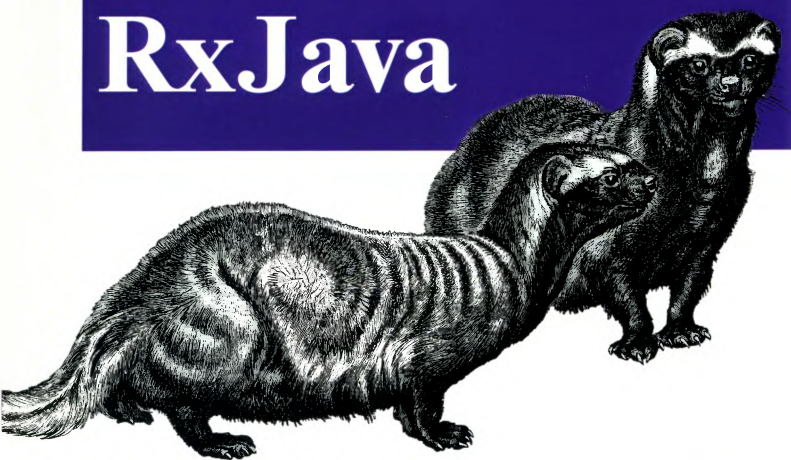


O'REILLY®

Реактивное программирование с применением RxJava



*Томаш Нуркевич
Бен Кристенсен*



Reactive Programming with RxJava

*Creating Asynchronous,
Event-Based Applications*

Tomasz Nurkiewicz and Ben Christensen

Реактивное программирование с применением RxJava

*Разработка асинхронных
событийно-ориентированных приложений*

Томаш Нуркевич, Бен Кристенсен

Москва, 2017



УДК 004.738.5:004.438 RxJava

ББК 32.973.2

M15

M15 Томаш Нуркевич, Бен Кристенсен

Реактивное программирование с применением RxJava / пер. с англ. Слинкин А. А. – М.: ДМК Пресс, 2017. – 358 с.: ил.

ISBN 978-5-97060-496-0

В наши дни, когда программы асинхронны, а быстрая реакция – важнейшее свойство, реактивное программирование поможет писать более надежный, лучше масштабируемый и быстрее работающий код. Благодаря этой книге программист на Java узнает о реактивном подходе к задачам и научится создавать программы, вобравшие в себя лучшие черты этой новой и весьма перспективной парадигмы. Данная книга содержит глубокое и подробное изложение концепций и принципов использования реактивного программирования вообще и RxJava в частности.

Книга может использоваться как для последовательного изучения предмета, так и в качестве справочника по библиотеке.

УДК 004.738.5:004.438 RxJava

ББК 32.973.2

Authorized Russian translation of the English edition of Reactive Programming with RxJava, ISBN 9781491931653. © 2017 Ben Christensen and Tomasz Nurkiewicz. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

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 978-1-49193-165-3 (англ.)

ISBN 978-5-97060-496-0 (рус.)

© Ben Christensen and Tomasz Nurkiewicz, 2017.

© Перевод на русский язык, оформление,
издание, ДМК Пресс, 2017

*Посвящается Паулине Съежжа, самому честному и прямому человеку
из всех моих знакомых.*

За доверие и помощь, отнюдь не ограничивающуюся написанием книги.

*За мою жизнь, которую она изменила в куда большей степени,
чем может себе представить.*

– Томаш Нуркевич

Оглавление

Предисловие	11
Вступление	14
Для кого написана эта книга	14
Несколько слов от Бена Кристенсена.....	14
Несколько слов от Томаша Нуркевича	16
О содержании книги	16
Ресурсы в Сети	17
Графические выделения	17
Как с нами связаться	18
Благодарности	18
От Бена	18
От Томаша	19
Глава 1. Реактивное программирование с применением RxJava	20
Реактивное программирование и RxJava	20
Когда возникает нужда в реактивном программировании.....	22
Как работает RxJava.....	23
Проталкивание и вытягивание.....	23
Синхронный и асинхронный режим	25
Конкурентность и параллелизм	28
Ленивые и энергичные типы.....	31
Двойственность	33
Одно или несколько?.....	34
Учет особенностей оборудования: блокирующий и неблокирующий ввод-вывод.....	39
Абстракция реактивности	44
Глава 2. Реактивные расширения	45
Анатомия rx.Observable.....	45
Подписка на уведомления от объекта Observable.....	48
Получение всех уведомлений с помощью типа Observer<T>	50
Управление прослушателями с помощью типов Subscription и Subscriber<T>	50

Создание объектов Observable	52
Подробнее о методе Observable.create()	53
Бесконечные потоки.....	56
Хронометраж: операторы timer() и interval().....	61
Горячие и холодные объекты Observable.....	61
Практический пример: от API обратных вызовов к потоку Observable	63
Управление подписчиками вручную	68
Класс rx.subjects.Subject.....	69
Тип ConnectableObservable	71
Реализация единственной подписки с помощью publish().refCount()	72
Жизненный цикл ConnectableObservable	74
Резюме.....	77
Глава 3. Операторы и преобразования	79
Базовые операторы: отображение и фильтрация.....	79
Взаимно однозначные преобразования с помощью map()	81
Обертывание с помощью flatMap()	85
Откладывание событий с помощью оператора delay()	90
Порядок событий после flatMap()	91
Сохранение порядка с помощью concatMap()	93
Более одного объекта Observable	95
Обращение с несколькими объектами Observable, как с одним, с помощью merge()	95
Попарная композиция с помощью zip() и zipWith()	97
Когда потоки не синхронизированы: combineLatest(), withLatestFrom() и amb().....	100
Более сложные операторы: collect(), reduce(), scan(), distinct() и groupBy()	106
Просмотр последовательности с помощью Scan и Reduce	106
Редукция с помощью изменяемого аккумулятора: collect().....	108
Проверка того, что Observable содержит ровно один элемент, с помощью single()	109
Устранение дубликатов с помощью distinct() и distinctUntilChanged()....	110
Выборка с помощью операторов skip(), takeWhile() и прочих	112
Способы комбинирования потоков: concat(), merge() и switchOnNext().....	114
Расщепление потока по условию с помощью groupBy().....	121
Написание пользовательских операторов.....	125
Повторное использование операторов с помощью compose().....	125
Реализация более сложных операторов с помощью lift()	127
Резюме.....	133
Глава 4. Применение реактивного программирования в существующих приложениях.....	134
От коллекций к Observable	135
BlockingObservable: выход из реактивного мира	135

О пользе лени	138
Композиция объектов Observable	140
Ленивое разбиение на страницы и конкатенация	141
Императивная конкурентность	142
flatMap() как оператор асинхронного сцепления	148
Замена обратных вызовов потоками	153
Периодический опрос изменений	156
Многопоточность в RxJava	157
Что такое диспетчер?	158
Декларативная подписка с помощью subscribeOn()	167
Конкурентность и поведение subscribeOn()	171
Создание пакета запросов с помощью groupBy()	175
Декларативная конкурентность с помощью observeOn()	177
Другие применения диспетчеров	180
Резюме	181
Глава 5. Реактивность сверху донизу	183
Решение проблемы C10k	183
Традиционные HTTP-серверы на основе потоков	185
Неблокирующий HTTP-сервер на основе Netty и RxJetty	187
Сравнение производительности блокирующего и реактивного сервера	195
Обзор реактивных HTTP-серверов	200
Код HTTP-клиента	201
Доступ к реляционной базе данных	205
NOTIFY и LISTEN на примере PostgreSQL	207
CompletableFuture и потоки	211
Краткое введение в CompletableFuture	211
Сравнение типов Observable и Single	220
Создание и потребление объектов типа Single	221
Объединение ответов с помощью zip, merge и concat	223
Интероперабельность с Observable и CompletableFuture	225
Когда использовать тип Single?	226
Резюме	227
Глава 6. Управление потоком и противодавление	228
Управление потоком	228
Периодическая выборка и отбрасывание событий	228
Скользящее окно	237
Пропуск устаревших событий с помощью debounce()	238
Противодавление	243
Противодавление в RxJava	244
Встроенное противодавление	247
Производители и исключение MissingBackpressureException	250
Учет запрошенного объема данных	253
Резюме	259

Глава 7. Тестирование и отладка	260
Обработка ошибок.....	260
А где же мои исключения?	261
Декларативная замена try-catch.....	264
Таймаут в случае отсутствия событий.....	268
Повтор после ошибки.....	271
Тестирование и отладка	275
Виртуальное время	275
Диспетчеры и автономное тестирование	277
Автономное тестирование.....	279
Мониторинг и отладка.....	287
Обратные вызовы doOn...()	287
Измерение и мониторинг	289
Резюме.....	292
Глава 8. Практические примеры	293
Применение RxJava в разработке программ для Android.....	293
Предотвращение утечек памяти в компонентах Activity.....	294
Библиотека Retrofit со встроенной поддержкой RxJava	296
Диспетчеры в Android	301
События пользовательского интерфейса как потоки	304
Управление отказами с помощью Hystrix.....	307
Hystrix: первые шаги.....	308
Неблокирующие команды и HystrixObservableCommand	310
Паттерн Переборка и быстрое прекращение.....	311
Пакетирование и объединение команд	313
Мониторинг и инструментальные панели	318
Опрос баз данных NoSQL.....	321
Клиентский API Couchbase	322
Клиентский API MongoDB	323
Интеграция с Camel	325
Потребление файлов с помощью Camel	325
Получение сообщений от Kafka	326
Потоки Java 8 и CompletableFuture	326
Полезность параллельных потоков.....	328
Выбор подходящей абстракции конкурентности	330
Когда выбирать Observable?.....	331
Потребление памяти и утечки	332
Операторы, потребляющие неконтролируемый объем памяти.....	332
Резюме.....	337
Глава 9. Направления будущего развития	338
Реактивные потоки	338
Типы Observable и Flowable	338
Производительность	339

Миграция.....	340
Приложение А. Дополнительные примеры	
HTTP-серверов.....	341
Системный вызов <code>fork()</code> в программе на C	341
Один поток – одно подключение	343
Пул потоков для обслуживания подключений	345
Приложение В. Решающее дерево для выбора	
операторов Observable	347
Об авторах	352
Об изображении на обложке	352
Предметный указатель	353



Предисловие

28 октября 2005 года Рэй Оззи (Ray Ozzie), только что назначенный главным архитектором Майкрософт, разослал своим сотрудникам получившее скандальную известность письмо, озаглавленное «Крах Интернет-служб». В нем он по сути дела описывает тот мир, который мы знаем сегодня, – мир, где такие корпорации, как Microsoft, Google, Facebook, Amazon и Netflix используют веб в качестве основного канала доставки своих услуг.

В письме Оззи есть мысль, которую разработчики не часто слышат от топ-менеджеров крупной корпорации:

Сложность убивает. Она высасывает соки из разработчиков, затрудняет планирование, сборку и тестирование продуктов, порождает проблемы в части безопасности и служит причиной горьких разочарований пользователей и администраторов.

Прежде всего, следует принять во внимание, что в 2005 году крупные ИТ-компании были всем сердцем влюблены в умопомрачительно сложные технологии типа SOAP, WS-* и XML. В то время еще не было слова «микросервисы» и даже на горизонте не просматривалась простая технология, которая позволила бы разработчикам справиться с проблемами составления асинхронной композиции простых служб для получения более сложных, не упуская из виду такие аспекты, как обработка ошибок, задержки, безопасность и эффективность.

Для моей группы облачных программных продуктов письмо Оззи стало призывом не заниматься ерундой, а сосредоточиться на придумывании простой модели программирования, которая позволит создавать крупномасштабные асинхронные архитектуры Интернет-служб для работы с большими объемами данных. После многих фальстартов на нас наконец снизошло озарение: взяв за основу интерфейсы Iterable/Iterator для синхронных коллекций, мы могли бы получить пару интерфейсов для представления потоков асинхронных событий и применять всем хорошо знакомые операции над последовательностями – map, filter, scan, zip, groupBy и другие – к преобразованию и комбинированию асинхронных потоков данных. Так летом 2007 года родилась идея Rx. В процессе реализации мы поняли, что необходимо как-то управлять конкурентностью и временем и с этой целью обобщили идею исполнителей в Java, дополнив ее виртуальным временем и кооперативной многозадачностью.

