кластер EMR можно разными способами, мы же воспользуемся инструментом командной строки `elastic-mapreduce`:

```
$ elastic-mapreduce --create --name wordcount --num-instances 11 \
--master-instance-type m1.large --slave-instance-type m1.large \
--ami-version 3.0.2 --jar s3://pb7con-lambda/wordcount.jar \
--arg s3://pb7con-wikipedia/text --arg s3://pb7con-wikipedia/counts
Created job flow j-2LSRGPBSR79ZV
```

¹¹ <http://aws.amazon.com/s3/>

Эта команда создаст кластер с именем «wordcount», состоящий из 11 узлов, — 1 главный узел и 10 подчиненных узлов — все типа `m1.large` и выполняются на образе машины версии 3.0.2 (AMI).¹² Последний аргумент сообщает EMR, где искать JAR-файл, выгруженный нами в S3, где искать входные данные и куда сохранить результаты.

Слежение за процессом выполнения

Чтобы установить SSH-соединение с главным узлом кластера, можно использовать идентификатор потока заданий (job flow), полученный при создании кластера:

```
$ elastic-mapreduce --jobflow j-2LSRGPBSR79ZV --ssh
```

После получения доступа к командной строке главного узла можно осуществлять слежение за выполнением задания путем просмотра файлов журналов:

```
$ tail -f /mnt/var/log/hadoop/steps/1/syslog
INFO org.apache.hadoop.mapreduce.Job (main): map 0% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 1% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 2% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 3% reduce 0%
INFO org.apache.hadoop.mapreduce.Job (main): map 4% reduce 0%
```

Исследование результатов

В моих тестах эта конфигурация позволила получить результаты подсчета слов в Википедии за час с небольшим. По завершении вы должны обнаружить множество файлов в своем хранилище S3:

```
part-r-00000
part-r-00001
part-r-00002
.
.
.
part-r-00028
```

Эти файлы все вместе хранят полный комплект результатов. Результаты отсортированы внутри каждого раздела, но не отсортированы между разделами (см. врезку «Результаты всегда сортируются?» выше).

Теперь мы можем выполнять подсчет слов в простых текстовых файлах, но в идеале хотелось бы иметь возможность обрабатывать

¹² <http://aws.amazon.com/ec2/instance-types/>

дамп Википедии непосредственно. Как это сделать, рассказывается далее.

Обработка XML

Файл XML, в конце концов, — это всего лишь текстовый файл, имеющий определенную структуру, поэтому будет простительно, если вы подумаете, что его можно обрабатывать почти так же, как было показано выше. Однако этот прием не сработает, потому что по умолчанию Hadoop разбивает файлы по границам строк, которые могут приходиться на середину тегов XML.

Несмотря на то что Hadoop не поддерживает разбиение XML-файлов по умолчанию, в компании Apache существует, как оказывается, еще один проект, Mahout,¹³ реализующий такую поддержку в виде класса `XmlInputFormat`.¹⁴ Чтобы воспользоваться этой поддержкой, нужно внести в наш драйвер несколько небольших изменений:

```
LambdaArchitecture/WordCountXml/src/main/java/com/paulbutcher/  
WordCount.java15
```

```
1 public int run(String[] args) throws Exception {  
-   Configuration conf = getConf();  
-   conf.set("xmlinput.start", "<text");  
-   conf.set("xmlinput.end", "</text>");  
5  
-   Job job = Job.getInstance(conf, "wordcount");  
-   job.setJarByClass(WordCount.class);  
-   job.setInputFormatClass(XmlInputFormat.class);  
-   job.setMapperClass(Map.class);  
10 job.setCombinerClass(Reduce.class);  
-   job.setReducerClass(Reduce.class);  
-   job.setOutputKeyClass(Text.class);  
-   job.setOutputValueClass(IntWritable.class);  
-   FileInputFormat.addInputPath(job, new Path(args[0]));  
15 FileOutputFormat.setOutputPath(job, new Path(args[1]));  
-  
-   boolean success = job.waitForCompletion(true);  
-   return success ? 0 : 1;  
- }
```

¹³ <http://mahout.apache.org>

¹⁴ <https://github.com/apache/mahout/blob/trunk/integration/src/main/java/org/apache/mahout/text/wikipedia/XmlInputFormat.java>

¹⁵ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCountXml/src/main/java/com/paulbutcher/WordCount.java>

С помощью вызова `setInputFormatClass()` (строка 8) мы сообщаем фреймворку Hadoop, что разбиение исходных данных на файлы должно производиться с помощью `XmlInputFormat`, и записываем в конфигурацию параметры `xmlinput.start` и `xmlinput.end` (строки 3 и 4), определяющие начальный и конечный теги XML-документа.

Если внимательно взглянуть на значение, которое назначается для параметра `xmlinput.start`, может показаться, что в нем допущена ошибка — мы указали незаконченный тег XML `<text`. Дело в том, что класс `XmlInputFormat` не выполняет полного парсинга XML; вместо этого он просто ищет начало и конец по указанным шаблонам. Так как тег `<text>` может иметь атрибуты, нам нужно искать последовательность символов `<text`, а не `<text>`.

Нам также нужно немного изменить нашу реализацию отображения:

```
LambdaArchitecture/WordCountXml/src/main/java/com/paulbutcher/  
WordCount.java16
```

```
private final static Pattern textPattern =  
    Pattern.compile("^<text.*>(.*?)</text>$", Pattern.DOTALL);  
  
public void map(Object key, Text value, Context context)  
    throws IOException, InterruptedException {  
  
    String text = value.toString();  
    Matcher matcher = textPattern.matcher(text);  
    if (matcher.find()) {  
        Iterable<String> words = new Words(matcher.group(1));  
        for (String word: words)  
            context.write(new Text(word), one);  
    }  
}
```

Все разбиения выполняются между шаблонами `xmlinput.start` и `xmlinput.end`, включая совпадения с шаблонами. Поэтому мы используем некоторое волшебство регулярных выражений, чтобы избавиться от тегов `<text></text>` перед подсчетом слов (и избежать излишнего подсчета слова *text*).

Возможно, вы обратили внимание на еще одну особенность нашего нового драйвера — мы определили *комбинатор* (*combiner*) методом `setCombinerClass()` (строка 10). Комбинатор — это инструмент оптимизации, позволяющий объединять пары ключ/значение перед

¹⁶ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WordCountXml/src/main/java/com/paulbutcher/WordCount.java>

отправкой их инструменту свертки (см. рис. 40). В моих тестах применение комбинатора помогло уменьшить время выполнения с часа с небольшим до примерно 45 минут.

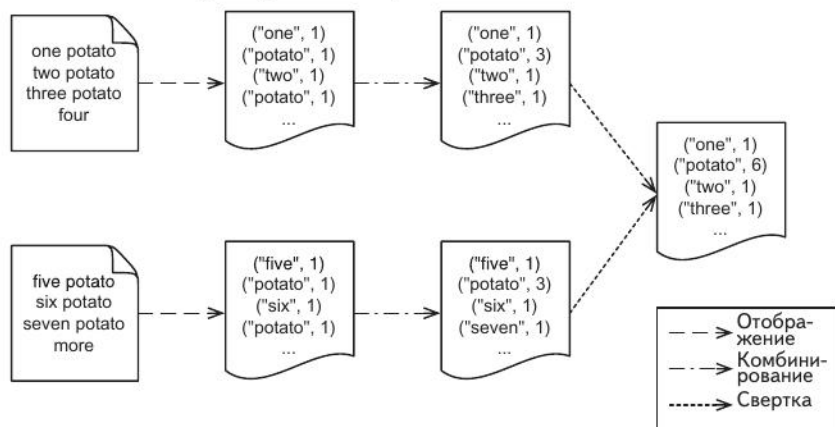


Рис. 40. Применение комбинатора

В нашем случае на роль комбинатора отлично подходит инструмент свертки, но в некоторых алгоритмах может потребоваться иная реализация комбинатора. Фреймворк Hadoop не гарантирует использования комбинатора, даже если вы явно укажете его, поэтому алгоритм обработки данных не должен зависеть от применения или неприменения комбинатора.

В завершение первого дня

Вот и закончился первый день. Завтра мы увидим, как с помощью Hadoop сконструировать пакетный уровень лямбда-архитектуры.

Что мы узнали в первый день

Деление задачи на два этапа – отображение в структуру данных и свертку – упрощает ее распараллеливание. Термин MapReduce в том смысле, в котором мы использовали его в этой главе, обозначает, в частности, систему, эффективно и надежно распределяющую задания по отображению и свертке среди множества компьютеров. Фреймворк Hadoop является системой MapReduce, которая:

- разбивает исходные данные между несколькими компьютерами, осуществляющими отображение, каждый из которых генерирует пары ключ/значение;

- полученные пары передаются компьютерам, осуществляющим свертку и генерирующим окончательные результаты (обычно так же в виде пар ключ/значение);
- отображенные данные так разбиваются между компьютерами, реализующими второй этап, что все пары с одним ключом обрабатываются одним и тем же компьютером.



Вопрос Джо: все это делается только для увеличения скорости обработки?

Ошибочно было бы думать, что Hadoop дает нам только скорость обработки, позволяя тем самым обрабатывать огромные объемы данных быстрее, чем это возможно на одном компьютере. Безусловно, это очень важное преимущество, но есть еще кое-что:

- когда мы начинаем говорить о кластерах из сотен компьютеров, проблема отказоустойчивости теряет свою остроту и становится вполне терпимой. Любая система, испытывающая проблемы из-за выхода из строя одного компьютера в кластере, быстро может стать неработоспособной. По этой причине фреймворк Hadoop изначально создавался с прицелом на поддержание высокой отказоустойчивости;
- учитывая вышесказанное, мы должны подумать не только о том, как передать на другой компьютер задание, выполнявшееся на вышедшем из строя компьютере, но также о том, как избежать потери данных, если один или более жестких дисков выйдут из строя. По умолчанию фреймворк Hadoop использует распределенную файловую систему Hadoop (HDFS), отказоустойчивую распределенную файловую систему, копирующую данные на множество узлов;
- как только мы начинаем говорить об объемах данных, измеряемых гигабайтами или более крупными единицами измерения, было бы неоправданно ожидать, что промежуточные данные или результаты поместятся в память. Фреймворк Hadoop хранит пары ключ/значение в файловой системе HDFS, позволяя создавать задания, обрабатывающие намного более объемные наборы данных, чем можно уместить в памяти.

Все эти аспекты являются главными. Не случайно, что только в этой главе мы выполнили подсчет слов по всему дампу Википедии, – технология MapReduce – единственная из рассматриваемых в этой книге, позволяющая обработать такой объем данных.

Упражнения к первому дню

Найдите:

- документацию с описанием потокового API фреймворка Hadoop, позволяющего создавать задания в стиле MapReduce на таких языках, как Ruby, Python или Perl;
- документацию с описанием API каналов фреймворка Hadoop, позволяющего создавать задания в стиле MapReduce на C++;
- на основе Hadoop Java API создано множество библиотек, упрощающих создание весьма сложных заданий в стиле MapReduce. Например, вас могут заинтересовать такие библиотеки, как Cascading, Cascalog или Scalding.

Задания:

- во время подсчета слов остановите один из компьютеров в кластере (только не главный компьютер – фреймворк Hadoop неспособен противостоять отказам главного компьютера). Исследуйте файлы журналов, чтобы увидеть, как Hadoop пытается перераспределить работу, предназначенную для этого компьютера. Убедитесь, что результаты получились теми же и выход одного компьютера из строя не повлиял на них;
- наша программа подсчета слов хорошо справляется с задачей, которую мы на нее возложили, но от нее мало проку, когда, например, нужно узнать 100 наиболее часто встречающихся слов. Реализуйте вторую сортировку (в Интернете можно найти массу статей, описывающих, как это сделать), чтобы получить полностью отсортированный вывод от нашей программы;
- шаблон «лучшая десятка» является альтернативным способом решить задачу выявления наиболее употребимых слов. Создайте версию программы подсчета слов, используя этот шаблон;
- не все задачи могут быть решены с применением единственного задания MapReduce – часто бывает необходимо объединить в цепочку несколько заданий, когда результаты одного задания являются исходными данными для следующего. Одним из примеров является алгоритм PageRank. Напишите программу на основе фреймворка Hadoop, вычисляющую ранг каждой страницы в Википедии. Подсчитайте, сколько итераций потребуется выполнить программе, прежде чем данные стабилизируются.

День 2: пакетный уровень

Вчера мы видели, как с помощью фреймворка Hadoop распределить работу в кластере компьютеров. Технологию MapReduce можно использовать для решения самых разных задач, но сегодня мы сосредоточимся на ее месте в лямбда-архитектуре.

Однако перед этим давайте рассмотрим проблему, для решения которой создавалась лямбда-архитектура, — что неправильно в традиционных системах данных?

Проблемы с традиционными системами данных

Системы данных не новы — базы данных, помогающие обрабатывать данные, появились почти одновременно с компьютерами. Традиционные базы данных хорошо справляются со своим предназначением, но объемы информации, которые требуется обрабатывать в наши дни, порой так велики, что не умещаются в базах данных.

Масштабирование

Существуют приемы, позволяющие масштабировать базы данных за рамки одного компьютера (репликация (replication), шардинг (sharding) и др.), но сложность их применения возрастает с ростом числа компьютеров, и они приводят к увеличению числа запросов. Кроме того, после некоторого предела они перестают давать выигрыш.

Администрирование

Администрирование базы данных, разбросанной по нескольким компьютерам, сопряжено с большими сложностями. Осуществлять профилактические работы без остановки этих компьютеров, например чтобы выполнить решардинг, еще сложнее. Кроме того, обеспечивать отказоустойчивость, проводить резервное копирование и гарантировать целостность данных становится все сложнее с ростом объема хранимых данных.

Сложность

Репликация и шардинг обычно требуют поддержки на прикладном уровне — приложения должны знать, к каким серверам направлять запросы и какие фрагменты обновлять (обычно разные запросы имеют

довольно неочевидные отличия). Зачастую программист лишается многих привычных средств поддержки, например в случае шардинга базы данных исчезает поддержка транзакций, из-за чего программист вынужден явно обрабатывать отказы и выполнять повторные попытки. Все это увеличивает вероятность появления ошибок.

Человеческие ошибки

Многие часто забывают еще один немаловажный аспект отказоустойчивости – человеческие ошибки. Большинство случаев повреждения данных связано не с выходами из строя дисков, а с ошибками, допущенными администратором или разработчиком. Если вам повезет, вы быстро найдете и устраните причину и восстановите данные из резервной копии, но не все ошибки обнаруживаются легко и просто. Представьте, что вы столкнулись с ошибкой, вызывающей масштабные повреждения данных, выявить которую не получается вот уже пару недель. Как вы собираетесь восстанавливать базу данных?

Иногда, чтобы ликвидировать повреждения, достаточно выяснить эффект, производимый ошибкой, и написать одноразовый сценарий, исправляющий базу данных. Бывает, что исправить повреждения можно, откатив изменения по файлам журналов (предполагается, что файлы журналов хранят всю необходимую информацию). А иногда удача просто покидает вас. Полагаться на удачу – не самая лучшая стратегия.

Журналирование и анализ

Традиционные базы данных отлично подходят для поддержки операций, например повседневных сделок в коммерческой организации. Но они намного менее эффективны, когда речь заходит о журналировании и анализе, требующих доступа к исторической информации.

Типичное решение заключается в том, чтобы создать отдельное хранилище данных для исторической информации. Данные в этом случае перемещаются из операционной базы данных в хранилище в ходе процесса, известного как «извлечение, преобразование, загрузка» (Extract, Transform, Load – ETL). Мало того, что это добавляет сложностей, но еще и зависит от точности выбора информации, которая может потребоваться в будущем, – не так уж редко возникают ситуации, когда необходимые сведения оказываются недоступны просто потому, что вы не предусмотрели ее сохранения.

В следующем разделе мы посмотрим, как лямбда-архитектура решает все эти проблемы. Она не только позволяет обрабатывать огромные объемы данных, с которыми сталкиваются современные приложения, но также упрощает эту обработку, обеспечивая устойчивость к аппаратным отказам и человеческим ошибкам, а также хранит полный объем исторической информации, что позволяет проводить в будущем любой анализ, какой только потребуется.

Вечные истины

Информацию можно разделить на две категории – исходные данные и производная информация.

Возьмем в качестве примера страницу в Википедии – страницы постоянно обновляются и дополняются, то есть любая конкретная страница завтра может отличаться от того, что я вижу сегодня. Но страницы не являются исходными данными, из которых состоит Википедия, – любая страница является результатом правок нескольких авторов. Эти правки как раз и являются исходными данными, а страница является производной этих правок.

Кроме того, страницы могут изменяться каждый день, а правки – нет. Как только автор выполнит правку, она больше никогда не изменится. Другой автор может добавить свою правку, изменяющую или отменяющую результат предыдущей, из-за чего изменится и порожденная страница, но сами правки остаются неизменными.

Подобное поведение можно реализовать в любой системе данных. Баланс вашего банковского счета является производной последовательности зачислений и списаний. График друзей в Facebook является производной от последовательности событий предложений дружбы и отказа от нее. И, подобно правкам в Википедии, зачисления и списания, события предложения дружбы и отказа от нее являются неизменными.

Это утверждение, что исходные данные являются вечной истиной, составляет фундаментальную основу лямбда-архитектуры. В следующем разделе мы посмотрим, как, опираясь на данное утверждение, можно решать проблемы, свойственные традиционным системам данных.

Лучшие данные – исходные данные

А теперь наведите уши. Как мы не раз видели в предыдущих главах, неизменяемость и параллелизм – это союз двоих, освященный на небесах.



Вопрос Джо: исходные данные действительно не изменяются?

Поначалу бывает трудно увидеть, как некоторые виды исходных данных могут быть вечной истиной. Например, что вы думаете об адресе проживания? Что случится, если человек переедет в другой дом?

Это ничего не меняет, исходная информация по-прежнему остается неизменной – нам просто нужно добавить привязку ко времени. Вместо записи «Шарлотта живет на Акация-авеню, 22» нам нужно создать запись «1 марта 1982 года Шарлотта жила на Акация-авеню, 22». И эта информация останется истинной в любой момент в будущем.

Приятные мечты

Давайте немного помечтаем. Представьте, что у нас есть бесконечно быстрый компьютер, способный мгновенно обрабатывать терабайты данных. Мы могли бы тогда хранить только исходные данные – нам не нужно было бы хранить производную информацию, потому что мы могли бы мгновенно получить ее в любой момент.

В этих мечтах мы одним махом избавились от большей части сложностей, присущих традиционным базам данных, потому что когда данные остаются неизменяемыми, организация их хранения превращается в тривиальную задачу. Наше хранилище должно лишь позволять добавлять новые данные – нам не нужно разрабатывать механизмы блокировок или транзакций, потому что после сохранения данных в хранилище они больше никогда не изменятся.

Более того. Когда данные остаются неизменными, к ним одновременно может обращаться множество потоков выполнения, не опасаясь конфликтов друг с другом. Мы можем извлекать копии данных и оперировать этими копиями, не волнуясь о том, что они устареют, благодаря чему распределение данных между компьютерами в кластере превращается в намного более простую задачу.

К сожалению, мы живем в реальном мире, но я думаю, что смогу удивить вас, когда покажу, насколько близко мы можем приблизиться к мечтам, взяв на вооружение технологию MapReduce.

Мечты становятся реальностью (почти)

Если заранее известно, какие запросы будут выполняться к исходным данным, мы можем заранее вычислить *пакетное представление*

(batch view), которое или непосредственно содержит производные данные, возвращаемые этими запросами, или содержит данные, которые легко можно объединить для получения желаемых результатов. Вычисление пакетных представлений является задачей пакетного уровня лямбда-архитектуры.



Вопрос Джо: а как же удаление данных?

Иногда есть серьезные причины удалить исходные данные. Это может потребоваться, например, если данные пережили свою ценность или их хранение может противоречить законодательству (например, законы о защите информации могут запрещать хранение некоторых данных сверх установленного периода).

Это не отменяет всего, что было сказано выше. Данные, выбранные для удаления, все еще остаются вечной истиной, даже если мы предпочли забыть о них.

В качестве первого примера пакетного представления рассмотрим создание страниц Википедии из последовательности правок – пакетное представление будет просто конструировать текст каждой страницы из всех имеющихся правок.

Второй пример пакетного представления немного сложнее, поэтому в оставшейся части дня мы сосредоточимся на нем. С помощью фреймворка Nadoop мы создадим пакетные представления, которые позволят нам узнавать количество правок, произведенных автором за некоторый период времени.

Авторы правок в Википедии

В идеале нам хотелось бы получить возможность отправлять запросы вида: «Сколько правок выполнил Фред Блоггс (Fred Bloggs) в период с 15:15 во вторник 5 июня 2012 по 10:45 в четверг 7 июня 2012?» Для этого нам необходимо индексировать записи по точному времени каждой правки. Если бы нам действительно было необходимо выполнять подобные запросы, тогда нам ничего не оставалось бы, как заплатить требуемую цену, но часто нам не требуется определять временные рамки с такой высокой точностью – обычно вполне достаточно границы суток.

В таком случае наше пакетное представление могло бы состоять из количеств правок по дням:

Правки Фреда

2012-02-26 15:04:16
 2012-02-26 16:23:43
 2012-02-26 18:59:03
 2012-02-27 12:56:32
 2012-02-28 17:09:12
 2012-02-28 18:54:28
 2012-03-02 12:00:36
 2012-03-05 10:34:19

Число правок Фреда

2012-02-26: 3
 2012-02-27: 1
 2012-02-28: 2
 2012-03-02: 1
 2012-03-05: 1

Такое решение хорошо подходило бы, если бы нам всегда требовалось получать информацию за период в несколько дней, но для периода в несколько месяцев желательно было бы продолжить объединение значений (например, при запросе числа правок за год). Мы могли бы уменьшить объем работы, чтобы ответить на такой запрос, сохраняя информацию о периодах и в месяцах, и в днях:

Правки Фреда

2012-02-26 15:04:16
 2012-02-26 16:23:43
 2012-02-26 18:59:03
 2012-02-27 12:56:32
 2012-02-28 17:09:12
 2012-02-28 18:54:28
 2012-03-02 12:00:36
 2012-03-05 10:34:19

Число правок Фреда

2012-02-26: 3
 2012-02-27: 1
 2012-02-28: 2
 2012-02: 6
 2012-03-02: 1
 2012-03-05: 1
 2012-03: 2

Это позволило бы уменьшить объем вычислений, необходимых для определения числа правок, выполненных пользователем в течение года, с суммирования 365 значений до 12. И мы могли бы обрабатывать периоды, не совпадающие с границами месяца, суммируя значения по месяцам и вычитая или добавляя значения по дням, как показано на рис. 41.

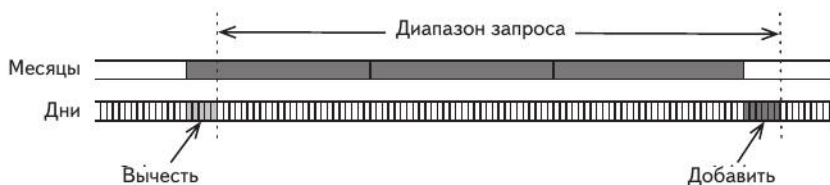


Рис. 41. Обработка периодов, не совпадающих с границами месяца

Журналирование правок

К сожалению, у нас нет доступа к ленте с оперативной информацией об авторах правок в Википедии. Но если бы она была, она могла бы выглядеть, например, так:

```
2012-09-01T14:18:13Z 123456789 1234 Fred Bloggs
2012-09-01T14:18:15Z 123456790 54321 John Doe
2012-09-01T14:18:16Z 123456791 6789 Paul Butcher
.
.
.
```

Первое поле — отметка времени, второе — числовой идентификатор, представляющий правку, третий — числовой идентификатор, представляющий пользователя, выполнившего правку, и все остальное в строке — имя пользователя.

Несмотря на то что Википедия не публикует подобные сведения, она создает периодические дампы в формате XML с полной историей (в файлах с именами `enwiki-latest-stub-meta-history`).¹⁷

В состав пакета примеров для этой главы входит проект `ExtractWikiContributors`, который принимает один из таких дампов и создает файл в формате, описанном выше.

В следующем разделе мы сконструируем задание для `Nadoor`, которое на основе этих файлов журналов будет генерировать данные для нашего пакетного представления.