После напряженной двухлетней работы, когда были опробованы и отвергнуты разнообразные проектные решения, 18 ноября 2009 года мы наконец выпустили

первую версию Rx.NET. Вскоре после этого мы перенесли Rx на Microsoft.Phone.Reactive для Windows Phone 7 и приступили к реализации Rx на таких языках, как JavaScript и C++, а заодно поэкспериментировали с Ruby и Objective-C.

Внутри Майкрософт первым пользователем Rx стал Джафар Хусейн (Jafar Husain), эту технологию он взял с собой, когда перешел в Netflix в 2011 году. Джафар всячески пропагандировал Rx в компании и в конце концов полностью переделал клиентскую часть пользовательского интерфейса Netflix на основе асинхронной обработки потоков. И, к счастью для всех нас, он заразил своим энтузиазмом Бена Кристенсена, занимавшегося в Netflix разработкой API промежуточного уровня. В 2012 году Бен начал работать над RxJava и в начале 2013 разместил весь код на Github, сделав его открытым. Еще одним из ранних приверженцев Rx в Майкрософт был Пол Беттс (Paul Betts), позже он перешел в Github и убедил коллег, в т. ч. Джастина Спар-Саммерса (Justin Spahr-Summers) реализовать и выпустить ReactiveCocoa для Objective-C, что и произошло весной 2012.

Поскольку Rx завоевывал популярность в отрасли, мы убедили отдел Microsoft Open Tech раскрыть код Rx.NET, это случилось осенью 2012. Вскоре после этого я ушел из Майкрософт, основал компанию Applied Duality и посвятил все свое время тому, чтобы сделать Rx стандартным кросс-языковым и кросс-платформенным API для асинхронной обработки потоков данных в режиме реального времени.

К 2016 году популярность Rx стремительно возросла, как и число пользователей. Весь трафик, проходящий через Netflix API, обрабатывается RxJava. То же можно сказать о библиотеке обеспечения отказоустойчивости Hystrix, лежащей в основе всего внутреннего трафика службы, и сопутствующих реактивных библиотеках RxNetty и Mantis. Сейчас Netflix работает над полностью реактивным сетевым стеком для связывания всех внутренних служб, пересекающих границы процессов и машин. RxJava нашла также весьма полезные применения в экосистеме Android. Компании SoundCloud, Square, NYT, Seatgeek используют RxJava в своих приложениях и вносят вклад в разработку дополнительной библиотеки RxAndroid. Такие поставщики NoSQL-решений, как Couchbase и Splunk, также предлагают основанные на Rx интерфейсы к уровню доступа к данным. Среди других Java-библиотек, воспринявших RxJava, упомянем Camel Rx, Square Retrofit и Vert.x. В сообществе JavaScript широко распространена библиотека RxJS, лежащая в основе популярного каркаса Angular 2. Сообщество поддерживает сайт <http://reactivex.io/>, на котором можно найти информацию о реализациях Rx на многих языках, а также фантастические камешковые диаграммы с пояснениями, созданные Дэвидом Гроссом (@CallHimMoorlock).

С самого начала проект Rx развивался в соответствии с потребностями сообщества разработчиков и при его активном участии. В оригинальной реализации Rx в .NET упор был сделан, прежде всего, на преобразовании асинхронных потоков событий и использовании асинхронных перечислимых объектов в ситуациях, где требуется противодействие. Поскольку в Java нет языковой поддержки асинхронного ожидания, сообщество дополнило типы Observer и Observable концепцией реактивного вытягивания и добавило интерфейс Producer. Благодаря усилиям

многих разработчиков реализация RxJava получилась весьма изощренной и в высшей степени оптимизированной.

Несмотря на то что детали RxJava несколько отличаются от других реализаций Rx, библиотека все равно ориентирована специально на разработчиков, стремящихся выжить в прекрасном новом мире распределенной обработки данных в реальном времени и сконцентрироваться на эссенциальной, а не акцидентальной сложности, высасывающей из нас все соки. Эта книга содержит глубокое и подробное изложение концепций и принципов использования Rx вообще и RxJava в частности, написанное двумя авторами, которые потратили бесчисленное количество часов на реализацию RxJava и применение ее к реальным задачам. Если вам нужна «реактивность», то лучшего способа, чем купить книгу, не придумаешь.

— Эрик Мейер, основатель и президент
компании *Applied Duality, Inc.*

Вступление

Для кого написана эта книга

Книга ориентирована на Java-программистов средней и высокой квалификации. Читатель должен свободно владеть языком Java, но предварительное знакомство с реактивным программированием не предполагается. Многие описываемые концепции относятся к функциональному программированию, но знакомство с ним также не обязательно. Особенно полезна книга будет двумя группам программистов.

- Профессионалы, которым нужно повысить производительность сервера или сделать код для мобильных устройств более удобным для сопровождения. Если вы из их числа, то найдете здесь идеи и готовые решения реальных проблем, а также практические советы. А RxJava тогда следует считать просто еще одним инструментом, который книга поможет освоить.
- Любопытные разработчики, которые слышали о реактивном программировании или конкретно о RxJava и хотели бы понять, что это такое. Если вы относите себя к этой категории и не планируете немедленно использовать преимущества RxJava в производственном коде, то книга заметно обогатит ваш багаж знаний.

Наконец, это книга станет подспорьем для практикующего архитектора программного обеспечения. RxJava оказала сильное влияние на общую архитектуру целых систем, поэтому знать об этой технологии полезно. Но даже если вы только начинаете путешествие в мир программирования, все равно попробуйте прочитать первые главы, в которых объяснены основы. Понятия преобразования и композиции универсальны и не являются спецификой реактивного программирования.

Несколько слов от Бена Кристенсена

В 2012 году я работал над новой архитектурой Netflix API. По ходу дела стало ясно, что для достижения поставленных целей необходимо включить конкурентность и асинхронные сетевые запросы. Исследуя различные подходы, я столкнулся с Джафаром Хусейном (<https://github.com/jhusain>), который попытался заинтересовать меня технологией Rx, с которой познакомился, работая в Майкрософт. В то время я довольно сносно владел техникой конкурентного программирования,

но размышлял о нем в императивных терминах и преимущественно с точки зрения Java, поскольку именно программированием на Java зарабатывал себе на хлеб.

Поэтому мне трудно было воспринять пропагандируемый Джафаром подход из-за его функциональной ориентированности, и я не поддавался на его убеждения. За этим последовали месяцы споров и дискуссий, архитектура системы становилась все более зрелой, а мы с Джафаром снова и снова встречались у доски, пока я, наконец, не врубился в теоретические принципы, а затем и не оценил элегантность и эффективность подходов, предлагаемых Rx.

Мы решили включить модель программирования Rx в Netflix API и в конечном итоге создали реализацию на Java, которую назвали RxJava, следуя заданным Майкрософт образцам: Rx.Net и RxJS.

За примерно три года, когда создавалась библиотека RxJava, по большей части на GitHub, в виде открытого кода, я имел удовольствие работать с растущим сообществом и соавторами, каковых было 120 с лишним, и вместе нам удалось превратить RxJava в зрелый продукт, используемый во многих системах как на стороне сервера, так и на стороне клиента. Он собрал больше 15 000 звезд на GitHub, что позволило войти в первые 200 проектов (<https://github.com/search?p=11&q=stars:%3E1&s=stars&type=Repositories>) и занять третье место среди проектов на Java (<https://github.com/search?l=Java&p=1&q=stars:%3E1&s=stars&type=Repositories>).

Джордж Кэмпбелл (George Campbell), Аарон Талл (Aaron Tull) и Мэтт Джекобс (Matt Jacobs) из Netflix много сделали для превращения RxJava из первых сборок в то, чем она является теперь. В частности, им проект обязан добавлением `lift`, `subscriber`, противодействия и поддержки других языков на базе JVM. Дэвид Карнок (David Karnok) присоединился к проекту позже, но уже обошел меня по числу фиксаций и написанных строк кода. Ему проект в значительной мере обязан своим успехом, а теперь он возглавил его.

Хочу поблагодарить Эрика Мейера, который создал Rx во время работы в Майкрософт. С тех пор как он уволился оттуда, я урывками общался с ним в Netflix, когда трудился над RxJava, а теперь счастлив работать вместе в Facebook. Я считаю большой честью обсуждать с ним различные вопросы у доски и учиться у него. С таким наставником, как Эрик, поднимаешься на новый уровень мышления.

Попутно мне приходилось много раз выступать на конференциях с докладами о RxJava и реактивном программировании, и я повстречал много людей, которые помогли мне узнать о коде и архитектуре куда больше, чем я мог бы достичь собственными силами.

Netflix оказала мне феноменальное содействие, позволяя тратить время и силы на этот проект и выделив специалистов для написания технической документации, – сам я с этим точно не справился бы. Проект с открытым исходным кодом такого масштаба и качества никогда не стал бы успешным, если бы у меня не было возможности заниматься им в рабочее время и привлекать людей с разными знаниями и умениями.

Первая глава книги представляет собой мою попытку объяснить, почему реактивное программирование вообще полезно и как конкретно в RxJava реализованы общие принципы.

Весь остальной текст написан Томашем, который проделал потрясающую работу. У меня была возможность читать черновики и вносить предложения, но это его книга, и именно он писал обо всех деталях, начиная со второй главы.

Несколько слов от Томаша Нуркевича

Я впервые столкнулся с RxJava в 2013 году, работая в одной финансовой организации. Мы занимались обработкой больших потоков рыночных данных в реальном времени. В тот момент конвейер состоял из Kafka (доставка сообщений), Akka (обработка данных о торговых сделках), Clojure (преобразование данных) и специально разработанный язык для распространения изменений по всей системе. Технология RxJava казалась очень соблазнительным выбором, поскольку предлагала единообразный API, отлично приспособленный для работы с разными источниками данных.

Со временем я пробовал применять реактивное программирование и в других ситуациях, где требовалась высокая масштабируемость и пропускная способность. Для реализации реактивных систем, безусловно, приходится прикладывать больше усилий. Но и выгода велика, в частности, более полное использование возможностей оборудования и, стало быть, экономия энергии. Чтобы по-настоящему оценить преимущества этой модели программирования, разработчик должен располагать относительно простым инструментарием. Мы полагаем, что Reactive Extensions – удачный компромисс между уровнем абстракции, сложностью и производительностью.

В этой книге рассматривается версия RxJava 1.1.6, если явно не оговорено противное. Хотя RxJava может работать с версиями Java, начиная с Java 6, почти во всех примерах применяется синтаксис лямбда-выражений, появившийся в Java 8. В некоторых примерах из главы 8, посвященной Android, продемонстрированы более многословные синтаксические конструкции без лямбда-выражений. Но все же мы не везде используем самый лаконичный синтаксис (например, ссылки на методы), стремясь сделать код понятнее там, где это имеет смысл.

О содержании книги

Книга написана так, что наибольшую пользу даст последовательное чтение от корки до корки. Но если столько времени у вас нет, то можете выбирать самые интересные для себя части. Если какое-то понятие было введено раньше, то в большинстве случаев мы даем на него обратную ссылку. Ниже приведен краткий обзор глав.

- В *главе 1* содержится очень краткое введение в основные идеи и понятия RxJava (*Бен*).
- В *главе 2* объясняется, как в вашем приложении может появиться библиотека RxJava и как с ней взаимодействовать. Здесь все довольно просто, но ясное понимание таких понятий, как горячий и холодный источник, очень важно для дальнейшего (*Томаш*).

- *Глава 3* – краткий экскурс в многочисленные операторы, имеющиеся в RxJava. Мы познакомимся с выразительными и мощными функциями, лежащими в основе этой библиотеки (*Томаш*).
- *Глава 4* носит более практический характер, здесь показано, как включать RxJava в различные места кода. Затрагивается также вопрос о конкурентности (*Томаш*).
- В более продвинутой *главе 5* объясняется, как реализовать реактивное приложение от начала до конца (*Томаш*).
- В *главе 6* рассказано о важной проблеме управления потоком и о том, как она решается в RxJava с помощью механизмов противодействия (*Томаш*).
- В *главе 7* описаны методы автономного тестирования, сопровождения и отладки приложений на основе Rx (*Томаш*).
- В *главе 8* приведены избранные примеры приложений RxJava, особенно в распределенных системах (*Томаш*).
- *Глава 9* посвящена планам развития RxJava 2.x (*Бен*).

Ресурсы в Сети

Все камешковые диаграммы, встречающиеся в книге, взяты из официальной документации по RxJava (<https://github.com/ReactiveX/RxJava/wiki>), опубликованной на условиях лицензии Apache License Version 2.0.

Графические выделения

В книге применяются следующие графические выделения:

Курсив

Новые термины, URL-адреса, адреса электронной почты, имена и расширения имен файлов.

Моноширинный

Листинги программ, а также элементы кода в основном тексте: имена переменных и функций, базы данных, типы данных, переменные окружения, предложения и ключевые слова языка.

Моноширинный полужирный

Команды и иные строки, которые следует вводить буквально.

Моноширинный курсив

Текст, вместо которого следует подставить значения, заданные пользователем или определяемые контекстом.



Таким значком обозначается совет или рекомендация общего характера.



Таким значком обозначается замечание общего характера.



Таким значком обозначается предупреждение или предостережение.

Как с нами связаться

Вопросы и замечания по поводу этой книги отправляйте в издательство:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (в США и Канаде)
707-829-0515 (международный или местный)
707-829-0104 (факс)

Для этой книги имеется веб-страница, на которой публикуются списки замеченных ошибок, примеры и прочая дополнительная информация. Адрес страницы: <http://bit.ly/reactive-prog-with-rxjava>.

Замечания и вопросы технического характера следует отправлять по адресу bookquestions@oreilly.com.

Дополнительную информацию о наших книгах, конференциях и новостях вы можете найти на нашем сайте по адресу <http://www.oreilly.com>.

Читайте нас на Facebook: <http://facebook.com/oreilly>.

Следите за нашей лентой в Twitter: <http://twitter.com/oreillymedia>.

Смотрите нас на YouTube: <http://www.youtube.com/oreillymedia>.

Благодарности

От Бена

Этой книги не было бы без Томаша, который написал большую часть текста, и Нэн Барбер, нашего редактора, которая с достойным восхищения терпением помогала нам до самого конца работы. Спасибо Томашу, откликнувшемуся на мое объяв-

Я также высоко ценю поддержку центра Netflix Open Source (<https://netflix.github.io/>) и Дэниэля Джекобсона (https://twitter.com/daniel_jacobson), оказанную мне лично и проекту в целом. Они были прекрасными спонсорами проекта, только благодаря им я мог уделять столько времени сообществу. Спасибо!

От Томаша

Исправив ошибки, найденные рецензентами, Венкат Субраманиам (Venkat Subramaniam) немало постарался, чтобы придать книге ясную и логичную структуру. Нередко он предлагал поменять порядок предложений, абзацев и глав или даже удалить целые страницы, не имеющие отношения к делу. Еще одним рецензентом – весьма знающим и опытным – был Дэвид Карнок. Будучи руководителем проекта RxJava, он нашел десятки ошибок, состояний гонки, несогласованностей и других проблем. Оба рецензента написали сотни замечаний, которые заметно улучшили качество книги. На ранних этапах работы рукопись читали многие мои коллеги, поделившиеся своим мнением. Выражаю благодарность Дариушу Бачински, Шимону Хома, Петру Петжаку, Якубу Пилимону, Адаму Войщику, Марчину Зайончковски и Мачею Зярко.

Глава 1.

Реактивное программирование с применением RxJava

RxJava – это конкретная реализация технологии реактивного программирования для Java и Android, на которую большое влияние оказало функциональное программирование. В RxJava отдается предпочтение композиции функций без глобального состояния и побочных эффектов, а также применению потоков для составления асинхронных событийно-ориентированных программ. За основу взят паттерн Наблюдатель, абстрагирующий обратные вызовы производителя и потребителя, и затем расширен десятками операторов, обеспечивающих композицию, преобразование, диспетчеризацию, дросселирование (throttling), обработку ошибок и управление жизненным циклом.

RxJava – зрелая библиотека с открытым исходным кодом (<https://github.com/ReactiveX/RxJava>), широко используемая как на серверах, так и на мобильных устройствах на платформе Android. Вокруг библиотеки RxJava и реактивного программирования в целом сложилось активное сообщество разработчиков (<http://reactivex.io/tutorials.html>), которые дополняют проект, выступают на конференциях, пишут статьи и помогают друг другу.

Эта глава содержит краткий обзор библиотеки RxJava – что это такое и как работает – а в остальной части книги детально описано, как ее использовать в приложениях. Для чтения книги необязательно иметь опыт реактивного программирования, мы начнем с самого начала и познакомим вас с идеями и практическим применением RxJava, чтобы вам было проще понять, поможет ли она в вашей конкретной ситуации.

Реактивное программирование и RxJava

Термином «реактивное программирование» обозначают технологию программирования, в которой акцент делается на реакции на изменения, например, на изменение значений данных или на события. Это можно сделать – и зачастую делается – императивно. Императивное реактивное программирование основано на обратных вызовах. Отличный пример реактивного программирования – элек-

тронные таблицы: если одна ячейка зависит от других, то она автоматически «реагирует» на изменение значений в них.

Функциональное реактивное программирование?

Хотя идеи функционального программирования оказали большое влияние на «реактивные расширения» (Reactive Extensions – Rx вообще и RxJava в частности), эту технологию нельзя назвать «функциональным реактивным программированием» (FRP). FRP – это весьма специфический частный случай реактивного программирования (<http://stackoverflow.com/questions/1028250/what-is-functional-reactive-programming/1030631#1030631>), в котором рассматривается непрерывное время, тогда как RxJava имеет дело только с дискретными событиями во времени. Я и сам впадал в это заблуждение в начале работы над RxJava, когда описывал библиотеку как «функциональную реактивную», пока не осознал, что это кажущееся таким естественным сочетание слов много лет назад уже было зарезервировано для чего-то совсем другого. Поэтому не существует никакого общепринятого термина, который описывал бы назначение RxJava более конкретно, чем «реактивное программирование». Акроним FRP все еще широко – и неправильно – используется для описания RxJava и других подобных библиотек, а в Интернете продолжают спорить, что правильнее: расширить значение термина (поскольку он неформально употребляется в этом смысле уже несколько лет) или сохранить его только за реализациями с непрерывным временем.

Устранив это недоразумение, мы можем сосредоточиться на том факте, что RxJava действительно создана под влиянием функционального программирования и в ее основу сознательно положена модель, отличная от императивного программирования. В этой главе слово «реактивный» обозначает реактивно-функциональный стиль, характерный для RxJava. Напротив, говоря «императивный», я не имею в виду, что реактивное программирование нельзя реализовать императивно, а лишь подчеркиваю, что императивное программирование противоположно функциональному подходу, принятому в RxJava. Сравнивая императивный и функциональный подходы, я буду употреблять термины «реактивно-функциональный» и «реактивно-императивный».

В современных компьютерах любой подход в какой-то момент оказывается императивным, потому что любая программа в конечном итоге опускается на уровень операционной системы и оборудования. Компьютеру необходимо явно сказать, что и как делать. Люди думают иначе, чем процессоры и основанные на них системы, поэтому мы добавляем уровни абстрагирования. Реактивно-функциональное программирование – это абстракция, как и высокоуровневые идиомы императивного программирования, абстрагирующие машинные команды. О том, что в конечном итоге любая программа императивна, не следует забывать, потому что это помогает выстроить умозрительную модель задачи, решаемой реактивно-функциональным программированием, и понять, как она в итоге выполняется, – никакой магии здесь нет.

Таким образом, реактивно-функциональное программирование – это такой подход к программированию (абстракция поверх императивных систем), кото-

рый позволяет писать асинхронные и событийно-ориентированные программы, не заставляя себя думать, как компьютер, и императивно описывать сложные взаимодействия состояний, особенно пересекающих границы потоков и сетей. Возможность не подражать «мышлению» компьютера – вещь полезная при разработке асинхронных и событийно-ориентированных систем, поскольку тут речь идет о вопросах конкурентности и параллелизма, где корректность и эффективность крайне важны, но трудно достижимы.

В сообществе Java эталонами глубины и широты охвата тематики, связанной со сложностями конкурентного программирования, считаются книги Brian Goetz «Java Concurrency in Practice» и Doug Lea «Concurrent Programming in Java» (Addison-Wesley), а также форумы типа «Mechanical Sympathy» (<https://groups.google.com/forum/m/#!forum/mechanical-sympathy>). Общаясь с экспертами, появляющимися на этих форумах, и вообще с членами сообщества в период начала работы над RxJava, я острее, чем когда-либо прежде, осознал, как трудно написать высокопроизводительную, эффективную, масштабируемую и вместе с тем корректную конкурентную программу. А мы ведь даже не упомянули распределенные системы, где конкурентность и параллелизм поднимаются на совсем другой уровень.

Таким образом, короткий ответ на вопрос, чем занимается реактивно-функциональное программирование, звучит так: конкурентностью и параллелизмом. А если прибегнуть к неформальной терминологии, то оно устраняет «ад обратных вызовов», являющийся неизбежным результатом императивного решения реактивных и асинхронных задач. Реактивное программирование в той форме, какая реализована в RxJava, основано на применении функционального программирования, в нем используется декларативный подход, позволяющий обойти типичные для реактивно-императивного стиля ловушки.

Когда возникает нужда в реактивном программировании

Реактивное программирование полезно в следующих ситуациях.

- Обработка событий, инициированных пользователем, например: перемещение мыши, щелчки мышью, ввод с клавиатуры, изменение сигналов GPS вследствие перемещения пользователя вместе со своим устройством, сигналы от встроенного гироскопа, события касания пальцем и т. д.
- Обработка любых событий ввода-вывода от диска или сети, характеризующихся наличием задержки. В этих случаях ввод-вывод по самой своей природе асинхронный (отправлен запрос, проходит время, затем получен – или не получен – ответ, запускающий дальнейшую обработку).
- Обработка событий или данных, поступающих приложению от производителя, которого оно не может контролировать (системные события сервера, вышеупомянутые пользовательские события, сигналы от оборудования, события аналоговых датчиков и т. д.).

Если программа обрабатывает только один поток событий, то реактивно-императивный стиль на основе обратных вызовов вполне может подойти, а написание реактивно-функциональной программы не принесет особой выгоды. Даже если различных потоков событий много сотен, но все они независимы, то и тогда императивное программирование годится. В таких простых случаях императивный подход оказывается наиболее эффективным, потому что отсутствует уровень абстракции реактивного программирования, т. е. программа оказывается ближе к тому уровню, для которого оптимизированы современные операционные системы, языки и компиляторы.

Но в большинстве программ приходится комбинировать события (или асинхронные ответы на вызовы функций или обращения в сеть), в них присутствует условная логика взаимодействия между событиями и необходимо обрабатывать ошибки, в т. ч. освобождать захваченные ресурсы. И вот тогда реактивно-императивный подход приводит к неприемлемому росту сложности, а достоинства реактивно-функционального программирования становятся очевидны. В результате я сформулировал для себя такую ненаучную точку зрения: начальные затраты на изучение реактивно-функционального программирования гораздо выше, но сложность результирующей программы гораздо ниже, чем в случае реактивно-императивного программирования.

Стало быть, одной фразой технологию Rx вообще и RxJava в частности можно описать так: «библиотека для разработки асинхронных событийно-ориентированных программ». Библиотека RxJava – это конкретная реализация принципов реактивного программирования, созданная под влиянием идей функционального программирования и программирования потоков данных. Есть различные подходы к «реактивности», и RxJava – лишь один из них. Разберемся, как она работает.

Как работает RxJava

В центре RxJava находится тип `Observable`, представляющий поток данных или событий. Он предназначен для проталкивания (push) (реактивность), но может использоваться и для вытягивания (pull) (интерактивность). Тип является ленивым (lazy), а не энергичным (eager). Допускает как синхронное, так и асинхронное использование. Может представлять 0, 1, много и даже бесконечно много значений или событий во времени.

В этом абзаце много технических терминов и деталей, нуждающихся в пояснении. Полное описание приведено в разделе «Анатомия `rx.Observable`» главы 2.