Подсчет правок

Как всегда, наше задание для `Nadoor` состоит из кода, осуществляющего отображение, и кода, осуществляющего свертку. Отображение реализуется достаточно просто, нужно лишь проанализировать строку из журнала и сгенерировать пары ключ/значение, в которых ключом будет числовой идентификатор пользователя (автора), а значением — время правки:

```
LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/
WikipediaContributors.java18
```

```
public static class Map extends Mapper<Object, Text, IntWritable,
LongWritable> {
```

```
    public void map(Object key, Text value, Context context)
```

¹⁷ <http://dumps.wikimedia.org/enwiki>

¹⁸ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java>


```

        throws IOException, InterruptedException {

    Contribution contribution = new Contribution(value.toString());
    context.write(new IntWritable(contribution.contributorId),
        new LongWritable(contribution.timestamp));
    }
}

```

Большая часть работы выполняется классом `Contribution`:

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/
Contribution.java¹⁹

```

1 class Contribution {
  - static final Pattern pattern = Pattern.compile("^([^\s]*)
  (\\d*) (\\d*) (.*?)$");
  - static final DateTimeFormatter isoFormat = ISODateTimeFormat.
  dateTimeNoMillis();
  -
  5 public long timestamp;
  - public int id;
  - public int contributorId;
  - public String username;
  -
  10 public Contribution(String line) {
  - Matcher matcher = pattern.matcher(line);
  - if(matcher.find()) {
  -     timestamp = isoFormat.parseDateTime(matcher.group(1)).getMillis();
  -     id = Integer.parseInt(matcher.group(2));
  15     contributorId = Integer.parseInt(matcher.group(3));
  -     username = matcher.group(4);
  - }
  - }
  - }

```

Мы могли бы организовать парсинг файла журнала разными способами – в данном случае мы использовали регулярное выражение (строка 2). Если обнаруживается совпадение с ним, с помощью класса `ISODateTimeFormat` из библиотеки `Joda-Time` отметка времени преобразуется в целое число миллисекунд, прошедших с 1 января 1970 года (строка 13).²⁰ Далее следуют идентификаторы правки и автора, являющиеся простыми целыми числами, и имя пользователя занимает остаток строки.

Код, реализующий свертку, получился более сложным:

¹⁹ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/Contribution.java>

²⁰ <http://www.joda.org/joda-time/>

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/
WikipediaContributors.java²¹

```

1 public static class Reduce
-     extends Reducer<IntWritable, LongWritable, IntWritable, Text> {
-     static DateTimeFormatter dayFormat = ISODateTimeFormat.yearMonthDay();
-     static DateTimeFormatter monthFormat = ISODateTimeFormat.yearMonth();
5
-     public void reduce(IntWritable key, Iterable<LongWritable> values,
-                       Context context) throws IOException,
InterruptedException {
-     HashMap<DateTime, Integer> days = new HashMap<DateTime, Integer>();
-     HashMap<DateTime, Integer> months = new HashMap<DateTime, Integer>();
10    for (LongWritable value: values) {
-        DateTime timestamp = new DateTime(value.get());
-        DateTime day = timestamp.withTimeAtStartOfDay();
-        DateTime month = day.withDayOfMonth(1);
-        incrementCount(days, day);
15    incrementCount(months, month);
-    }
-    for (Entry<DateTime, Integer> entry: days.entrySet())
-        context.write(key, formatEntry(entry, dayFormat));
-    for (Entry<DateTime, Integer> entry: months.entrySet())
20    context.write(key, formatEntry(entry, monthFormat));
-    }
- }

```

Для каждого автора создаются два объекта `HashMap`, `days` (строка 8) и `months` (строка 9). Мы заполняем их, выполняя итерации по отметкам времени (не забывайте, что значениями будут списки отметок времени) и используя методы из библиотеки `Joda-Time` `withTimeAtStartOfDay()` и `withDayOfMonth()` для преобразования этих отметок времени в начало суток и в первый день месяца (строки 12 и 13 соответственно). Затем мы увеличиваем соответствующие счетчики в днях и месяцах с помощью простого вспомогательного метода:

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/
WikipediaContributors.java²²

```

private void incrementCount(HashMap<DateTime, Integer> counts, DateTime
key) {
    Integer currentCount = counts.get(key);
    if (currentCount == null)

```

²¹ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java>

²² <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java>

```

        counts.put(key, 1);
    else
        counts.put(key, currentCount + 1);
}

```

Наконец, после завершения конструирования отображений по каждому из них выполняются итерации и генерируется вывод для каждого дня и месяца, в которых имеется хотя бы одна правка (строки с 17 по 20).

Реализация получилась немного сложной, потому что, как мы видели ранее, результатом работы задания для Hadoop всегда является множество пар ключ/значение, а нам нужно вернуть три значения – идентификатор автора правки, дату (месяц или день) и счетчик. Мы могли бы добиться желаемого, определив составное значение. В таком случае роль ключа выполнял бы числовой идентификатор автора, а роль значения – составное значение с датой и счетчиком. Но наш случай достаточно прост, чтобы мы могли в качестве значения использовать строку, сформированную с помощью `formatEntry()`:

```

LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/
WikipediaContributors.java23

```

```

private Text formatEntry(Entry<DateTime, Integer> entry,
                        DateTimeFormatter formatter) {
    return new Text(formatter.print(entry.getKey()) + "\t" + entry.
getValue())
}

```

Ниже приводится фрагмент вывода этого задания:

```

463 2001-11-24 1
463 2002-02-14 1
463 2001-11-26 6
463 2001-10-01 1
463 2002-02 1
463 2001-10 1
463 2001-11 7

```

Он содержит все необходимые нам данные, но коллекция текстовых файлов не особенно удобна в обработке. В следующем разделе мы поговорим об уровне обслуживания (*serving layer*), где происходят индексирование и объединение вывода пакетного уровня.

²³ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsBatch/src/main/java/com/paulbutcher/WikipediaContributors.java>



Вопрос Джо: можно ли генерировать пакетные представления постепенно?

Пакетный уровень, описывавшийся до сих пор, повторно вычисляет все пакетные представления с самого начала при каждом запуске. Этот прием, конечно, работает, но очевидно, что при таком подходе будет выполняться много лишней работы – почему не обновлять пакетные представления постепенно, по мере появления новых данных с момента последнего обновления представления?

Ответ на этот вопрос прост: нет ничего, что мешало бы поступать таким образом, и это было бы хорошей оптимизацией. Но мы не можем полагаться только на такие пошаговые обновления – значительная доля мощности лямбда-архитектуры происходит из того факта, что всегда есть возможность перестроить представление с самого начала, если в этом возникнет необходимость. Поэтому вы вольны реализовать свой алгоритм постепенного конструирования представления, если такая оптимизация стоит затраченных усилий, но учитывайте, что он никогда не сможет служить заменой полной перестройке представления с нуля.

Завершение картины

Одного только пакетного уровня недостаточно для создания законченного приложения. Требуется еще один элемент лямбда-архитектуры – сервисный уровень.

Сервисный уровень

Вновь созданное пакетное представление необходимо индексировать, чтобы потом эффективнее обрабатывать запросы, и нам нужно поместить куда-то прикладную логику, которая будет решать, как объединять элементы пакетного представления, чтобы удовлетворить каждый конкретный запрос. Эти задачи решает *сервисный уровень* (serving layer), как показано на рис. 42.

Я оставлю реализацию сервисного уровня вам, уважаемый читатель, как задание для самостоятельного решения, поскольку он слабо связан с темой данной книги, но в нем есть один немаловажный аспект – база данных, – на котором хотелось бы остановиться отдельно.

Несмотря на то что сервисный уровень строится на основе традиционной базы данных, шаблоны доступа к ней отличаются от шабло-

нов, используемых в традиционных приложениях. В частности, отсутствует требование к возможности произвольного доступа – база данных обновляется, только когда обновляются пакетные представления из-за необходимости обновления пакетов.

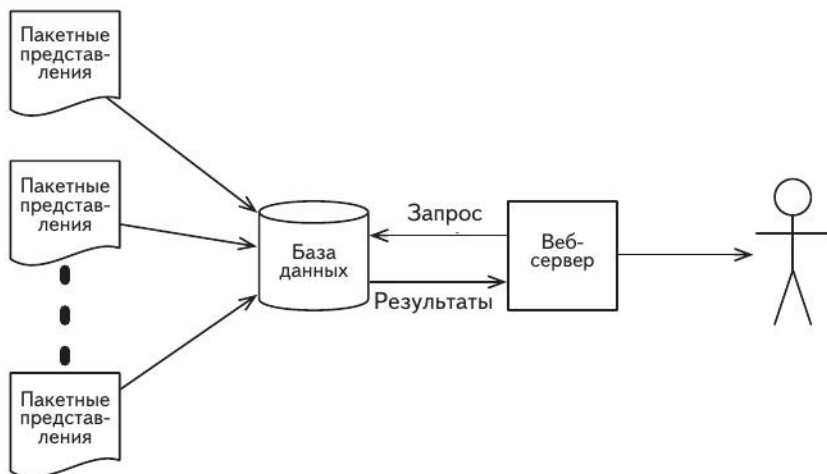


Рис. 42. Сервисный уровень

По этой причине появилась категория баз данных для данного шаблона использования, наиболее известными из которых являются ElephantDB²⁴ и Voldemort²⁵.

Почти нирвана

Вместе пакетный и сервисный уровни образуют систему данных, которая решает все проблемы, выявленные нами в начале сегодняшнего дня, как показано на рис. 43.

Реализация пакетного уровня выполняет бесконечный цикл, регенерируя пакетные представления из исходных данных. Каждый раз, когда пакетный уровень завершает цикл, сервисный уровень обновляет содержимое базы данных.

Так как исходные данные остаются неизменными, реализацию пакетного уровня легко можно распараллелить. Исходные данные могут быть распределены между компьютерами в кластере, позволяя пересчитывать пакетные представления за приемлемый промежуток времени, даже если объем исходных данных исчисляется терабайтами.

²⁴ <https://github.com/nathanmarz/elephantdb>

²⁵ <http://www.project-voldemort.com/voldemort/>



Рис. 43. Законченная система данных

Неизменяемость исходных данных также означает устойчивость системы к техническим отказам и человеческим ошибкам. Мало того, что для исходных данных проще создавать резервные копии, если будет обнаружена ошибка, самое худшее, что может произойти, – пакетное представление временно станет ошибочным, что легко исправить, снова запустив вычисления.

Наконец, благодаря сохранению доступа к исходным данным мы всегда можем сгенерировать любой отчет или провести любой анализ, который может понадобиться в будущем.

Однако остается одна существенная проблема – задержка. Если для выполнения расчетов пакетному уровню требуется время, измеряемое часами, наши пакетные представления всегда будут получаться устаревшими, как минимум на этот промежуток времени. Но эта тема – уровень ускорения – будет обсуждаться завтра.

В завершение второго дня

На этом мы заканчиваем второй день исследования лямбда-архитектуры. Завтра мы дополним картину, добавив в нее уровень ускорения.

Что мы узнали во второй день

Информацию часто можно разделить на исходные и производные данные. Исходные данные являются истиной в последней инстанции

и потому неизменны. Пакетный уровень в лямбда-архитектуре использует эту их особенность и позволяет создавать системы, которые:

- широко используют приемы параллельной обработки, давая возможность обрабатывать терабайты данных;
- просты в разработке и менее чреваты ошибками;
- устойчивы к аппаратным отказам и человеческим ошибкам и способны поддерживать повседневные операции и извлекать историческую информацию для последующего составления отчетов и анализа.

Главным недостатком пакетного уровня является задержка, которую в лямбда-архитектуре можно компенсировать добавлением уровня ускорения.

Упражнения ко второму дню

Найдите:

- подходы к построению систем данных на основе фреймворка Hadoop и лямбда-архитектуры, обсуждавшиеся здесь, не являются единственно возможными – из альтернатив можно назвать фреймворки HBase, Pig и Hive. Все три имеют намного больше общего с традиционными системами данных. Выберите один из них и сравните с пакетным уровнем лямбда-архитектуры. В каких ситуациях они могут оказаться более предпочтительным выбором?