Проталкивание и вытягивание

Весь смысл реактивности RxJava в том, чтобы поддержать режим проталкивания, поэтому сигнатуры методов типа `Observable` и связанного с ним типа `Observer` поддерживают поступление входящих событий. Это естественно сопровождается поддержкой асинхронности, о чем пойдет речь в следующем разделе. Но тип `Observable` поддерживает также асинхронный канал обратной связи (ино-

гда употребляются также термины асинхронное вытягивание или реактивное вытягивание) для управления потоком или реализации противодействия в асинхронных системах. Ниже в этой главе мы поговорим об управлении потоком и роли этого механизма.

Для поддержки входящих событий объекты типа `Observable` и `Observer` связываются посредством подписки. Объект типа `Observable` представляет поток данных, на который может подписаться объект типа `Observer` (подробнее о нем – в разделе «Получение всех уведомлений с помощью типа `Observer<T>`» главы 2):

```
interface Observable<T> {  
    Subscription subscribe(Observer s)  
}
```

После оформления подписки объект `Observer` может получать события трех типов:

- данные – с помощью функции `onNext()`;
- ошибки (объекты типа `Throwable`) – с помощью функции `onError()`;
- события завершения потока – с помощью функции `onCompleted()`.

```
interface Observer<T> {  
    void onNext(T t)  
    void onError(Throwable t)  
    void onCompleted()  
}
```

Метод `onNext()` может вызываться сколько угодно раз, в т. ч. ни одного. Методы же `onError()` и `onCompleted()` – терминальные в том смысле, что может быть вызван только один из них и только один раз. После терминального события поток `Observable` завершается, и больше никаких событий в нем появиться не может. Терминальное событие может и не наступить, если поток бесконечен и работает без ошибок.

В разделах «Управление потоком» и «Противодавление» главы 6 мы познакомимся также с типом для поддержки интерактивного вытягивания:

```
interface Producer {  
    void request(long n)  
}
```

Он используется в сочетании с производным от `Observer` типом `Subscriber` (дополнительные сведения о нем см. в разделе «Управление прослушивателями с помощью типов `Subscription` и `Subscriber<T>`» главы 2):

```
interface Subscriber<T> implements Observer<T>, Subscription {  
    void onNext(T t)  
    void onError(Throwable t)  
    void onCompleted()  
    ...  
    void unsubscribe()  
    void setProducer(Producer p)  
}
```

Метод `unsubscribe` интерфейса `Subscription` позволяет подписчику отписаться от потока `Observable`. Метод `setProducer` и тип `Producer` служат для установления двустороннего канала связи между производителем и потребителем, что необходимо для управления потоком.

Синхронный и асинхронный режим

Обычно объект `Observable` асинхронный, но это не обязательно. Он может быть и синхронным и по умолчанию таковым и является. RxJava не включает асинхронный режим, если ее об этом не просят. Если на синхронный объект `Observable` подписаться, то он будет передавать все данные в потоке подписчика, а затем завершится (если поток конечный). Если за объектом `Observable` стоит блокирующий сетевой ввод-вывод, то он будет синхронно блокировать поток подписчика и передавать событие методу `onNext()` после возврата из блокирующего обращения к сети.

Например, следующий объект ведет себя синхронно:

```
Observable.create(s -> {
    s.onNext("Hello World!");
    s.onCompleted();
}).subscribe(hello -> System.out.println(hello));
```

Подробнее о методе `Observable.create` написано в разделе «Метод `Observable.create()`» главы 2, а о методе `Observable.subscribe` – в разделе «Подписка на уведомления от объекта `Observable`» там же.

Но вы, наверное, подозреваете, что такое поведение в реактивной системе нежелательно, – и в этом вы совершенно правы. Категорически не рекомендуется использовать `Observable` для синхронного блокирующего ввода-вывода (если уж приходится работать с блокирующим вводом-выводом, то его следует сделать асинхронным с помощью потоков). Но иногда синхронный доступ оправдан, например, чтобы извлечь и сразу же вернуть данные из кэша в памяти. В показанном выше примере «Hello World» конкурентность не нужна, более того, из-за асинхронной диспетчеризации он будет работать гораздо медленнее. В общем случае решающий критерий – является ли производитель событий `Observable` блокирующим или неблокирующим, а не синхронным или асинхронным. Пример «Hello World» неблокирующий, потому что он ни в каком случае не блокирует поток выполнения, поэтому такое использование `Observable` допустимо (хотя без него вполне можно обойтись).

Тип `Observable` в RxJava намеренно не знает о различиях между синхронностью и асинхронностью и о том, имеет ли место конкурентность и, если да, то каковы ее источники. Это позволяет вызывающей программе решать, что лучше. Почему это может оказаться полезно?

Во-первых, конкурентность возникает необязательно из-за использования пула потоков. Если источник данных уже является асинхронным, поскольку включен в цикл обработки событий, то RxJava не добавит накладных расходов на диспетчеризацию и не заставит использовать конкретный механизм диспетчеризации. Ис-

точниками конкурентности могут быть пулы потоков, циклы обработки событий, акторы и т. д. Ее можно добавить, или она может быть присуща самому источнику данных. RxJava безразлична природе асинхронности.

Во-вторых, есть две основательные причины использовать синхронное поведение, и мы рассмотрим их ниже.

Данные в памяти

Если данные находятся в кэше в памяти (с постоянным временем поиска порядка микро или наносекунд), то не имеет смысла делать доступ асинхронным и нести расходы на диспетчеризацию. Объект `Observable` может синхронно извлечь данные и передать их непосредственно потоку подписчика:

```
Observable.create(s -> {
    s.onNext(cache.get(SOME_KEY));
    s.onCompleted();
}).subscribe(value -> System.out.println(value));
```

Выбрать способ диспетчеризации можно динамически в зависимости от того, находятся данные в памяти или нет. Если в памяти, передаем синхронно, иначе выполняем сетевой вызов и возвращаем данные, когда они поступят:

```
// псевдокод
Observable.create(s -> {
    T fromCache = getFromCache(SOME_KEY);
    if(fromCache != null) {
        // передаем данные синхронно
        s.onNext(fromCache);
        s.onCompleted();
    } else {
        // получаем асинхронно
        getDataAsynchronously(SOME_KEY)
        .onResponse(v -> {
            putInCache(SOME_KEY, v);
            s.onNext(v);
            s.onCompleted();
        })
        .onFailure(exception -> {
            s.onError(exception);
        });
    }
}).subscribe(s -> System.out.println(s));
```

Синхронное вычисление (как в операторах)

Более частая причина синхронности – композиция и преобразование потоков¹ (stream) с помощью операторов. В RxJava для манипулирования, комбинирования и преобразования данных обычно применяются разнообразные операторы,

¹ К сожалению, в русскоязычной литературе слово «поток» многозначно. Оно означает «поток данных» (stream), «поток выполнения» (thread) и «поток управления» (flow). Хочется надеяться, что из контекста понятно, в каком смысле употреблено слово. – *Прим. перевод.*

например: `map()`, `filter()`, `take()`, `flatMap()`, `groupBy()`. Большинство операторов синхронно, т. е. вычисление целиком выполняется внутри `onNext()` по мере поступления событий.

Операторы сделаны синхронными из соображений производительности. Рассмотрим пример:

```
Observable<Integer> o = Observable.create(s -> {
    s.onNext(1);
    s.onNext(2);
    s.onNext(3);
    s.onCompleted();
});

o.map(i -> "Число " + i)
  .subscribe(s -> System.out.println(s));
```

Если бы оператор `map` по умолчанию был асинхронным, то каждое из чисел 1, 2, 3 нужно было бы передать в поток для выполнения конкатенации строк ("Число" + `i`). Это очень неэффективно и в общем случае дает недетерминированную задержку из-за диспетчеризации, контекстного переключения и т. д.

Важно понимать, что большинство конвейеров функций в объекте `Observable` выполняется синхронно (если только оператор по своей природе не является асинхронным, как, например, `timeout` или `observeOn()`), тогда как сам объект `Observable` может быть асинхронным. Этот вопрос более подробно обсуждается в разделе «Организация декларативной конкурентности с помощью метода `observeOn()`» главы 4 и в разделе «Таймаут в случае отсутствия событий» главы 7.

В примере ниже демонстрируется смешение синхронного и асинхронного выполнения:

```
Observable.create(s -> {
    ... асинхронная подписка и порождение данных ...
})
.doOnNext(i -> System.out.println(Thread.currentThread()))
.filter(i -> i % 2 == 0)
.map(i -> "Значение " + i + " обработано в потоке " +
    Thread.currentThread())
.subscribe(s -> System.out.println("КАКОЕ-ТО ЗНАЧЕНИЕ => " + s));
System.out.println("Печатается ДО порождения значений");
```

В этом примере объект `Observable` асинхронный (данные порождаются не в том потоке, в каком работает подписчик), поэтому вызов `subscribe` неблокирующий, а `println` в конце выводит сообщение до того, как получено первое событие и напечатана строка "КАКОЕ-ТО ЗНАЧЕНИЕ =>".

Однако функции `filter()` и `map()` выполняются синхронно в том потоке, который порождает события. В общем случае это как раз то, что нам нужно: асинхронный конвейер (`observable` вместе с операторами), в котором вычисления над событиями производятся синхронно, т. е. максимально эффективно.

Таким образом, тип `Observable` поддерживает как синхронное, так и асинхронное поведение, и это осознанное проектное решение.

Конкурентность и параллелизм

Отдельные потоки `Observable` не допускают ни конкурентности, ни параллелизма. То и другое достигается путем композиции асинхронных объектов `Observable`.

Под параллелизмом понимается истинно одновременное выполнение задач, обычно на разных процессорах или компьютерах. С другой стороны, конкурентность – это чередование нескольких задач. Если одному процессору назначено несколько задач (к примеру, потоков выполнения), то они выполняются не параллельно, а конкурентно – с помощью механизма квантования времени. Каждый поток получает квант времени процессора, а затем уступает процессор другому потоку, даже если еще не завершил работу.

Параллельное выполнение, по определению, является конкурентным, но обратное неверно. На практике это означает, что многопоточная программа всегда конкурентна, но параллелизм возникает, лишь если потоки выполняются на разных процессорах строго в одно и то же время. Поэтому мы обычно говорим о конкурентности, понимая, что параллелизм – частный случай этого понятия.

Согласно контракту типа `Observable`, никакие события (`onNext()`, `onCompleted()`, `onError()`) не могут возникать одновременно. Иными словами, один поток данных `Observable` всегда должен быть сериализованным и потокобезопасным. События могут порождаться в разных потоках выполнения при условии, что это не происходит одновременно. Это означает, что чередующиеся или одновременные обращения к методу `onNext()` невозможны. Если `onNext()` еще выполняется в одном потоке, то никакой другой поток не может еще раз вызвать его (это и называется чередованием). Код в следующем примере правилен:

```
Observable.create(s -> {
    new Thread(() -> {
        s.onNext("one");
        s.onNext("two");
        s.onNext("three");
        s.onNext("four");
        s.onCompleted();
    }).start();
});
```

Здесь данные порождаются последовательно, так что контракт выполнен. (Отметим, однако, что, вообще говоря, не рекомендуется запускать потоки изнутри `Observable`, как в этом примере. Пользуйтесь вместо этого диспетчерами, как описано в разделе «Многопоточность в RxJava» главы 4.)

А такой код недопустим:

```
// НЕ ПОСТУПАЙТЕ ТАК
Observable.create(s -> {
    // Поток A
    new Thread(() -> {
        s.onNext("one");
        s.onNext("two");
```

```
}).start();

// Поток B
new Thread(() -> {
    s.onNext("three");
    s.onNext("four");
}).start();

// из-за гонки потоков игнорируется необходимость вызова s.onCompleted()
});
// НЕ ПОСТУПАЙТЕ ТАК
```

Этот под недопустим, потому что существуют два потока, в которых может одновременно вызываться метод `onNext()`. Это нарушение контракта. (Кроме того, нужно было бы безопасно дожидаться завершения обоих потоков путем вызова `onComplete` и, как уже сказано выше, вручную запускать потоки таким образом – вообще неудачная идея.)

Ну и как же воспользоваться конкурентностью или параллелизмом в RxJava? С помощью композиции.

Один поток `Observable` всегда сериализован, но разные потоки могут работать независимо, в частности, конкурентно или параллельно. Именно поэтому в RxJava так часто употребляются методы `merge` и `flatMap` – чтобы конкурентно выполнить композицию асинхронных потоков данных. (Подробнее об этих методах написано в разделах «Обертывание с помощью `flatMap()`» и «Обращение с несколькими объектами `Observable`, как с одним, с помощью `merge()`» главы 3.)

Ниже приведен искусственный пример, демонстрирующий, как объединяются два асинхронных объекта `Observable`, порождаемых в разных потоках:

```
Observable<String> a = Observable.create(s -> {
    new Thread(() -> {
        s.onNext("one");
        s.onNext("two");
        s.onCompleted();
    }).start();
});

Observable<String> b = Observable.create(s -> {
    new Thread(() -> {
        s.onNext("three");
        s.onNext("four");
        s.onCompleted();
    }).start();
});

// конкурентно подписывается на a и b и объединяет
// их в третий последовательный поток данных
Observable<String> c = Observable.merge(a, b);
```

Объект `Observable c` получает элементы из потоков `a` и `b`. Вследствие асинхронности справедливы следующие утверждения:

- "one" предшествует "two";
- "three" предшествует "four";
- порядок следования пары one-two и three-four не определен.

А почему бы не разрешить конкурентные вызовы `onNext()`?

Прежде всего, потому что `onNext()` предназначен для использования программистом, а учитывать конкурентность трудно. Если разрешить конкурентные вызовы `onNext()`, то при кодировании любого объекта `Observer` нужно было бы предусматривать возможность конкурентного вызова, даже если это неожиданно или нежелательно.

Вторая причина заключается в том, что если данные могут порождаться конкурентно, то некоторые операции просто невыполнимы, например, такие важные и распространенные, как `scan` и `reduce`. Разрешение конкурентных потоков `Observable` (с чередованием вызовов `onNext()`) привело бы к ограничениям на типы допустимых событий и потребовало бы использования потокобезопасных структур данных.



Тип `java.util.stream.Stream` в Java 8 поддерживает конкурентное порождение данных. Именно поэтому требуется, чтобы метод `reduce` был ассоциативным (<http://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html#reduce-java.util.function.BinaryOperator->). Документация по пакету `java.util.stream` (<http://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html>) в части параллелизма, упорядочения (тесно связанного с коммутативностью), операций редукции и ассоциативности иллюстрирует те сложности, с которыми приходится сталкиваться, если тип `Stream` допускает последовательное и конкурентное порождение данных.

Третья причина – накладные расходы на синхронизацию, поскольку все наблюдатели и операторы должны быть потокобезопасными, даже если в большинстве случаев данные поступают последовательно. И хотя JVM часто удается устранить эти расходы, все-таки это возможно не всегда (особенно в неблокирующих алгоритмах с использованием атомарных операций), поэтому приходится жертвовать производительностью даже в случае последовательных потоков, когда такие жертвы не нужны.

Кроме того, часто обобщенная реализация параллелизма с мелким уровнем детализации оказывается медленнее. Распараллеливать лучше большие куски работы, чтобы не нести расходы на переключение потоков, диспетчеризацию и объединение результатов. Гораздо эффективнее синхронно выполнить операцию в одном потоке, имея возможность воспользоваться разнообразными оптимизациями памяти и процессора. В случае коллекций `List` и `array` легко задать разумные умолчания для пакетного параллелизма, поскольку все элементы известны заранее и можно разбить весь объем работы на порции (хотя даже в этой ситуации часто быстрее обработать весь список на одном процессоре, если только он не очень длинный и время обработки одного элемента не слишком велико). Но для потока

данных объем работы заранее неизвестен, мы просто получаем очередной элемент с помощью метода `onNext()`. Поэтому автоматически разбить работу на части невозможно.

На самом деле, до выхода версии RxJava v1 был добавлен оператор `.parallel(Function f)`, стремящийся имитировать поведение `java.util.stream.Stream.parallel()`, поскольку считалось, что это будет удобно. Сделано это было таким образом, чтобы не нарушать контракт RxJava: один поток событий `Observable` разбивался на несколько, исполняемых параллельно, а затем потоки снова объединялись. Но кончилось все изъятием этого оператора из библиотеки (<https://github.com/ReactiveX/RxJava/blob/e8041725306b20231fcc1590b2049ddcb9a38920/CHANGES.md#removed-observableparallel>), т. к. он только служил источником недоразумений и почти всегда приводил к снижению производительности. Добавление вычислительного параллелизма в поток событий почти всегда нуждается в тщательном осмыслении и тестировании. Быть может, тип `ParallelObservable` и имел бы смысл – при условии, что на операторы наложено ограничение ассоциативности, – но за годы использования RxJava в этом никогда не возникало острой необходимости, поскольку композиции, включающие `merge` и `flatMap`, вполне эффективно справляются с возникающими на практике ситуациями.

В главе 3 мы рассмотрим, как с помощью операторов составлять композиции объектов `Observable`, способные воспользоваться всеми преимуществами конкурентности.

Ленивые и энергичные типы

Тип `Observable` *ленивый*, т. е. ничего не делает, пока на него кто-то не подпишется. Этим он отличается от *энергичных* типов, например `Future`, который, будучи создан, уже представляет активную работу. Благодаря ленивости композиция объектов `Observable` не приводит к потере данных из-за состояния гонки без кэширования. В случае `Future` это не проблема, потому что одиночное значение можно кэшировать, так что если значение доставлено до формирования композиции, оно все равно будет получено. Но если поток данных неограничен, то для предоставления аналогичной гарантии понадобился бы неограниченный буфер. Поэтому тип `Observable` ленивый и не начинает работу до оформления подписки, так что композицию можно целиком создать еще до того, как начнется поступление данных.

На практике это означает две вещи.

- *Сигналом к началу работы является подписка, а не конструирование*

Благодаря ленивости `Observable` создание объекта этого типа не приводит к началу выполнения работы (если не считать «работой» выделение памяти для самого объекта `Observable`). В этот момент лишь определяется, какую работу предстоит выполнить, когда кто-то подпишется на объект. Рассмотрим такое определение `Observable`:

```
Observable<T> someData = Observable.create(s -> {  
    getDataFromServerWithCallback(args, data -> {
```

```

        s.onNext(data);
        s.onCompleted();
    });
}

```

Ссылка `someData` уже существует, но функция `getDataFromServerWithCallback` еще не выполнялась. Пока что только объявлена обертка `Observable` вокруг единицы работы, которую еще предстоит выполнить.

Эта работа начнет выполняться, когда где-то будет создана подписка на объект `Observable`:

```
someData.subscribe(s -> System.out.println(s));
```

- *Объекты `Observable` можно использовать повторно*

Тот факт, что тип `Observable` ленивый, позволяет использовать один экземпляр несколько раз. Это означает, что следующая последовательность операций законна:

```

someData.subscribe(s -> System.out.println("Subscriber 1: " + s));
someData.subscribe(s -> System.out.println("Subscriber 2: " + s));

```

Теперь у нас есть две отдельных подписки, каждая из них вызывает функцию `getDataFromServerWithCallback` и порождает события.

Такого рода ленивость отличается от асинхронных типов, например `Future`, поскольку созданный объект `Future` представляет уже начатую работу. Объекты `Future` нельзя использовать повторно (подписываться на них несколько раз, чтобы инициировать работу). Если существует ссылка на `Future`, значит, работа уже производится. В предыдущем примере ясно видно, где именно реализована энергичность: метод `getDataFromServerWithCallback` энергичный, потому что выполняется сразу в момент обращения. А обертывание его объектом `Observable` позволяет отложить вызов, т. е. превратить метод в ленивый.

Преимущества ленивости наглядно проявляются при построении композиции, например:

```

someData
    .onErrorResumeNext(lazyFallback)
    .subscribe(s -> System.out.println(s));

```

В данном случае объект `lazyFallback` типа `Observable` представляет работу, которая *потенциально может быть* выполнена, но будет реально выполнена, только если кто-то подпишется на объект. Кроме того, мы хотим подписаться лишь на уведомление об ошибке `someData`. Разумеется, энергичные типы можно сделать ленивыми с помощью вызова функций (например, `getDataAsFutureA()`).

У энергичности и ленивости есть свои плюсы и минусы, но в RxJava тип `Observable` ленивый. Помните, что объект `Observable` ничего не сделает, если на него не подписаться.

Эта тема обсуждается подробнее в разделе «О пользе лени» главы 4.

Двойственность

Тип `Observable` является асинхронным «двойником» типа `Iterable`. Говоря «двойник», мы имеем в виду, что `Observable` предоставляет всю функциональность `Iterable`, но с противоположным направлением потока данных: проталкивание вместо вытягивания. В следующей таблице показано, как эти типы обслуживают проталкивание и вытягивание.

Вытягивание (<code>Iterable</code>)	Проталкивание (<code>Observable</code>)
<code>T next()</code>	<code>onNext(T)</code>
throws <code>Exception</code>	<code>onError(Throwable)</code>
returns	<code>onCompleted()</code>

Как видим, данные не вытягиваются потребителем с помощью метода `next()`, а проталкиваются производителем путем обращения к `onNext(T)`. Об успешном завершении сигнализирует обратный вызов `onCompleted()`, а не блокировка потока до тех пор, пока не завершится обход всех элементов. Место исключений, распространяющихся вверх по стеку, занимают ошибки, генерируемые в виде событий, передаваемых методу обратного вызова `onError(Throwable)`.

Такая двойственность означает, что любое действие, которое можно выполнить синхронно путем вытягивания с помощью объектов `Iterable` и `Iterator`, может быть также выполнено асинхронно путем проталкивания с помощью объектов `Observable` и `Observer`. Следовательно, в обоих случаях применима одна и та же модель программирования!

Например, в версии Java 8 тип `Iterable` можно модернизировать, добавив композицию функций с помощью типа `java.util.stream.Stream`:

```
// Iterable<String> как Stream<String>,
// содержащий 75 строк
getDataFromLocalMemorySynchronously()
    .skip(10)
    .limit(5)
    .map(s -> s + "_transformed")
    .forEach(System.out::println)
```

Этот код получает 75 строк от метода `getDataFromLocalMemorySynchronously()`, игнорирует все элементы, кроме занимающих позиции с 11 по 15, преобразует строки и печатает их. (Подробнее об операторах `take`, `skip` и `limit` рассказано в разделе «Выборка с помощью методов `skip()`, `takeWhile()` и прочих» главы 3.)

В RxJava тип `Observable` используется аналогично:

```
// Observable<String>,
// порождающий 75 строк
getDataFromNetworkAsynchronously()
    .skip(10)
    .take(5)
    .map(s -> s + "_transformed")
    .subscribe(System.out::println)
```

Здесь мы получаем 5 строк (порождено 15, но 10 из них отброшено), а затем отписываемся (игнорируя последующие строки или прекращая их порождение). Код преобразования и печати строки ничем не отличается от предыдущего примера.

Иными словами, тип `Observable` допускает асинхронное программирование путем проталкивания данных точно так же, как тип `Stream`, обертывающий типы `Iterable` и `List`, используется для синхронного вытягивания.

Одно или несколько?

Тип `Observable` поддерживает асинхронное проталкивание нескольких значений и, следовательно, попадает в правый нижний угол показанной ниже таблицы, оказываясь асинхронным двойником типа `Iterable` (или `Stream`, `List`, `Enumerable` и т. д.) и многозначной версии `Future`:

	Один	Несколько
Синхронно	<code>T getData()</code>	<code>Iterable<T> getData()</code>
Асинхронно	<code>Future<T> getData()</code>	<code>Observable<T> getData()</code>

Отметим, что в этом разделе тип `Future` рассматривается обобщенно. Его поведение описывается методом `Future.onSuccess(callback)`. Существуют различные реализации, например: `CompletableFuture` (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html>), `ListenableFuture` (<https://github.io/guava/releases/snapshot/api/docs/com/google/common/util/concurrent/ListenableFuture.html>) или `Scala Future` (<http://docs.scala-lang.org/overviews/core/futures.html>). Но ни в коем случае не следует использовать тип `java.util.Future`, который блокирует текущий поток, чтобы получить значение.

Так в чем же ценность `Observable` по сравнению с простым `Future`? Очевидная причина заключается в том, что мы имеем дело либо с потоком событий, либо с многозначным ответом. Менее очевидная – композиция нескольких однозначных ответов. Рассмотрим то и другое по очереди.