Задания:

- закончите систему, созданную сегодня, добавив в нее сервисный уровень, принимающий результаты работы пакетного уровня, помещающий их в базу данных и позволяющий запрашивать число правок, выполненных определенным пользователем за определенный период времени. Вы можете сконструировать сервисный уровень на основе традиционной базы данных или взять за основу базу данных ElephantDB;
- добавьте в пакетный уровень возможность постепенного обновления – для этого вам потребуется создать кластер Hadoop с доступом к базе данных сервисного уровня. Насколько эффективнее работает такая реализация? Стоит ли увеличение эффективности затраченных усилий? В каких приложениях имеет смысл использовать постепенное вычисление пакетных представлений, а в каких – нет?

День 3: уровень ускорения

Как мы видели вчера, пакетный уровень лямбда-архитектуры решает все проблемы, свойственные традиционным системам данных, но это решение дается ценой задержки. На практике эта проблема обычно решается за счет добавления *уровня ускорения* (speed layer). На рис. 44 показано, как взаимодействуют пакетный уровень и уровень ускорения.

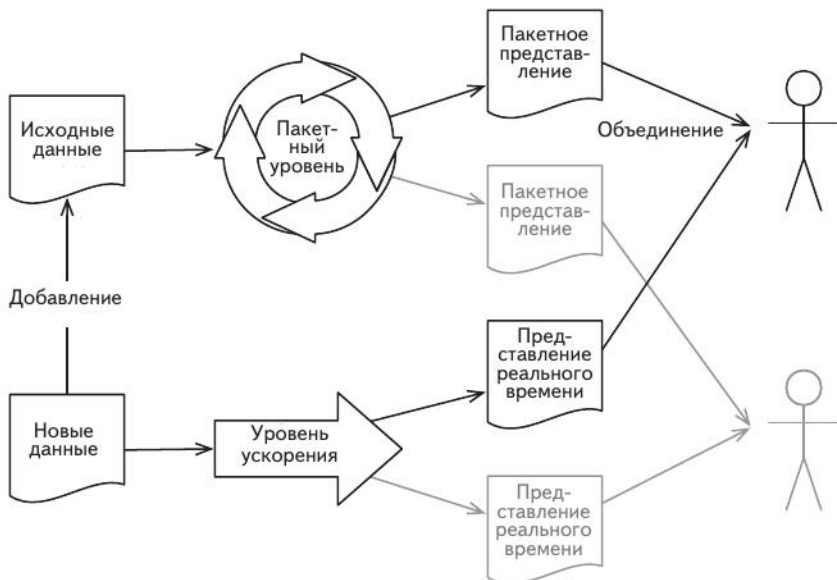


Рис. 44. Лямбда-архитектура

При получении новых данных необходимо добавить их в исходные данные, которые обрабатываются пакетным уровнем, и послать их уровню ускорения. Уровень ускорения предназначен для создания представлений реального времени, которые объединяются с пакетными представлениями с целью вернуть самую свежую информацию в ответ на запросы.

Представления реального времени содержат только производную информацию, полученную на основе исходных данных, прибывших после создания последних пакетных представлений, и уничтожаемую после обработки исходных данных пакетным уровнем.

Сегодня мы посмотрим, как создать уровень ускорения с помощью Storm.²⁶

²⁶ <http://storm.incubator.apache.org>

Архитектура уровня ускорения

Понятие «реального времени» в разных приложениях интерпретируется по-разному – некоторые требуют, чтобы новые данные были доступны в течение нескольких секунд, другие – в течение миллисекунд. Но независимо от интерпретации весьма маловероятно, что подход исключительно на основе пакетных представлений будет удовлетворять даже самым мягким требованиям.

Соответственно, уровень ускорения будет иметь более сложную конструкцию, чем пакетный уровень, потому что он вынужден реализовывать обработку данных порциями, по мере их поступления, то есть использовать инкрементальный подход (incremental approach). Это, в свою очередь, означает, что он не может ограничиваться только обработкой исходных данных и не может полагаться на свойства исходных данных, обсуждавшиеся вчера. Поэтому мы вернемся к традиционным базам данных, поддерживающим произвольную запись, и ко всем связанным с этим сложностям (блокировки, транзакции и т. д.).

Но есть и хорошая новость: уровень ускорения должен обрабатывать только те порции данных, которые еще не были обработаны пакетным уровнем (обычно полученные в течение последних нескольких часов). Как только пакетный уровень догонит уровень ускорения, старые данные можно будет удалить из уровня ускорения.

Синхронно или асинхронно?

Один из очевидных подходов к реализации уровня ускорения заключается в том, чтобы реализовать его в виде традиционного синхронного приложения базы данных. Фактически традиционное приложение базы данных можно считать вырожденной формой лямбда-архитектуры, в которой пакетный уровень никогда не выполняется (см. рис. 45).

При таком подходе клиенты взаимодействуют непосредственно с базой данных и блокируются при обработке каждого обновления. Это вполне разумное решение, и для некоторых приложений оно может оказаться единственно возможным, отвечающим предъявляемым требованиям. Но чаще асинхронная архитектура выглядит предпочтительнее (см. рис. 46).

В данном случае клиенты добавляют обновления в очередь (реализованную, например, с применением Kafka²⁷ или Kestrel²⁸) без при-

²⁷ <http://kafka.apache.org>

²⁸ <http://robey.github.io/kestrel/>

менения блокировок. Поточковый процессор затем обрабатывает эти обновления и обновляет информацию в базе данных.

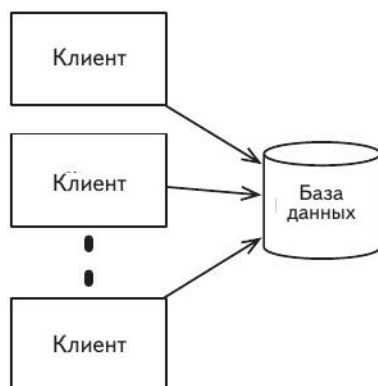


Рис. 45. Традиционное приложение базы данных

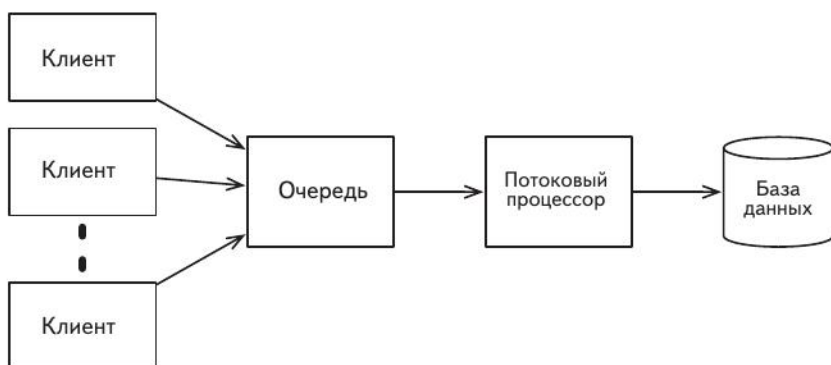


Рис. 46. Асинхронная организация приложений баз данных

Применение очереди разрывает зависимость клиентов от базы данных, но усложняет координацию записи изменений с другими операциями. Для приложений, где это приемлемо, подобная организация дает существенные выгоды:

- так как клиенты не блокируются, большие объемы данных могут обрабатываться меньшим числом клиентов, что дает более высокую пропускную способность;
- всплески нагрузки могут приводить синхронные системы к временным зависаниям или потере обновлений из-за перегрузки клиентов или базы данных. Асинхронные системы, на-

- против, будут просто запаздывать, сохраняя обновления в очереди и нагоняя при снижении нагрузки до обычного уровня;
- как мы увидим далее, потоковый процессор может использовать приемы параллельной обработки данных, распределяя вычисления между несколькими компьютерами, чтобы обеспечить одновременно и устойчивость к отказам, и увеличение производительности.

По этим причинам, а также потому, что синхронная реализация уровня ускорения не представляет интереса с точки зрения параллелизма и конкуренции, мы не будем рассматривать ее далее в этой книге. Но, прежде чем перейти к асинхронной реализации, нам следует коснуться еще одной темы – определение момента, когда данные могут считаться устаревшими.

Устаревание данных

Если пакетному уровню требуется (допустим) два часа на обработку исходных данных, вы могли бы подумать, что уровень ускорения должен обрабатывать данные, поступившие за два последних часа. В действительности это не так, мы должны увеличить этот интервал вдвое, как показано на рис. 47.

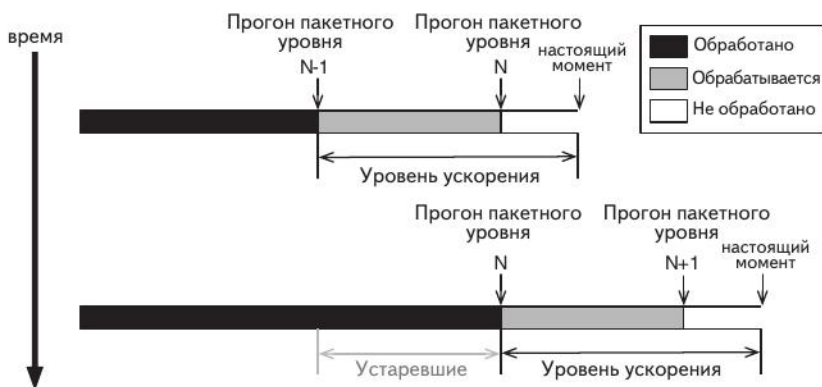


Рис. 47. Устаревание данных на уровне ускорения

Представьте, что прогон ($N - 1$) пакетного уровня только что завершился и начался прогон N . Если каждый прогон занимает два часа, это означает, что готовые пакетные представления будут иметь двухчасовой возраст. Соответственно, уровень ускорения должен обслуживать запросы на получение данных, поступивших за те два часа,

плюс данные, которые поступят до завершения прогона N , то есть получается интервал в четыре часа.

Когда прогон N пакетного уровня завершится, данные, полученные за предыдущие два часа, должны быть объявлены устаревшими, но данные, поступившие за последние два часа, все еще должны обслуживаться уровнем ускорения. Безусловно, можно придумать разные схемы определения устаревших данных, но самый простой путь – параллельно выполнять две копии уровня ускорения и переключаться между ними, как показано на рис. 48.

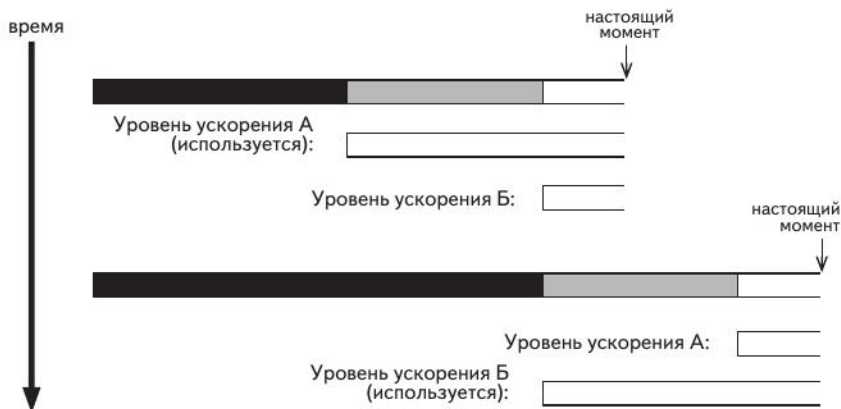


Рис. 48. Переключение между уровнями ускорения

Всякий раз, когда пакетный уровень завершает прогон и новые данные становятся доступны через пакетные представления, мы переключаемся с текущего уровня ускорения на другой, хранящий более свежие данные. После этого прежде использовавшийся уровень ускорения очищает свою базу данных и приступает к формированию нового набора данных с нуля, начиная с момента запуска нового прогона пакетного уровня.

Такой подход не только избавляет нас от необходимости идентифицировать данные, подлежащие удалению из базы данных уровня ускорения, но также повышает производительность и читаемость кода, гарантируя, что каждая новая итерация уровня ускорения будет начинаться с совершенно чистой базы данных. Безусловно, за это придется заплатить обслуживанием двух копий данных на двух уровнях ускорения и удвоенным потреблением вычислительных ресурсов, но эта цена едва ли будет слишком велика, учитывая, что уровень ускорения обслуживает лишь малую часть всех данных.