Поток событий

В потоке событий нет ничего сложного. Производитель проталкивает события потребителю, как показано в следующем фрагменте кода:

```
// производитель
Observable<Event> mouseEvents = ...;

// потребитель
mouseEvents.subscribe(e -> doSomethingWithEvent(e));
```

С типом `Future` все не так гладко:

```
// производитель
Future<Event> mouseEvents = ...;

// потребитель
mouseEvents.onSuccess(e -> doSomethingWithEvent(e));
```


Функция обратного вызова `onSuccess` могла бы получить «последнее событие», но остается ряд вопросов. Должен ли потребитель производить опрос? Будет ли производитель ставить события в очередь или же все события, произошедшие между двумя операциями выборки, теряются? Определенно у `Observable` есть преимущества. Не будь `Observable`, лучше было бы моделировать эту ситуацию с помощью обратных вызовов, а не `Future`.

Многозначность

Многозначные ответы – еще одно применение `Observable`. По сути дела, всюду, где можно использовать `List`, `Iterable` или `Stream`, подойдет и `Observable`:

```
// производитель
Observable<Friend> friends = ...

// потребитель
friends.subscribe(friend -> sayHello(friend));
```

Это можно сделать и с помощью `Future`:

```
// производитель
Future<List<Friend>> friends = ...

// потребитель
friends onSuccess(listOfFriends -> {
    listOfFriends.forEach(friend -> sayHello(friend));
});
```

Тогда зачем использовать `Observable<Friend>`?

Если список возвращаемых данных мал, то с точки зрения производительности выбор не имеет значения, это дело вкуса. Но если список велик или удаленный источник данных получает разные части списка из различных мест, то подход на основе `Observable<Friend>` может повысить производительность или снизить задержку.

Но самая убедительная причина состоит в том, что элементы можно обрабатывать по мере поступления, а не дожидаться получения всей коллекции. Это особенно важно, когда сетевые задержки для каждого элемента могут различаться, что типично для задержек с вытянутым хвостом (как в сервисно-ориентированных или микросервисных архитектурах) и разделяемых хранилищ данных. Если дожидаться всей коллекции, то потребитель всегда будет наблюдать максимальную задержку, имевшую место для какого-то элемента коллекции. Если же элементы возвращаются в виде потока `Observable`, то потребитель получает их сразу же, и «время до получения первого элемента» может оказаться существенно ниже максимальной задержки. Но при этом придется пожертвовать упорядоченностью элементов потока, т. е. смириться с тем, что элементы могут обрабатываться не в том порядке, в каком порождались. Если потребителю важен порядок, то можно включить в состав данных элемента или метаданных ранг или позицию элемента, тогда клиент сможет правильно отсортировать поступившие элементы.

Ко всему прочему, потребление памяти ограничено размером одного элемента, выделять память для всей коллекции не нужно.

Композиция

Многозначный тип `Observable` полезен также, когда производится композиция однозначных ответов, например, из объектов `Future`.

Результатом объединения нескольких объектов `Future` является новый объект `Future` с одним значением, например:

```
CompletableFuture<String> f1 = getDataAsFuture(1);
CompletableFuture<String> f2 = getDataAsFuture(2);

CompletableFuture<String> f3 = f1.thenCombine(f2, (x, y) -> {
    return x+y;
});
```

Иногда это именно то, что нам нужно, и для этой цели в RxJava предусмотрен метод `Observable.zip` (подробнее о нем см. раздел «Попарная композиция с помощью методов `zip()` и `zipWith()`» главы 3):

```
Observable<String> o1 = getDataAsObservable(1);
Observable<String> o2 = getDataAsObservable(2);

Observable<String> o3 = Observable.zip(o1, o2, (x, y) -> {
    return x+y;
});
```

Однако это означает, что для порождения результата нужно дождаться завершения всех объектов `Future`. Зачастую предпочтительно отдавать возвращенное значение `Future`, как только объект завершится. В таком случае лучше воспользоваться методом `Observable.merge` (или родственным ему методом `flatMap`). Он позволяет помещать композицию результатов (даже если это просто объект `observable`, порождающий одно значение) в поток значений, которые отдаются по мере готовности:

```
Observable<String> o1 = getDataAsObservable(1);
Observable<String> o2 = getDataAsObservable(2);

// теперь o3 - поток, состоящий из o1 и o2, который порождает элементы
// без ожидания
Observable<String> o3 = Observable.merge(o1, o2);
```

Тип Single

Хотя тип `Observable` прекрасно справляется с многозначными потоками, при проектировании API и с точки зрения потребления удобнее однозначное представление в силу своей простоты. К тому же, простейшее поведение запрос-ответ встречается в приложениях сплошь и рядом. Поэтому RxJava предоставляет тип `Single` — ленивый эквивалент `Future`. Можете считать, что это тип `Future` с двумя

полезными дополнениями: во-первых, он ленивый, т. е. на него можно подписываться несколько раз и легко производить композицию, а, во-вторых, он согласован с RxJava и потому допускает простое взаимодействие с Observable.

Рассмотрим, к примеру, следующие аксессоры:

```
public static Single<String> getDataA() {
    return Single.<String> create(o -> {
        o.onSuccess("DataA");
    }).subscribeOn(Schedulers.io());
}

public static Single<String> getDataB() {
    return Single.just("DataB")
        .subscribeOn(Schedulers.io());
}
```

Из них можно составить такую композицию:

```
// объединить a и b в поток Observable, содержащий 2 значения
Observable<String> a_merge_b = getDataA().mergeWith(getDataB());
```

Обратите внимание, как два объекта Single объединяются в один Observable. В результате могут быть порождены значения [A, B] или [B, A] в зависимости от того, кто завершится первым.

Возвращаясь к предыдущему примеру, мы теперь можем использовать для представления выборки данных объекты Single вместо Observable, но объединить их в поток значений:

```
// Observable<String> o1 = getDataAsObservable(1);
// Observable<String> o2 = getDataAsObservable(2);

Single<String> s1 = getDataAsSingle(1);
Single<String> s2 = getDataAsSingle(2);

// Теперь o3 - поток значений s1 и s2, который отдает их без ожидания
Observable<String> o3 = Single.merge(s1, s2);
```

Использование single вместо Observable для представления «потока с одним элементом» упрощает потребление, потому что разработчику нужно рассматривать только следующие варианты поведения типа Single:

- возвращает в ответ ошибку;
- вообще не отвечает;
- возвращает в ответ значение.

А теперь посмотрим, какие дополнительные состояния должен учитывать потребитель, имея дело с объектом Observable:

- возвращает в ответ ошибку;
- вообще не отвечает;
- отвечает без ошибки, не возвращая никаких данных, затем завершается;
- отвечает без ошибки, возвращая одно значение, затем завершается;

- отвечает без ошибки, возвращая несколько значений, затем завершается;
- отвечает без ошибки, возвращая одно или несколько значений и никогда не завершается (ждет поступления дополнительных данных).

Использование `single` упрощает мысленную модель потребления данных, лишь после композиции в объект `Observable` разработчику приходится рассматривать дополнительные состояния. И зачастую так делать удобнее, потому что этот код разработчик обычно контролирует, тогда как API работы с данными определен третьей стороной.

Подробнее о типе `single` см. раздел «Сравнение типов `Observable` и `Single`» главы 5.

Тип `Completable`

Помимо типа `single`, в RxJava имеется тип `Completable`, рассчитанный на ситуацию, когда никакие данные не возвращаются, а нужно лишь знать, успешно или неудачно завершилась операция, — этот случай встречается на удивление часто. Можно было бы воспользоваться типом `Observable<Void>` или `Single<Void>`, но это некрасиво, поэтому на помощь приходит `Completable`:

```
Completable c = writeToDatabase("data");
```

Это типично в ситуации, когда производится асинхронная запись — возвращаемого значения нет, но надо знать, как завершилась операция: успешно или неудачно. Показанное выше предложение аналогично такому:

```
Observable<Void> c = writeToDatabase("data");
```

Сам тип `Completable` абстрагирует два обратных вызова: успешное завершение и ошибка:

```
static Completable writeToDatabase(Object data) {
    return Completable.create(s -> {
        doAsyncWrite(data,
            // обратный вызов в случае успешного завершения
            () -> s.onCompleted(),
            // обратный вызов в случае ошибки типа Throwable
            error -> s.onError(error));
    });
}
```

От нуля до бесконечности

Тип `Observable` поддерживает от нуля до бесконечного числа элементов (см. раздел «Бесконечные потоки» главы 2). Но ради простоты и понятности введены дополнительные типы: `Single` — «`Observable` с одним элементом» и `Completable` — «`Observable` без элементов».

С учетом этих типов наша таблица принимает такой вид:

	Ноль	Один	Несколько
Синхронно	<code>void doSomething()</code>	<code>T getData()</code>	<code>Iterable<T> getData()</code>
Асинхронно	<code>Completable doSomething()</code>	<code>Future<T> getData()</code>	<code>Observable<T> getData()</code>

Учет особенностей оборудования: блокирующий и неблокирующий ввод-вывод

До сих пор аргументом в пользу реактивно-функционального стиля программирования было преимущественно абстрагирование асинхронных обратных вызовов с целью упростить составление композиций. Очевидно, что параллельное выполнение несвязанных сетевых запросов лучше последовательного с точки зрения наблюдаемой задержки, отсюда и стремление к асинхронности и композиции.

Но оказывает ли влияние на эффективность реактивного подхода (все равно, императивного или функционального) способ выполнения ввода-вывода? Есть ли какие-нибудь преимущества у неблокирующего ввода-вывода или блокирование потоков выполнения в ожидании ответа на сетевой запрос ничем не хуже? Я принимал участие в тестировании производительности Netflix, где было убедительно продемонстрировано, что неблокирующий ввод-вывод и циклы обработки событий дают объективно измеримый прирост эффективности по сравнению с блокированием потока при каждом запросе. В этом разделе объясняется, почему это так, а также приводятся данные, которые помогут вам принять собственное решение.

Как сказано во врезке, сравнительные тесты производительности блокирующего и неблокирующего ввода-вывода проводились для программ Tomcat и Netty на платформе Linux. Поскольку такого рода тесты всегда вызывают споры, а корректно осуществить их трудно, я хочу предельно четко сформулировать условия эксперимента.

- Типичная Linux-система периода 2015–2016 годов.
- Java 8 (OpenJDK и Oracle).
- Немодифицированные Tomcat и Netty, используемые в типичной производственной среде.
- Репрезентативная (<https://github.com/Netflix-Skunkworks/WSPerfLab/tree/master/ws-impls#test-case-a>) нагрузка на веб-сервис, обращающийся к нескольким другим веб-сервисам.

В этом контексте мы получили следующие результаты:

- Netty эффективнее, чем Tomcat, в том смысле, что потребляет меньше ресурсов процессора в расчете на один запрос;

- Архитектура цикла обработки событий в Netty уменьшает миграцию потоков при высокой нагрузке, что увеличивает частоту попаданий в кэш и локальность обращений к памяти, а это, в свою очередь, увеличивает число команд, выполняемых за один такт процессора (Instructions-per-Cycle – IPC), и, следовательно, уменьшает число тактов на один запрос.
- Для Tomcat характерна более высокая задержка при высокой нагрузке вследствие архитектуры пула потоков, которая влечет за собой блокировки пула (и конкуренцию за блокировки) и миграцию потоков.

В поисках ответов

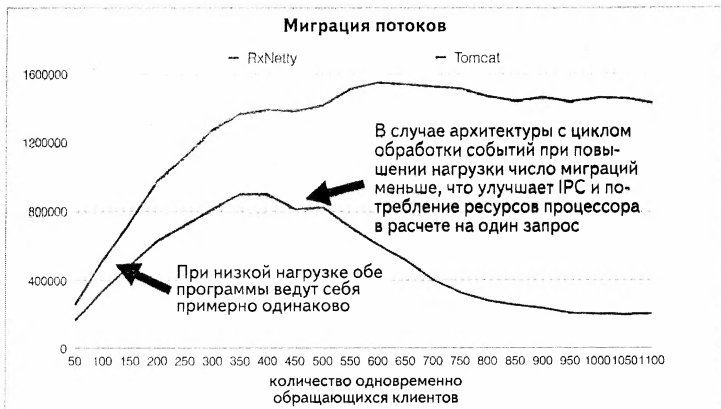
Поработав с RxJava некоторое время, я захотел получить ответ на вопрос о сравнительной эффективности блокирующего и неблокирующего ввода-вывода (обработка каждого запроса в отдельном потоке по сравнению с циклами обработки событий), но обнаружил, что получить определенные ответы очень сложно. Более того, исследуя этот вопрос, я столкнулся с противоречивыми ответами, различными мифами, теориями, мнениями и путаницей. В конечном итоге я пришел к выводу, что теоретически все существующие подходы (потоки, циклы обработки событий, волокна (fiber) и взаимодействующие последовательные процессы) должны давать одинаковую производительность (в терминах пропускной способности и задержки), поскольку в любом случае потребляются ресурсы одного и того же процессора. Но на практике любая реализация подразумевает определенные структуры данных и алгоритмы и имеет дело с аппаратными реалиями, а потому должна принимать во внимание, как работает оборудование – во-первых – и особенности реализации операционной системы исполняющей среды – во-вторых.