Storm

Оставшуюся часть сегодняшнего дня мы посвятим знакомству с асинхронной реализацией уровня ускорения на основе системы Storm. Применение системы Storm – слишком обширная тема, поэтому мы затронем лишь те ее аспекты, которые нам потребуются. За дополнительной информацией обращайтесь к документации по этой системе.²⁹

Назначение системы Storm – обеспечить обработку данных в режиме реального времени, которую фреймворк Hadoop выполняет в пакетном режиме, и упростить распределение вычислений по нескольким компьютерам, чтобы повысить производительность и устойчивость к отказам.

Воронки, сита и топологии

Система Storm обрабатывает потоки кортежей. Кортежи в Storm похожи на кортежи, которые мы видели в главе 5 «Акторы». Основное отличие кортежей в системе Storm от кортежей в языке Elixir заключается в том, что кортежи в Storm являются именованными.

Кортежи создаются *воронками* (spouts) и обрабатываются *ситами* (bolts), которые сами могут создавать новые кортежи. Воронки и сита связаны *потоками данных* (streams), образующими *топологию* (topology). На рис. 49 показан пример простой топологии, образованной единственной воронкой, создающей кортежи, и конвейером сит, обрабатывающих их.

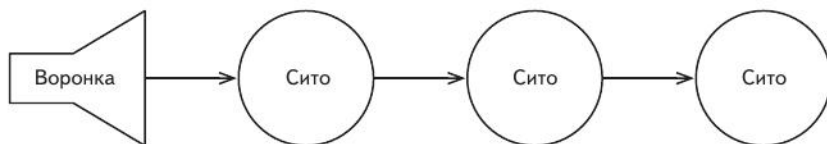


Рис. 49. Простая топология

Топологии могут иметь намного более сложный вид – сита могут принимать несколько потоков данных, и один поток данных может проходить множество сит, образуя ориентированный ациклический граф (Directed Acyclic Graph, DAG), как показано на рис. 50.

Однако даже наша простая топология в виде конвейера не так проста, как кажется, потому что воронки и сита действуют параллельно и на разных компьютерах.

²⁹ <http://storm.incubator.apache.org/documentation/Home.html>

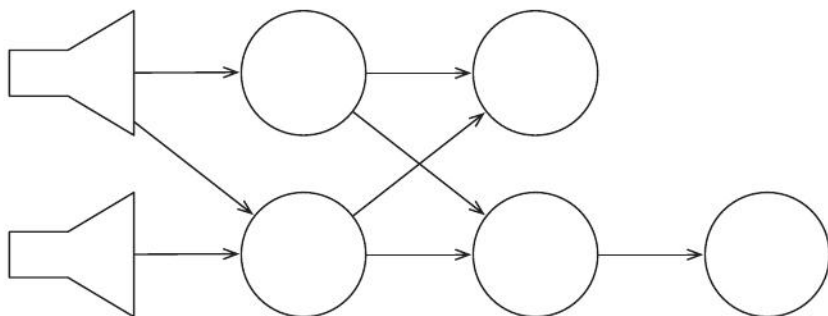


Рис. 50. Сложная топология

Рабочие процессы

В нашей реализации не только воронки и сита выполняются параллельно, но и сами они имеют внутреннюю параллельную организацию – каждый компонент (воронка или сито) реализован как множество рабочих процессов. На рис. 51 показано, как могла бы выглядеть наша простая топология, если бы каждая воронка и сито имели по три рабочих процесса.

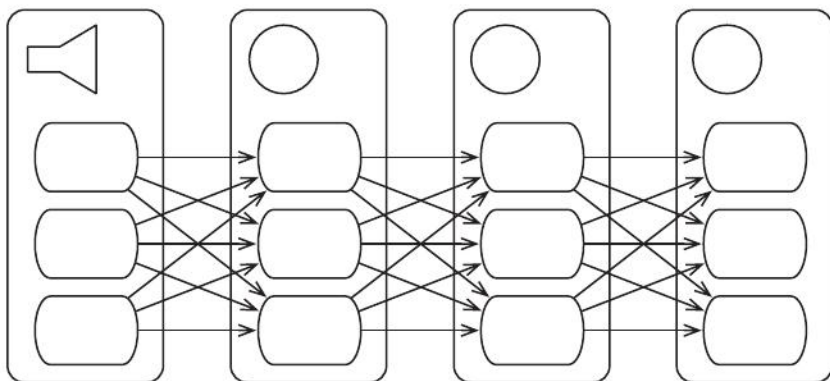


Рис. 51. Рабочие процессы воронки и сит

Как видно на диаграмме, рабочие процессы каждого узла конвейера могут отправлять кортежи любым рабочим процессам следующего узла в конвейере. Как управлять, какой рабочий процесс должен принять кортеж, мы увидим далее, когда будем обсуждать группировку потоков данных.

Наконец, рабочие процессы могут действовать. При выполнении на кластере из четырех узлов, например, рабочие процессы воронки могут действовать на узлах 1, 2 и 3; рабочие процессы первого сита могут выполняться на узлах 2 и 4 (два на узле 2 и один на узле 4) и т. д.

Вся прелесть системы Storm в том, что она избавляет нас от необходимости заботиться о распределении вычислений. Все, что мы должны сделать, – определить топологию, а среда выполнения Storm создаст рабочие процессы на узлах и обеспечит соответствующую маршрутизацию потоков данных

Устойчивость к отказам

Важнейшей причиной распределения рабочих процессов единственной воронки или сита по нескольким компьютерам является обеспечение устойчивости к отказам. Если один из компьютеров в кластере выйдет из строя, наша топология сможет продолжить работу, перераспределив потоки данных между компьютерами, остающимися в строю.

Система Storm следит за зависимостями между кортежами – если обработка конкретного кортежа не закончена, Storm повторит создание кортежа в воронке, от которого зависит необработанный кортеж. Это означает, что по умолчанию система Storm гарантирует «как минимум однократную» обработку. Приложения должны создаваться с учетом того факта, что кортежи могут повторяться, и продолжать действовать в штатном режиме, если это произойдет.

Но хватит теории – давайте посмотрим, как с помощью Storm создать уровень ускорения для нашего приложения, позволяющего получать сведения об авторах статей в Википедии.



Вопрос Джо: а что, если мое приложение не сможет обрабатывать повторы?

Семантика системы Storm «как минимум однократной» обработки без труда может быть поддержана в большинстве приложений, но иногда все же требуется гарантировать более строгую семантику «точно одной» обработки.

Система Storm поддерживает такую семантику посредством Trident API.* Однако мы не будем рассматривать подсистему Trident в этой книге.

* <http://storm.incubator.apache.org/documentation/Trident-tutorial.html>

Подсчет правок с помощью Storm

На рис. 52 изображена возможная топология для нашего уровня ускорения.



Рис. 52. Возможная топология уровня ускорения для программы подсчета правок

Начнем с воронки, импортирующей журналы с информацией о правках и преобразующей их в поток кортежей. Этот поток данных будет передаваться ситу, осуществляющему парсинг записей из журнала и генерирующему поток результатов парсинга. Наконец, этот поток данных будет передаваться ситу, обновляющему базу данных, содержащую представление реального времени.

Фактически мы реализуем немного отличную топологию по двум причинам. Во-первых, у нас нет доступа к журналу правок в Википедии; и во-вторых – подробности обновления базы данных нам не интересны (основной интерес для нас представляют конкуренция и параллелизм). На рис. 53 показана топология, которая будет реализована в действительности.



Рис. 53. Фактическая топология уровня ускорения для программы подсчета правок

Сначала мы создадим воронку, генерирующую имитации записей в журналах Википедии, затем сито парсера и, наконец, сито, обновляющее представление реального времени в памяти. Это не самое лучшее решение для промышленного использования, но оно прекрасно служит целям исследования возможностей системы Storm.

Имитации записей в журналах

Ниже приводится реализация воронки, имитирующей журнал правок и генерирующей случайные записи в журнале:

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/
RandomContributorSpout.java³⁰

```

1 public class RandomContributorSpout extends BaseRichSpout {
-
-     private static final Random rand = new Random();
-     private static final DateTimeFormatter isoFormat =
5         ISODateTimeFormat.dateTimeNoMillis();
-
-     private SpoutOutputCollector collector;
-     private int contributionId = 10000;
-
10    public void open(Map conf, TopologyContext context,
-        SpoutOutputCollector collector) {
-
-        this.collector = collector;
-    }
15
-    public void declareOutputFields(OutputFieldsDeclarer declarer) {
-        declarer.declare(new Fields("line"));
-    }
-
20    public void nextTuple() {
-        Utils.sleep(rand.nextInt(100));
-        ++contributionId;
-        String line = isoFormat.print(DateTime.now()) + " " + contributionId
+ " " +
-        rand.nextInt(10000) + " " + "dummyusername";
25    collector.emit(new Values(line));
-    }
- }

```

Обратите внимание, что класс нашей воронки наследует класс `BaseRichSpout` (строка 1). Система Storm вызовет метод `open()` (строка 10) в ходе инициализации – в нем мы просто сохраняем запись типа `SpoutOutputCollector`, куда будет направляться весь вывод. В ходе инициализации система Storm вызовет также наш метод `declareOutputFields()` (строка 16), чтобы определить, как должны структурироваться кортежи, генерируемые воронкой, – в данном случае кортежи состоят из единственного поля с именем `line`.

Основную работу выполняет метод `nextTuple()` (строка 20). Он сначала приостанавливается на случайный период времени от 0 до 100 миллисекунд, затем создает строку, имеющую формат, который

³⁰ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/RandomContributorSpout.java>

мы уже видели в разделе «Журналирование правок» выше, и выводит ее вызовом `collector.emit()`.

Эти строки передаются для обработки в сито, осуществляющее парсинг.

Парсинг записей в журнале

Наше сито, осуществляющее парсинг, принимает кортежи с записями из журнала, анализирует их и выводит кортежи с четырьмя полями, по одному на каждый компонент записи:

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/ContributionParser.java³¹

```
1 class ContributionParser extends BaseBasicBolt {
2     public void declareOutputFields(OutputFieldsDeclarer declarer) {
3         declarer.declare(new Fields("timestamp", "id", "contributorId",
"username"));
4     }
5     public void execute(Tuple tuple, BasicOutputCollector collector) {
6         Contribution contribution = new Contribution(tuple.getString(0));
7         collector.emit(new Values(contribution.timestamp, contribution.id,
8             contribution.contributorId, contribution.username));
9     }
10 }
```

Обратите внимание, что класс нашего сита наследует `BaseBasicBolt` (строка 1). Как и в случае с воронкой, мы реализуем метод `declareOutputFields()` (строка 2), чтобы сообщить системе Storm структуру выходных кортежей – в данном случае они состоят из четырех полей с именами `timestamp`, `id`, `contributorId` и `username`.

Основную работу выполняет метод `execute()` (строка 5). С помощью той же структуры `Contribution`, что используется в пакетном уровне, исходная запись из журнала делится на составляющие, и полученный кортеж посылается дальше вызовом `collector.emit()`.

Эти кортежи передаются далее в сито, сохраняющее информацию об авторах правок.

Сохранение информации о правках

Наше последнее сито реализует простенькую базу данных в памяти (отображающую идентификатор автора во множество отметок

³¹ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/ContributionParser.java>

времени), где хранится информация о правках в Википедии и их авторах:

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/ContributionRecord.java³²

```
1 class ContributionRecord extends BaseBasicBolt {
-   private static final HashMap<Integer, HashSet<Long>> timestamps =
-       new HashMap<Integer, HashSet<Long>>();
-
5   public void declareOutputFields(OutputFieldsDeclarer declarer) {
-   }
-   public void execute(Tuple tuple, BasicOutputCollector collector) {
-       addTimestamp(tuple.getInteger(2), tuple.getLong(0));
-   }
10
-   private void addTimestamp(int contributorId, long timestamp) {
-       HashSet<Long> contributorTimestamps = timestamps.get(contributorId);
-       if (contributorTimestamps == null) {
-           contributorTimestamps = new HashSet<Long>();
15       timestamps.put(contributorId, contributorTimestamps);
-       }
-       contributorTimestamps.add(timestamp);
-   }
- }
```

В данном случае сито не генерирует никакого вывода, поэтому метод `declareOutputFields()` пуст (строка 5). Наш метод `execute()` (строка 7) просто извлекает соответствующие поля из принятого кортежа и передает их методу `addTimestamp()`, который просто добавляет отметку времени во множество, ассоциированное с конкретным автором.

В заключение мы посмотрим, как объявить топологию, которая будет использовать наши воронку и сита.

Объявление топологии

Кого-то может взволновать следующая проблема: учитывая, что каждое сито имеет множество рабочих процессов, как гарантировать, что для каждого автора будет существовать только одно множество отметок времени? Мы увидим это, когда закончим объявление топологии.

³² <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/ContributionRecord.java>

**Вопрос Джо: почему сохраняется множество отметок времени?**

Пакетные представления, которые мы рассматривали вчера, сохраняют лишь ежедневные и ежемесячные счетчики для каждого автора. Почему же в представлениях реального времени сохраняются списки отметок времени?

Во-первых, как уже говорилось выше, наши представления реального времени необходимы для хранения информации всего лишь за последние несколько часов, поэтому цена хранения списков отметок времени относительно невысока. Но имеется еще одна, более важная причина – добавление элемента во множество является идемпотентной (idempotent) операцией.

Напомню, что по умолчанию система Storm поддерживает семантику «как минимум однократной» обработки (см. раздел «Устойчивость к отказам» выше в этой же главе), поэтому кортежи могут повторяться. Идемпотентная операция возвращает один и тот же результат, независимо от того, сколько раз она будет выполнена, то есть именно то, что нам нужно, чтобы справиться с повторяющимися кортежами.

LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/WikiContributorsTopology.java³³

```
1 public class WikiContributorsTopology {  
-  
-   public static void main(String[] args) throws Exception {  
-  
5       TopologyBuilder builder = new TopologyBuilder();  
-  
-       builder.setSpout("contribution_spout", new RandomContributorSpout(), 4);  
-  
-       builder.setBolt("contribution_parser", new ContributionParser(), 4).  
10         shuffleGrouping("contribution_spout");  
-  
-       builder.setBolt("contribution_recorder", new ContributionRecord(), 4).  
-         fieldsGrouping("contribution_parser", new Fields("contributorId"));  
-  
15       LocalCluster cluster = new LocalCluster();  
-       Config conf = new Config();  
-       cluster.submitTopology("wiki-contributors", conf, builder.  
createTopology());
```

³³ <http://media.pragprog.com/titles/pb7con/code/LambdaArchitecture/WikiContributorsSpeed/src/main/java/com/paulbutcher/WikiContributorsTopology.java>


```
-  
-      Thread.sleep(10000);  
20  
-      cluster.shutdown();  
-    }  
- }
```

Определение топологии начинается с создания экземпляра `TopologyBuilder` (строка 5), в который затем добавляется экземпляр воронки, вызовом метода `setSpout()` (строка 7), во втором аргументе которому передается желаемое число рабочих процессов. Слово «желаемое» здесь употреблено не просто так – это число является выражением наших желаний, а не инструкцией к действию, но в данном случае можно сказать, что мы инструктируем систему Storm о необходимости создать четыре процесса. Полное описание управления параллелизмом в системе Storm можно найти в статье «What Makes a Running Topology: Worker Processes, Executors and Tasks».³⁴

Далее добавляется экземпляр сита `ContributionParser` вызовом метода `setBolt()` (строка 9). При этом системе Storm сообщается, что сито должно принимать результаты работы воронки, вызывая метод `shuffleGrouping()`, которому передается имя воронки, что естественным образом подводит нас к теме группировки потоков данных.

Группировка потоков данных

Механизм *группировки потоков данных* в системе Storm отвечает на вопрос о том, какие рабочие процессы какие кортежи должны принимать. *Группировка перемешиванием* (`shuffle grouping`), используемая для сита, осуществляющего парсинг, является самой простой – кортежи передаются случайно выбранному рабочему процессу.

Для сита, осуществляющего запись информации об авторах правок, используется *группировка по значениям полей* (`fields grouping`) (строка 12), гарантирующая, что все кортежи с одинаковыми значениями указанных полей (в данном случае – единственного поля `contributorId`) всегда будут передаваться одному и тому же рабочему процессу. Именно таким способом гарантируется присутствие единственного множества отметок времени для каждого автора.

Локальный кластер

Настройка кластера Storm не является какой-то архисложной задачей, но ее описание далеко выходит за рамки данной книги. И, к

³⁴ <http://storm.incubator.apache.org/documentation/Understanding-the-parallelism-of-a-Storm-topology.html>

сожалению, из-за относительной молодости технологии никто пока не предоставляет услуги кластеров под управлением Storm, которыми мы могли бы воспользоваться. Поэтому мы запускаем эту технологию локально, создав экземпляр `LocalCluster` локального кластера (строка 15).

Наш пример позволяет данной топологии выполняться в течение 10 секунд и затем останавливает ее вызовом `cluster.shutdown()`. В промышленном варианте, разумеется, мы могли бы выполнять остановку топологии, когда генерируемые ею представления реального времени станут не нужны, после того как пакетный уровень нагонит уровень ускорения.

В завершение третьего дня

Итак, мы подошли к концу третьего дня и к концу обсуждения уровня ускорения лямбда-архитектуры.

Что мы узнали в третий день

Уровень ускорения является заключительным компонентом лямбда-архитектуры, который генерирует представления реального времени на основе данных, поступивших с момента сборки последних версий пакетных представлений. Уровень ускорения может иметь синхронную или асинхронную архитектуру – система Storm является одним из средств, с помощью которых можно конструировать уровни ускорения с асинхронной архитектурой:

- система Storm обрабатывает потоки кортежей в режиме реального времени. Кортежи создаются воронками (spouts) и обрабатываются ситами (bolts), выстроенными в топологию;
- каждая воронка и каждое сито выполняются в виде нескольких рабочих процессов, действующих параллельно и разбросанных по нескольким узлам в кластере;
- по умолчанию система Storm реализует семантику «как минимум однократной» обработки – сита должны учитывать возможность повторного появления одних и тех же кортежей.

Упражнения к третьему дню

Найдите:

- подсистема Trident – это высокоуровневый API, базирующийся на Storm API, которая, кроме всего прочего, реализует се-

мантику «точно однократной» обработки, так же как и семантику «как минимум однократной» обработки. В каких случаях имеет смысл использовать Trident API, а в каких – низкоуровневый Storm API?

- какие другие варианты группировки потоков данных поддерживаются системой Storm, помимо группировки перемешиванием и группировки по значениям полей?

Задания:

- создайте кластер Storm и запустите на нем сегодняшний пример, чтобы он выполнялся распределенно, а не локально;
- создайте сито, сохраняющее общее число правок, которые выполнялись каждую минуту, и определите топологию, в которой оба сита – это и `ContributionRecord` – получают вывод от `ContributionParser`;
- в сегодняшнем примере мы использовали базовый класс `BaseBasicBolt`, который автоматически подтверждает прием кортежей. Возьмите за основу другой базовый класс, `BaseRichBolt`, – в этом случае вам придется явно подтверждать получение каждого кортежа. Как бы вы реализовали сито, сначала обрабатывающее несколько кортежей и только потом подтверждающее их прием?

В завершение

Лямбда-архитектура объединяет множество концепций, с которыми мы познакомились в предыдущих главах:

- утверждение о бесспорной истинности исходных данных должно напомнить вам отделение идентичности от состояния в языке Clojure;
- подход к распараллеливанию задач путем разбиения их на фазу отображения и фазу свертки, реализованный в фреймворке `Nadoop`, должен напомнить вам приемы распараллеливания в функциональном программировании;
- подобно актерам, лямбда-архитектура распределяет обработку среди множества компьютеров в кластере, давая увеличение производительности и устойчивости к отказам;
- потоки кортежей в системе Storm имеют много общего с передачей сообщений между актерами и в модели CSP.

Сильные стороны

Лямбда-архитектура разрабатывалась с целью дать возможность обрабатывать гигантские объемы данных, то есть решать задачи, с которыми с большим трудом справляются традиционные архитектуры данных. Она особенно хорошо подходит для составления отчетов и анализа – разновидностей задач, которые в прошлом решались с применением хранилищ данных.

Слабые стороны

Лямбда-архитектура обладает большой мощностью – она способна обрабатывать гигантские объемы данных – но ее сильные стороны являются ее слабостью. Если у вас объемы данных не исчисляются хотя бы десятками гигабайт, накладные расходы (интеллектуальные и вычислительные) окажутся слишком велики.

Альтернативы

Лямбда-архитектура не имеет тесной связи с технологией MapReduce – пакетный уровень может быть реализован на основе любой распределенной системы пакетной обработки.

С этой точки зрения особого внимания заслуживает Apache Spark.³⁵ Spark – это фреймворк для создания вычислительных кластеров, реализующий механизм выполнения DAG, позволяющий более естественно выражать различные алгоритмы (в первую очередь алгоритмы на основе графов), чем это дает технология MapReduce. Он также имеет потоковый API, что дает возможность реализовать пакетный уровень и уровень ускорения.³⁶

Напоследок

Обзор лямбда-архитектуры является отличным завершением данной книги, так как в ней используются многие приемы, обсуждавшиеся в предыдущих главах. Он наглядно демонстрирует, как параллелизм и конкуренция позволяют решать задачи, неразрешимые при использовании других подходов.

В последней главе мы еще раз пройдемся по всему, что изучили за последние семь недель, и коснемся некоторых новых тем.

³⁵ <http://spark.apache.org>

³⁶ <http://spark.apache.org/streaming/>



ГЛАВА 9.

В заключение

Поздравляю с завершением семинедельного курса!

Мы охватили массу подробностей, от параллелизма с тонким разделением, поддерживаемым графическими процессорами, до масштабируемых кластеров и применения технологии MapReduce. Попутно мы увидели, как приемы конкурентного и параллельного программирования позволяют использовать мощности современных многоядерных процессоров, а также познакомились со многими другими преимуществами, которые дает выход за рамки привычного последовательного программирования:

- мы видели, как язык Elixir, фреймворк Hadoop и система Storm обеспечивают распределенные вычисления в кластерах независимых компьютеров, позволяя нам создавать решения, устойчивые к отказам техники;
- во время знакомства с библиотекой `core.async` мы видели, как конкуренция спасает нас от «ада функций обратного вызова», обычно связанного с обработкой событий;
- в главе, рассказывающей о функциональном программировании, мы видели, что конкурентное решение может быть проще и легче для понимания, чем последовательный эквивалент.

А теперь посмотрим, что это означает для нашего с вами будущего.

Куда мы идем?

Более двадцати лет тому назад я предсказывал, что со временем параллельное и распределенное программирование превратится в господствующую тенденцию, поэтому у меня нет потрясающего послужного списка, как у ученого. Тем не менее я полагаю, что знание приемов конкурентного и параллельного программирования безусловно найдет применение в будущем.

Будущее за неизменяемостью

Как мне кажется, из уроков прошлых лет один выглядит ярче других – в будущем неизменяемость будет играть еще более преобладающую роль.

Неизменяемость наилучшим образом подходит для функционального программирования – именно отказ от изменяемого состояния делает реализацию параллельных и конкурентных программ такой простой на функциональных языках. Но чтобы воспользоваться преимуществами неизменяемости, совсем необязательно писать функциональные программы. Давайте рассмотрим еще раз факты, рассматривавшиеся нами последние несколько недель:

- хотя Clojure не является чисто функциональным языком программирования, его базовые структуры данных являются неизменяемыми и, соответственно, сохраняемыми (как было показано в разделе «Сохраняемые структуры данных» в главе 4). А сохранность структур данных не мешает языку Clojure поддерживать изменяемые ссылки, отделяющие идентичность от состояния и помогающие избежать проблем, обычно свойственных изменяемому состоянию;
- хотя обычно на самом низком уровне лямбда-архитектуры не используется функциональный код, в основе ее все-таки лежит неизменяемость – ограничивая пакетный уровень всегда истинными (неизменяемыми) исходными данными, мы получаем простую возможность распределять эти данные между компьютерами в кластере, обрабатывать их параллельно и успешно противостоять отказам техники и человеческим ошибкам;
- хотя Elixir не является чисто функциональным языком программирования, отсутствие изменяемых переменных в нем является ключом к потрясающей эффективности и надежности виртуальной машины Erlang, на которой он выполняется;
- сообщения, которыми обмениваются акторы и приложения, реализованные на основе модели CSP, являются неизменяемыми;
- неизменяемость особенно выгодна при создании многопоточных программ, использующих блокировки, – чем больше неизменяемых данных используется в программе, тем реже требуется применять блокировки и тем меньше приходится беспокоиться о проблеме видимости памяти.