Сам я не мог ответить на эти вопросы, но мне повезло работать с Бренданом Греггом (Brendan Gregg), которому опыта в этом деле было не занимать (<https://www.amazon.com/Systems-Performance-Enterprise-Brendan-Gregg-ebook/dp/B00FLYU9T2#nav-subnav>). У нас с Нитешом Кантом (Nitesh Kant) была возможность на протяжении нескольких месяцев составлять профиль приложений на базе Tomcat и Netty.

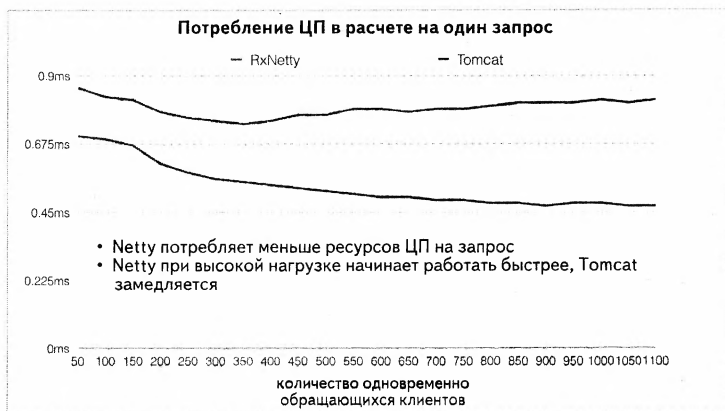
Мы специально выбрали «реальный» код – Tomcat и Netty – потому что эти программы имели непосредственное отношение к нашим производственным системам (мы уже работали с Tomcat и примеривались к Netty). Архитектура этих программ существенно различна: в одной каждый запрос обрабатывается в отдельном потоке, в другой используется цикл обработки событий.

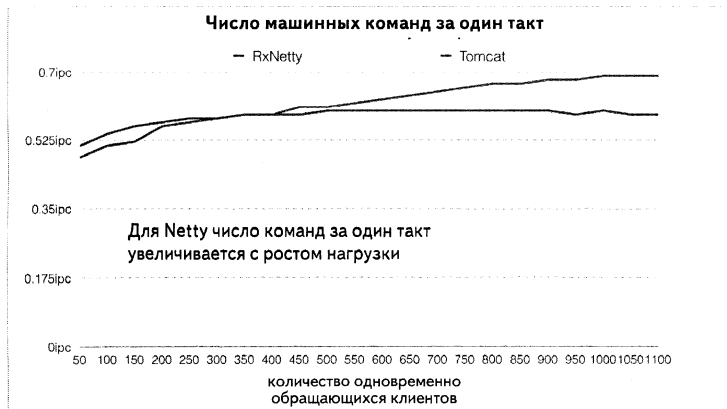
Детали нашего исследования опубликованы на GitHub по адресу https://github.com/Netflix-Skunkworks/WSPerfLab/blob/master/test-results/RxNetty_vs_Tomcat_April2015.pdf вместе с кодом тестов (<https://github.com/Netflix-Skunkworks/WSPerfLab/tree/master/ws-impls>). Презентацию результатов, озаглавленную «Applying Reactive Programming with RxJava», можно посмотреть по адресу <https://speakerdeck.com/benjchristensen/applying-reactive-programming-with-rxjava-at-goto-chicago-2015?slide=146>.

Следующий график иллюстрирует различие между архитектурами:



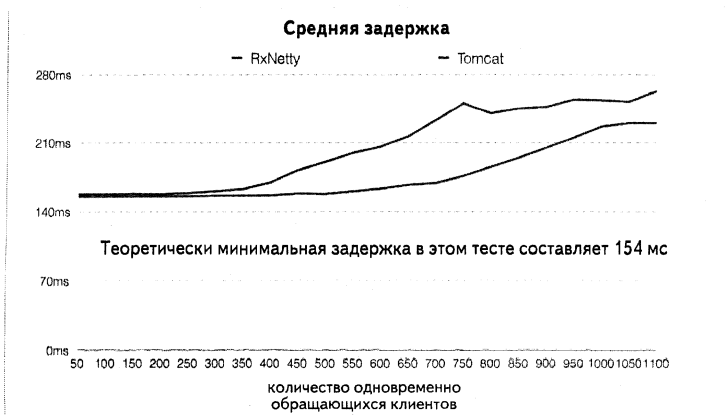
Обратите внимание, как расходятся кривые при повышении нагрузки. На этом рисунке показано количество миграций потоков. Для меня самым интересным оказался тот факт, что эффективность Netty с ростом нагрузки повышается, т. к. потоки становятся «горячими» и оказываются привязаны к одному и тому же процессорному ядру. С другой стороны, в Tomcat каждый запрос обрабатывается в отдельном потоке, поэтому он ничего не выигрывает от повышения нагрузки и миграция остается высокой, т. к. потоки планируются заново при поступлении нового запроса.





В случае Netty график потребления процессора при повышении нагрузки остается близким к горизонтальной линии и даже показывает небольшой прирост эффективности при максимальной нагрузке – в отличие от Tomcat, эффективность которого снижается.

На следующих графиках показано влияние на задержку.



Хотя средние величины не очень убедительны (в отличие от перцентилей), этот график все же показывает, что задержки в обеих программах близки при низкой нагрузке, но заметно расходятся при ее повышении. Нагрузка, при которой задержка начинает расти, для Netty выше, а влияние нагрузки на задержку не так велико.

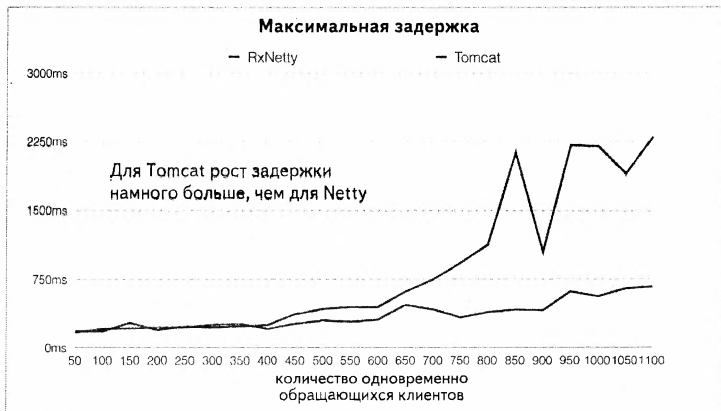
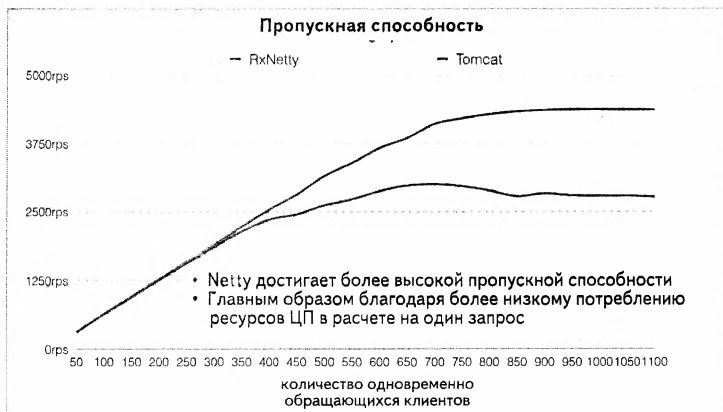


График максимальной задержки выбран для того, чтобы показать, как выбросы могут оказывать влияние на пользователей и потребление системных ресурсов. Netty реагирует на повышение нагрузки гораздо более плавно, выбросы в «худшем случае» отсутствуют.

На графике ниже показана зависимость пропускной способности от нагрузки.



Из этих результатов можно сделать два вывода. Во-первых, более низкая задержка и более высокая пропускная способность благотворно сказываются на работе пользователей и стоимости инфраструктуры. Во-вторых, архитектура с циклом обработки событий лучше адаптируется к нагрузке. По мере ее увеличе-

ния система не «разваливается», а задействует ресурсы машины до предела, производительность снижается плавно. Это очень убедительный аргумент, когда речь идет о крупномасштабных системах, которые должны выдерживать неожиданные всплески нагрузки без существенного снижения скорости реакции (<http://www.reactivemanifesto.org/>).

Я также обнаружил, что архитектура с циклом обработки событий удобнее для работы. Она не требует² настройки для получения оптимальной производительности, тогда как в случае архитектуры с отдельным потоком на каждый запрос часто приходится подстраивать размеры пулов (и, следовательно, параметры сборки мусора) под рабочую нагрузку.

Приведенное описание не претендует на исчерпывающее исследование вопроса, но мне кажется, что сам эксперимент и полученные данные – убедительные доводы за движение в сторону «реактивной» архитектуры в форме неблокирующего ввода-вывода и циклов обработки событий. Иными словами, при том оборудовании, ядре Linux и JVM, которые существовали в 2015–2016 годах, неблокирующий ввод-вывод с циклом обработки событий имеет преимущества.

Об использовании Netty в контексте RxJava мы еще будем говорить в разделе «Неблокирующий HTTP-сервер на основе Netty и RxNetty» главы 5.

Абстракция реактивности

В конечном итоге, типы и операторы RxJava – не более чем абстракция поверх императивных обратных вызовов. Однако эта абстракция кардинально изменяет стиль кодирования и дает весьма мощные средства для написания асинхронных неблокирующих программ. Требуются определенные усилия, чтобы изменить привычные подходы и освоиться с композицией функций и потоками, зато в награду вы получите очень эффективный инструмент вдобавок к знакомым объектно-ориентированному и императивному стилям программирования.

В последующих главах будет подробно описано внутреннее устройство и применение RxJava. В главе 2 объясняется, как возникают объекты `Observable` и как их потреблять. А в главе 3 вы узнаете о нескольких десятках декларативных преобразований для самых разных целей.

² За исключением, пожалуй, споров о том, сколько должно быть таких циклов: 1х, 1.5х или 2 х число процессорных ядер. Я, впрочем, не обнаружил заметных различий и обычно оставляю значение по умолчанию 1х.

Глава 2.

Реактивные расширения

В этой главе мы рассмотрим основные концепции, относящиеся к «реактивным расширениям» (Reactive Extensions) и RxJava. Мы близко познакомимся с типами `Observable<T>`, `Observer<T>` и `Subscriber<T>`, а также с несколькими полезными вспомогательными методами, которые называются *операторами*. Тип `Observable` лежит в основе API RxJava, поэтому необходимо ясно понимать, как он работает и какие реалии представляет. Из этой главы вы поймете, что такое `Observable` на самом деле, как создавать объекты этого типа и как взаимодействовать с ними. Полученные знания позволят идиоматически пользоваться реактивными API порождения и потребления в RxJava. Библиотека RxJava проектировалась с целью упростить асинхронное событийно-ориентированное программирование, но чтобы с пользой применять ее, нужно освоить некоторые базовые принципы и семантику. Поняв, как `Observable` взаимодействует с клиентским кодом, вы почувствуете, как ваши пальцы исполнятся великой силой. Прочитав эту главу, вы сможете создавать простые потоки данных, из которых можно составлять интересные комбинации и композиции.

Анатомия rx.Observable

Тип `rx.Observable<T>` представляет последовательность значений. Это именно та абстракция, которой мы будем пользоваться все время. Поскольку значения часто охватывают широкий временной диапазон, мы склонны рассуждать об объекте `observable` как о потоке событий. Оглядевшись вокруг, вы встретите много примеров потоков:

- события пользовательского интерфейса;
- байты, передаваемые по сети;
- новые заказы в Интернет-магазине;
- посты на сайтах социальных сетей.