На мой взгляд, совершенно очевидно, что даже если вы не пользуетесь функциональными языками, фреймворки, которые вы используете, и код, который вы пишете, все более будут оказываться под влиянием принципов функционального программирования. Это не только упростит для вас реализацию конкурентных и параллельных программ, но также сделает ваш код более простым и более надежным.

Будущее за распределенными вычислениями

Текущий всплеск интереса к параллелизму и конкуренции заключается в кризисе многоядерных систем. Вместо ускорения работы отдельных ядер мы наблюдаем появление процессоров со все большим и большим числом ядер. Положительной стороной такого направления развития является возможность использовать все эти ядра с применением приемов, которые мы изучали последние несколько недель.

Но на нас надвигается еще один кризис – кризис ограниченной пропускной способности памяти. Современные компьютеры с 2-, 4- и 8-ядерными процессорами позволяют эффективно организовать взаимодействия через разделяемую память. Но представьте, как будет складываться ситуация с появлением 16-, 32- или 64-ядерных процессоров.

Если число ядер в одном процессоре продолжит увеличиваться с той же скоростью, разделяемая память станет узким местом, то есть нам придется беспокоиться о распределенной памяти. Компьютер будущего, возможно, останется в единственном корпусе, но с точки зрения программиста он, скорее всего, будет представлять собой кластер независимых компьютеров.

Это делает неизбежным, как мне кажется, повсеместное распространение приемов обмена сообщениями, по аналогии с акторами или моделью CSP.

Я думаю, что не удивлю вас, если скажу, что последние семь недель не стали для нас исчерпывающим исследованием возможностей в области конкурентного и параллельного программирования. Так что же мы пропустили?

Темы, оставшиеся за бортом

Одно из самых сложных решений, которые нам пришлось принять при создании этой книги, – определить, какие темы оставить за бор-

том. Ниже приводится краткое перечисление тем, которые не затрагивались здесь, а также даются некоторые указания, которым вы можете руководствоваться в собственных исследованиях.

Fork/Join и захват задачи

Fork/Join – это модель параллелизма, широко поддерживаемая в языке Cilk,¹ параллельной разновидности C/C++, но в настоящее время существуют ее реализации для многих окружений, включая Java.² Модель Fork/Join особенно хорошо подходит для реализации алгоритмов вида «разделяй и властвуй», как те, что мы видели в разделе «Разделяй и властвуй» в главе 3 (механизм свертки (reducers) в Clojure в действительности использует инфраструктуру Fork/Join языка Java).

Реализации модели Fork/Join обычно используют прием захвата задач для распределения задач между потоками выполнения, очень похожий на блоки `go` в Clojure (см. раздел «Блоки `go`» в главе 6).

Потоки данных

Мы коротко коснулись модели потоков данных в разделе «Потоки данных» в главе 3, но эта тема действительно заслуживает пристального внимания. Главная причина, почему эта тема не была включена в данную книгу, заключается в том, что ни одна из попыток создать универсальный потоковый язык не увенчалась большим успехом. Наиболее удачным примером может служить язык программирования Oz (часть системы программирования Mozart).³

Это не означает, что потоковое программирование неактуально, – как раз наоборот. Параллелизм на основе потоков данных чрезвычайно широко используется в аппаратных архитектурах – оба языка, VHDL⁴ и Verilog⁵, являются потоковыми языками.

Реактивное программирование

С потоками данных тесно связано реактивное программирование, когда программы автоматически реагируют на распространение изменений. В последнее время наблюдается рост интереса к реактивно-

¹ <http://www.cilkplus.org>

² <http://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html>

³ <http://mozart.github.io>

⁴ <https://ru.wikipedia.org/wiki/VHDL>

⁵ <https://ru.wikipedia.org/wiki/Verilog>

му программированию, в основном благодаря появлению библиотеки Rx⁶ (Reactive Extensions) компании Microsoft и др.⁷

В этой форме реактивное программирование имеет много общего с некоторыми технологиями, рассматривавшимися в данной книге, такими как топологии Storm, а также основанными на обмене сообщениями, такими как акторы и CSP.

Функциональное реактивное программирование

Функциональное реактивное программирование – это разновидность реактивного программирования, дополняющая функциональное программирование явным моделированием времени. Язык Elm позволяет писать сценарии, выполняющиеся под управлением браузера, и реализует функциональную версию ФРП.⁸ Подобно библиотеке `core.async`, он позволяет избежать «ада функций обратного вызова». Elm является одним из языков, охватываемых в следующей книге серии «Seven More Languages in Seven Weeks» [TDMD14].

Grid-вычисления

Grid-вычисления – это подход к организации распределенных вычислений в сети слабосвязанных кластеров. Элементы «решетки» (grid) обычно весьма разнородны и географически разбросаны по всему миру, могут присоединяться или покидать решетку при определенных условиях.

Наиболее известным примером grid-вычислений, вероятно, является проект SETI@Home, который позволяет любому желающему передать часть вычислительной мощности своего компьютера тем или иным проектам.⁹

Пространства кортежей

Пространство кортежей (tuple space) – это форма распределенной, ассоциативной памяти, которую можно использовать для организации взаимодействий между процессами. Парадигма пространств кортежей впервые была представлена в языке Linda¹⁰ (который, так

⁶ <https://rx.codeplex.com>

⁷ <https://github.com/Netflix/RxJava>

⁸ <http://elm-lang.org>

⁹ <http://setiathome.ssl.berkeley.edu>

¹⁰ <https://ru.wikipedia.org/wiki/Linda>

получилось, стал темой моей докторской диссертации, которую я защитил в начале 1990-х). В настоящее время существует несколько активно развивающихся систем, реализованных на основе парадигмы пространств кортежей.^{11 12}

Выбор за вами

Я – фанат автомобильной техники, поэтому все метафоры, которые я использовал в начале каждой главы, были на автомобильную тему. Как и автомобили, проблемы программирования имеют самые разные формы и размеры. Над какими бы вычислительными задачами вам не пришлось работать – подобными легкому гоночному автомобилю, солидному семейному минивэну или тяжелому грузовику, – я с уверенностью могу сказать, что параллелизм и конкуренция будут приобретать все большую и большую важность в вашей работе.

Я искренне надеюсь, независимо от того, будете вы их использовать или нет, что различные подходы, с которыми мы знакомились в течение последних семи недель, придадут вам уверенности в работе над вашими будущими проектами. Желаю вам безаварийного вождения!

¹¹ <http://river.apache.org/>

¹² <https://github.com/vjoel/tupelo>



БИБЛИОГРАФИЯ

- [Arm13] Joe Armstrong, «Programming Erlang: Software for a Concurrent World», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Second, 2013.¹
- [Goe06] Brian Goetz, «Java Concurrency in Practice», Addison-Wesley, Reading, MA, 2006.
- [HB12] Stuart Halloway and Aaron Bedra, «Programming Clojure», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, Second, 2012.²
- [MD14] Jack Moffitt and Fred Daoud, «Seven Web Frameworks in Seven Weeks: Adventures in Better Web Apps», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2014.
- [MW14] Nathan Marz and James Warren, «Big Data: Principles and best practices of scalable realtime data systems», Manning Publications Co., Greenwich, CT, 2014.
- [RW12] Eric Redmond and Jim R. Wilson, «Seven Databases in Seven Weeks: A Guide to Modern Databases and the NoSQL Movement», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2012.³
- [Tat10] Bruce A. Tate, «Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2010.⁴

¹ На русском языке есть аналогичная книга других авторов: Франческо Чезарини, Симон Томпсон. Программирование в Erlang. – М.: ДМК Пресс, 2012. – ISBN 978-5-94074-617-1, 978-0-596-51818-9. – *Прим. перев.*

² На русском языке есть аналогичная книга других авторов: Чаз Эмерик, Брайен Карпер, Кристоф Гранд. Программирование на Clojure. – М.: ДМК Пресс, 2013. – ISBN 978-5-94074-925-7, 978-1-449-39470-7. – *Прим. перев.*

³ Эрик Редмонд, Джим Р. Уилсон. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL. – М.: ДМК Пресс, 2012. – ISBN 978-5-94074-866-3, 978-1-93435-692-0. – *Прим. перев.*

⁴ Брюс Тейт. Семь языков за семь недель. – М.: ДМК Пресс, 2014. – ISBN 978-5-94074-539-6, 978-1-934356-59-3. – *Прим. перев.*

- [TDMD14] Bruce A. Tate, Fred Daoud, Jack Moffitt, and Ian Dees, «Seven More Languages in Seven Weeks», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2014.
- [Tho14] David Thomas, «Programming Elixir: Functional |> Concurrent |> Pragmatic |> Fun», The Pragmatic Bookshelf, Raleigh, NC and Dallas, TX, 2014.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Символы

#(...), макрос 136
.
(ClojureScript) 244
:require-macros (ClojureScript) 244
:trap_exit, флаг 169, 183

А

агенты 141
 и акторы 142
 и обработка ошибок 144
акторов, модель 164, 208
акторы
 в сравнении с CSP 252
анонимные функции
 в Clojure 96
 в Elixir 176
арифметико-логические устройства 256
арифметические операторы (Clojure) 84
асинхронный ввод/вывод 233
атомарные переменные
 в Clojure 127
 в Java 57
 и блокировки 58
атомы 127
 сравнение с программной
 транзакционной памятью 157

Б

базы данных, недостатки 313
барьеры (OpenCL) 279
блокирование
 и условные переменные 54
 методом перебора 51
блокировки
 встроенные, в Java 36, 42, 46
 и атомарные переменные 58

 и условные переменные 54
 несколько 39
 прерываемые 47
 приобретение в фиксированном
 порядке 41
 с таймаутами 49
Большие данные 295
 и лямбда-архитектура 315
 и проблемы баз данных 313
буферы
 в OpenGL 285
буферы (OpenCL) 262

В

валидаторы (Clojure) 135
веб-службы 113, 136
векторов литералы (Clojure) 85
вершинные буферы 285
взаимоблокировки
 в Java 41, 47
 и заикливание 50
 прерываемое блокирование 47
взаимоисключение 32
видимость памяти
 в Java 38
 в OpenCL 279
Википедия, подсчет слов 64
воронки 332, 335
встроенные блокировки 36, 46

Г

группировки потоков данных 340

Д

данные
 исходные и производные 315

неизменяемые 316, 324, 345
драйверы (Hadoop) 303

З

задача о пяти философях
 решение на Java 49, 55
задача о философях
 на Clojure (STM) 153
 на Clojure (атомы) 155
 решение на Java 41
зацикливание 50
защитное программирование 188
защищенные копии 44

И

идентификатор процесса (Elixir) 167
изменяемое состояние 80
 в Clojure 126
 в модели акторов 164
 скрытое, в Java 82
 ускользающее, в Java 83
имитация ряби на воде 284
императивные языки 127
 идентичность и состояние 134
индексные буферы 285

К

каналы
 буферизованные 214, 216, 240
 закрытие 214
 множество 227
 небуферизованные 214
 операции с каналами 222
 опрос 230
 создание 213
 чтение/запись 213
кластеры
 Amazon EMR 305
 локальные 196
 на основе Apache Spark 343
кластеры, локальные 340
ключевые слова
 в Clojure 85
 в Elixir 166

коллекции
 в Clojure 87, 90, 98, 131
 в Elixir 176
комбинаторы 309
конвейерная обработка 256
конкурентное программирование
 распределенные приложения 196
конкурентные ассоциативные массивы 70
конкурентные очереди 66
конкурентные программы
 для конкурентного мира 26
 и изменяемые данные 80
 и недетерминированность 23
 и неизменяемые данные 134
 и оптимизация 38
 и тестирование 76
 отказоустойчивость 208
параллельные программы 23
сопровождение 77
транзакции 146
конкуренция
 параллелизм 21
конкурентные программы
 надежность 206
 отказоустойчивость 206
константы (Clojure) 84
контексты (OpenGL) 260
кортежи
 в Elixir 166
 в Storm 332
кризис многоядерных систем 21

Л

ленивый последовательности 93
ленивые последовательности 108
лямбда-архитектура 295, 345
 пакетные представления 316
 пакетный уровень 296, 316, 324
 сервисный уровень 323, 324
 уровень ускорения 296, 327

М

массивов литералы (Clojure) 85
методы

- сторонние 43
- множество блокировок 39
- модели памяти 25
- модель акторов 164, 208
- модель памяти
 - в OpenCL 276, 279

Н

- недетерминизм
 - и параллельные программы 122
- недетерминированнизм
 - и конкурентные программы 23
- неизменяемость 134, 316, 345
 - в Clojure 81
 - в Elixir 345
 - в лямбда-архитектуре 324, 345

О

- обработка графики 255, 285
- обработка ошибок
 - в Elixir 182
 - в OpenCL 266
 - в агентах Clojure 144
- обработка событий
 - в ClojureScript 245
 - и глобальное состояние 218
- обработчики 43, 62
- овеществление
 - таймаутов 229
- опрос (CSP) 230, 236
- оптимизации компилятора 38
- оптимизация
 - и OpenCL 259, 294
- отбрасывающий буфер 216
- отказоустойчивость
 - и кластеры Hadoop 311
 - и человеческие ошибки 314
- отказоустойчивость
 - в Storm 334

П

- пакетные представления 316
- пакетный уровень 296, 316, 324
- память

- разделяемая 25
- распределенная 26
- параллелизм
 - Fork/Join, модель 347
 - в Storm 340
 - данных 24
 - и Большие данные 295
 - и модель CSP 253
 - и функциональные программы 108
- конкуренция 21
- на уровне битов 24
- на уровне задач 25
- на уровне инструкций 24
- потокное программирование 347
- поточный стиль программирования 110
- реактивное программирование 347
- функциональное реактивное
 - программирование 348
- параллелизм данных 254
- параллельная обработка
 - множество арифметико-логических устройств 256
- параллельное программирование
 - в Elixir 176
 - на Clojure 103
- параллельные архитектуры
 - конвейерная обработка 256
- параллелизм данных 24
- параллелизм на уровне битов 24
- параллелизм на уровне задач 25
- параллелизм на уровне инструкций 24
- параллельные программы
 - в OpenCL 258
 - графики производительности 73
 - дробление 258
 - и недетерминизм 122
 - и тестирование 76
 - на Clojure 95
 - на Java 64
 - на OpenCL 277
 - на основе Hadoop 299, 319
- пилюля с ядом 67
- поведения (behaviours) 193
- подсчет правок в Википедии 317

подсчет слов в Википедии
 параллельная реализация на основе Hadoop 301
 распараллеливание на Clojure 95
 функциональная программа 89
полиморфизм 101
порядок выполнения операций
 в конкурентных программах 38
порядок вычислений
 и функциональные программы 108
потоки выполнения
 в Java 31, 61, 82
 и тестирование 76
 пулы потоков 62
 управление состоятельностью 70
потокосовое программирование 347
потокосовый стиль программирования 110
почтовые ящики (Elixir) 167
предложения-ограничители (Elixir) 205
приостановка 218
программная транзакционная память
 в Clojure 146
 в других языках 162
 сравнение с атомами 157
производитель-потребитель 68
несколько потребителей 68
пространства кортежей 348
протоколы (Clojure) 100
процессы (Elixir) 174
 диспетчеры 185
 определение момента отказа 182
 порождение 167
 связывание 169
 управление состоянием 170
процессы-диспетчеры 185
 на основе OTP 195
 стратегии перезапуска 196
псевдопеременные 172
пулы потоков 62

Р

рабочие группы 275, 276
рабочие процессы
 в Storm 333

рабочие элементы 258, 270, 275, 276
разделяемая память 25
 и взаимноеисключение с помощью блокировок 32
распределенная память 26
распределенные приложения 196
распределенные программы 346
реактивное программирование 347
реализация протоколов 102
редуценты (reducers)
 в Clojure 99
редуцируемые значения (reducible), Clojure 99
рекурсия
 в Clojure 86
 в Elixir 169
Решето Эратосфена 223

С

свертка
 на OpenCL 277
сервисный уровень 323, 324
синхронизация
 java.util.concurrent, пакет 46
 со встроенными блокировками Java 36
 с применением барьеров 279
системные процессы (Elixir) 184
сита 332, 337
скользящий буфер 216
совместное использование структур 131
сообщения
 в Elixir 167, 172, 187, 197
 в модели CSP 212
сопровождение, многопоточного кода 77
состояние гонки 35, 37
сохраненные структуры данных
 в Clojure 130
 идентичность и состояние 134
специальная форма точка 244
ссылки (Clojure) 146
ссылочная прозрачность 109
сторонние методы 43
стратегии перезапуска 196
строки документирования (Clojure) 159

счетчик слов в Википедии
на языке Elixir 200

Т

таймауты
в Elixir 186
и каналы 228
тестирование
многопоточного кода 76
топологии 332, 338
транзакции (Clojure) 146

У

узлы (Erlang) 196
умножение матриц 270
управляющие конструкции (Clojure) 85
уровень ускорения 296, 327
переключение 331
синхронная и асинхронная
реализации 328
условные переменные 54

Ф

фаза перемешивания 300
функции
безопасные с точки зрения
транзакций 151
в Clojure 85
в Elixir 168, 192
неиспользуемые параметры 102
первого аорядка 176
функции-наблюдатели 135
функциональное программирование
определение 80
функциональное реактивное
программирование 348
функциональные программы
и параллелизм 108, 122
подсчет слов в Википедии 89
сумма чисел 86

Х

хвостовая рекурсия
в Elixir 169

Ш

шаблон Мастер 248

Я

ядра (OpenCL) 259, 270
ядро ошибки, шаблон проектирования 187

А

add-watch, функция 136
agent-error, функция 145
Akka, библиотека 162, 209
alter, функция 147, 148
alt!, макрос 227
alt!!, макрос 230
Amazon EMR 298, 305
Apache Spark, фреймворк 343
apply, функция 88, 159
ArrayBlockingQueue, класс 66
assoc, функция 90, 156
async/into, функция 215
async/merge, функция 239
AtomicInteger, класс 58
atom, функция 128
await(), метод (Java) 55
await, функция (Clojure) 143

С

chan, функция 213
Cilk, язык программирования 347
clBuildProgram(), функция 262
clCreateCommandQueue(), функция 261
clCreateContext(), функция 261
clCreateKernel(), функция 262
clCreateProgramWithSource(), функция 262
clEnqueueNDRangeKernel(), функция 263
clGetDeviceIDs(), функция 261, 265
clGetDeviceInfo(), функция 273
clGetEventProfilingInfo(), функция 264
clGetPlatformIDs(), функция 261
clj-http, библиотека 119
Clojure
ClojureScript 241
атомарные переменные 127

динамическая система типов 123
значение тождественности 88
изменяемые данные 129
и хвостовая рекурсия 160
макросы 218, 231
обзор синтаксиса 84
сохраненные структуры данных 130
ClojureScript 241
и JavaScript 243
close!, функция 214
clSetKernelArg(), функция 263, 280
commute, функция 148
compare-and-set!, функция 159
Comprojure, библиотека 115
ConcurrentHashMap, класс 70
ConcurrentLinkedQueue, класс 67
CopyOnWriteArrayList, класс 62
core.async, библиотека 212
CSP, модель 212
в сравнении с акторами 252
на стороне клиента 241

D

defn, форма 85, 101
deliver, функция 112
deref, функция
для агентов 141
для атомов 128
Dict.put(), функция 179
doseq, функция 115
dosync, функция 148

E

elastic-mapreduce, инструмент командной строки 306
ElephantDB, база данных 324
Elixir 165
акторы и процессы 165
определение момента отказа 182
параллелизм 176
процессы 167, 169
распределенные приложения 196
рекурсия 169
словари 179

сокрытие сообщений за фасадом API 171
сообщения 167, 172, 187, 197
функции 176, 192
Elm, язык программирования 348
ensure, функция 155
Erlang, язык программирования 165

F

filter<, функция 223
fold, функция 89
и параллелизм 103
Fork/Join, модель параллелизма 347
frequencies, функция 91
frequencies, функция (Clojure)
и параллелизм 105
future, форма 111

G

GenServer, программный компонент 193
get_global_id(), функция 270, 281
get_global_size(), функция 272, 281
get_local_id(), функция 281
get_local_size(), функция 281
get, функция 90
glBindBuffer(), функция 287
glBufferData(), функция 287
glGenBuffers(), функция 287
Google Closure, библиотека 244
go, блоки 217, 221
go, макрос 218
Go, язык программирования 212
GPGPU, программирование 254
grid-вычисления 348

H

Hadoop, фреймворк
драйверы 303
и файлы XML 308
комбинаторы 309
на Amazon EMR 305
на локальном компьютере 304
нумерация версий 299
обзор 298

- основы 299
- распределенная файловая система 311
- сортировка результатов 305
- средства отображения 299
- средства свертки 299
- Haskell 124, 162
- http-kit, библиотека 234

I

- is-sentence?, функция 117
- iterate, функция 93

J

- JavaScript
 - и ClojureScript 243
 - функции обратного вызова 248
- java.util.concurrent.atomic, пакет
 - атомарные примитивы 57
- java.util.concurrent, пакет 46
- Jetty, веб-сервер 116
- Joda-Time, библиотека 320

L

- Lightweight Java Graphics Library (LWJGL),
 - библиотека 284
- Linda, язык программирования 348
- listen, функция 247
- lockInterruptibly(), метод 49
- loop_system(), функция 185
- loop, макрос 160

M

- Mahout, проект 308
- make-reducer, функция (Clojure) 102
- make_ref(), функция (Elixir) 206
- mapcat<, функция 223
- mapcat, функция (Clojure) 92
- MapReduce, технология 297
 - и неизменяемые данные 316
 - реализация в фреймворке Hadoop 298
- map<, функция 223
- map, функция (Clojure) 91
- map(), функция (Elixir) 176

- merge-with, функция 96

O

- onto-chan, функция 215
- OpenCL 254
 - барьеры 279
 - буферы 262
 - возврат множества значений 265
 - доступ к буферу OpenGL 287
 - контексты 260
 - модель памяти 276, 279
 - модель платформы 275
 - обработка ошибок 266
 - поддержка CPU 274, 275
 - получение информации об устройстве 273
 - профилирование 264
 - рабочие группы 275, 276
 - рабочие элементы 258, 270, 275, 276
 - ядра 259, 270
- OpenGL 285
- OTP, библиотека 192
- Oz, язык программирования 347

P

- partial, функция 91
- partition-all, функция 99
- pmap, функция 95
- Process.register(), функция 174
- promise, форма 112
- putIfAbsent(), метод 71
- put!, функция 235

R

- range, функция 93
- receive, блок 168, 187
- recur, макрос 160
- reduce (Clojure) 87
- reductions, функция 118
- ReentrantLock, класс 47
 - блокирование методом перебора 52
 - прерываемые блокировки 49
 - условные переменные 57
- ref-set, функция 147

remove<, функция 224
replace(), метод 71
re-seq, функция 117
reset! , функция 128
restart-agent, функция 145
Rome, библиотека 236
Rx, библиотека 348

S

Schejulture, библиотека 138
send-off, функция (Clojure) 144
send-via, функция (Clojure) 144
send, функция (Clojure) 142, 151
send(), функция (Elixir) 166, 175
sentence-join, функция 118
spawn_link(), функция 169
spawn(), функция 171
spawn(), функция (Elixir) 167
Storm, система
 воронки 332, 335
 обзор 332
 отказоустойчивость 334
 рабочие процессы 333
 сита 332, 337
 топологии 332, 338

strings->sentences, функция 118
swap! , функция 128
swap!, функция 157
 повторения 134
synchronizedMap(), метод 68
s-выражения 84

T

to-chan, функция 223
tryLock(), метод 50

U

Useful, библиотека 139

V

volatile-непеременные 59
Voldemort, база данных 324

X

xmerl, библиотека 205

Y

yeld(), метод 34

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.**

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.aliants-kniga.ru.**

Электронный адрес: **books@aliants-kniga.ru.**

Оптовые закупки: тел. +7(499) 782-38-89.

Пол Батчер

Семь моделей конкуренции и параллелизма за семь недель

Главный редактор	<i>Мовчан Д. А.</i>
	<i>dmkpress@gmail.com</i>
Перевод с английского	<i>Киселев А. Н.</i>
Корректор	<i>Синяева Г. И.</i>
Верстка	<i>Паранская Н. В.</i>
Дизайн обложки	<i>Мовчан А. Г.</i>

Формат 60×90 ¹/₁₆. Гарнитура «Петербург».
Печать офсетная. Усл. печ. л. 29,25.
Тираж 200 экз.

Web-сайт издательства: www.dmk.ru