Если хотите сравнить `Observable<T>` с чем-то более знакомым, то самой близкой абстракцией будет `Iterable<T>`. Как и объект `Iterator<T>`, полученный от `Iterable<T>`, поток `Observable<T>` может содержать от нуля до бесконечного множества значений типа `T`. `Iterator` очень эффективно генерирует бесконечные последовательности, например, последовательность натуральных чисел:

```
class NaturalNumbersIterator implements Iterator<BigInteger> {  
  
    private BigInteger current = BigInteger.ZERO;  
  
    public boolean hasNext() {  
        return true;  
    }  
  
    @Override  
    public BigInteger next() {  
        current = current.add(BigInteger.ONE);  
        return current;  
    }  
}
```

Есть и еще одна общая черта: `Iterator` может сам известить клиента, что больше нечего порождать (подробнее об этом ниже). Но на этом сходство и заканчивается. Тип `Observable` устроен по принципу проталкивания, т. е. он сам решает, когда порождать значения. С другой стороны, `Iterator` сидит себе тихо, пока кто-нибудь не попросит у него следующий элемент, вызвав метод `next()`. Традиционно такое поведения для `Observable` не поддерживалось – в какой-то момент клиент должен *подписаться* на `Observable`, после чего будет получать уведомления всякий раз, как `Observable` захочет породить значение. Это может случиться в любой момент: как немедленно, так и никогда. Далее, в главе 6, мы рассмотрим механизм проталкивания, который дает подписчикам средства для управления темпом работы `Observable` при определенных условиях.

Кроме того, `Observable` может порождать произвольное число событий. На первый взгляд, это очень похоже на классический паттерн Наблюдатель, известный также под названием издатель-подписчик (подробнее эта тема освещена в книге Эриха Гаммы и Ричарда Хелма «Приемы объектно-ориентированного проектирования. Паттерны проектирования»). Но точно так же, как за объектом `Iterator` необязательно стоит какая-то коллекция (см. `NaturalNumbersIterator`), так и `Observable` не обязан представлять реальный поток событий. Самое время познаться с несколькими примерами `Observable`:

```
Observable<Tweet> tweets
```

Твиты – пожалуй, самый очевидный пример потока событий. Нет сомнений, что обновления состояния на сайте любой социальной сети случаются постоянно и, разумеется, могут быть представлены в виде потока событий. Кроме того, в отличие от `Iterator`, мы не можем вытянуть данные самостоятельно в удобный для нас момент. `Observable` проталкивает данные по мере их поступления.

```
Observable<Double> temperature
```

Объект `temperature` похож на предыдущий; он получает значения температуры от некоторого устройства и проталкивает их подписчикам. Как `tweets`, так и `temperature` – примеры бесконечных потоков будущих событий.

Observable<Customer> customers

Что именно представляет тип Observable<Customer>, зависит от контекста. Скорее всего, он возвращает список заказчиков, возможно, в ответ на запрос к базе данных. В списке может быть нуль, несколько или даже тысячи записей, быть может, загружаемых лениво. Но этот тип может представлять также поток входов пользователей (объектов типа Customer) в систему. Клиентская модель программирования не зависит от того, как реализован тип Observable<Customer>.

Observable<HttpResponse> response

С другой стороны, тип Observable<HttpResponse>, наверное, представляет всего одно событие (значение), после получения которого поток завершается. Это значение появится в будущем и будет доставлено клиентской программе. Чтобы прочитать ответ, мы должны подписаться.

Observable<Void> completionCallback

А вот и старый добрый Observable<Void>. Технически объект Observable может завершиться, не породив ни одного значения. В таком случае нас не интересует тип значений, проталкиваемых Observable, потому что они никогда не встречаются.

На самом деле, Observable<T> может порождать события трех типов:

- значения типа T в соответствии с объявлением;
- событие завершения;
- событие ошибки.

В спецификации реактивных расширений ясно сказано, что любой объект Observable может порождать произвольное число значений, за которыми, возможно, следует событие завершения или ошибки (но не то и другое сразу). В руководстве «Rx Design Guidelines» это правило строго формулируется так¹: `OnNext*(OnCompleted | OnError)?`, где `OnNext` обозначает новое событие. Интересно, что любая комбинация членов в этом правиле, напоминающем регулярное выражение, допустима и полезна.

OnNext OnCompleted

Observable порождает одно значение и корректно завершается. Так бывает, когда Observable представляет запрос к внешней системе, и мы ожидаем получить единственный ответ.

OnNext+ OnCompleted

Observable порождает несколько событий, затем завершается. Так можно представить чтение списка из базы данных, когда каждая запись считается одним значением. Другой пример – наблюдение за ходом длительного процесса, который в конце концов завершается.

¹ <http://go.microsoft.com/fwlink/?LinkID=205219>

OnNext+

Бесконечный список событий, например, комментариев на сайте социальной сети или обновлений состояния некоторого компонента (скажем, перемещений мыши или ring-запросов). Поток бесконечен и должен потребляться на лету.

OnCompleted ИЛИ OnError

Такой `Observable` сигнализирует о завершении – нормальном или аномальном – и только. Событие `OnError` дополнительно содержит объект `Throwable`, описывающий причину завершения потока. Ошибки передаются в виде события, а не с помощью стандартного предложения `throw`.

OnNext+ OnError

Поток может успешно породить одно или несколько событий, а затем завершиться с ошибкой. Обычно это означает, что поток должен был бы быть бесконечным, но из-за какой-то фатальной ошибки прервался. Представьте себе последовательность сетевых пакетов: доставка осуществлялась в течение нескольких часов, а затем прервалась из-за потери связности сети.

Уведомление `OnError` весьма интересно. Из-за асинхронной природы объектов `Observable` просто возбуждать исключения не имеет смысла. Мы должны передать ошибку тому, кто в ней заинтересован, возможно, через границы потоков и спустя некоторое время. `OnError` – специальный тип события, который инкапсулирует исключения в функциональном стиле. Подробнее об исключениях см. раздел «Обработка ошибок» главы 7.

Наконец, можно реализовать `Observable` так, что не будет порождаться вообще ни одного события, даже завершения и ошибки. Такой `Observable` полезен для тестирования, например, таймаутов.

Подписка на уведомления от объекта `Observable`

Экземпляр типа `Observable` не порождает никаких событий, пока кто-то не выразит заинтересованность в их получении. Чтобы начать наблюдение за `Observable`, нужно вызвать один из методов семейства `subscribe()`:

```
Observable<Tweet> tweets = //...
```

```
tweets.subscribe((Tweet tweet) -> System.out.println(tweet));
```

Здесь мы подписываемся на объект `tweets` типа `Observable`, зарегистрировав функцию обратного вызова. Эта функция будет вызываться всякий раз, как поток `tweets` решит протолкнуть событие подписчику. Контракт RxJava гарантирует, что функция обратного вызова не будет вызвана одновременно из двух или более потоков, хотя события могут порождаться в разных потоках. Существует несколько

специализированных перегруженных вариантов `subscribe()`. Мы уже говорили, что `Observable` не возбуждает исключений. Вместо этого ошибки рассматриваются как специальный вид уведомления (событие), распространяемого `Observable`. Следовательно, не нужно заключать вызов `subscribe()` в блок `try-catch` для перехвата исключений, а надо предоставить отдельную функцию обратного вызова:

```
tweets.subscribe(  
    (Tweet tweet) -> { System.out.println(tweet); },  
    (Throwable t) -> { t.printStackTrace(); }  
);
```

Второй аргумент `subscribe()` необязателен. Он уведомляет об исключениях, которые могут произойти во время порождения элементов. Гарантируется, что после исключения не будет доставлено ни одного объекта `tweet`. Почти всегда имеет смысл подписываться не только на обычные элементы, но и на исключения, даже если вы их не ожидаете. Исключения – полноправные граждане в мире `Observable`. Возбуждаемые исключения могут быстро распространяться вверх по стеку, вызывая массу побочных эффектов от несогласованных структур данных до невыполненных транзакций. Вообще говоря, это хорошая идея, но часто исключения не являются фатальными. Поэтому устойчивая система должна предвидеть и методично обрабатывать исключения. Именно поэтому `Observable` моделирует их явно.

Третья необязательная функция обратного вызова позволяет обнаруживать завершение потока:

```
tweets.subscribe(  
    (Tweet tweet) -> { System.out.println(tweet); },  
    (Throwable t) -> { t.printStackTrace(); },  
    () -> {this.noMore();}  
);
```

Напомним, что библиотеке `RxJava` безразлично, сколько элементов порождено, и она не знает, когда останавливаться. Поток может быть бесконечным, а может завершиться сразу после подписки, поэтому только объект `Subscriber` может решить, хочет ли он получать уведомление о завершении. Если вы априори знаете, что поток бесконечен, то, очевидно, подписываться на такое уведомление бессмысленно. С другой стороны, иногда нас интересует именно событие завершения потока. Представьте, к примеру, объект типа `Observable<Progress>`, который следит за длительными процессами. Клиенту может быть и неинтересно, как протекает процесс, но он хочет знать, когда процесс остановится.

Отметим попутно, что в `Java 8` часто можно использовать ссылки на методы вместо лямбда-выражений для большей понятности:

```
tweets.subscribe(  
    System.out::println,  
    Throwable::printStackTrace,  
    this::noMore);
```

Получение всех уведомлений с помощью типа `Observer<T>`

Коль скоро задание всех трех аргументов метода `subscribe()` так полезно, то почему бы не создать простую обертку, включающую все три функции обратного вызова. Именно для этого и предназначен тип `Observer<T>`. Он получает все возможные уведомления от `Observable<T>`. Вот как регистрируется объект `Observer<T>`:

```
Observer<Tweet> observer = new Observer<Tweet>() {  
    @Override  
    public void onNext(Tweet tweet) {  
        System.out.println(tweet);  
    }  
  
    @Override  
    public void onError(Throwable e) {  
        e.printStackTrace();  
    }  
  
    @Override  
    public void onCompleted() {  
        noMore();  
    }  
};  
  
//...  
  
tweets.subscribe(observer);
```

На самом деле, `Observer<T>` – основная абстракция прослушивания в RxJava. Но если такого уровня контроля недостаточно, то к нашим услугам еще более мощный тип `Subscriber` (абстрактная реализация `Observer`).

Управление прослушивателями с помощью типов `Subscription` и `Subscriber<T>`

У одного объекта `Observable` вполне может быть много подписчиков. Как в паттерне издатель-подписчик, один издатель может отправлять события многим потребителям. В RxJava тип `Observable<T>` – это просто структура данных, которая может существовать очень недолго или в течение многих дней, пока приложение работает. То же относится и к подписчикам. Мы можем подписаться на `Observable`, потребить какое-то количество событий, а остальные проигнорировать. Или наоборот: потреблять события все время, пока объект `Observable` жив, быть может, много часов или дней.

Представим себе объект `Observer`, который заранее знает, сколько событий хочет получить или когда прекращать получение. Например, мы подписались на из-

менение курса акций, но если цена падает ниже 1 доллара, то дальнейшее прослушивание нам не интересно. Понятно, что `Observer` должен иметь возможность не только подписаться, но и отписаться в любой момент. Для этого есть два механизма: `Subscription` и `Subscriber`. Сначала рассмотрим первый. Мы еще не говорили о том, что же возвращает метод `subscribe()`:

```
Subscription subscription =  
    tweets.subscribe(System.out::println);
```

```
//...
```

```
subscription.unsubscribe();
```

`Subscription` — это описатель, который позволяет отменить подписку, вызвав оператор `unsubscribe()`. Он также позволяет опросить состояние подписки методом `isUnsubscribed()`. Очень важно отписываться от объекта `Observable<T>`, если необходимость в получении событий отпала; это позволит избежать утечек памяти и уменьшить нагрузку на систему. Иногда мы подписываемся на `Observable` и потребляем все события, даже если их поток бесконечен. Но бывают случаи, когда подписчики приходят и уходят, а `Observable` продолжает порождать события вечно.

Существует еще один способ отписаться, на этот раз изнутри прослушивателя. Мы знаем, что можем использовать тип `Subscription` для управления подпиской извне `Observer` или функции обратного вызова. С другой стороны, тип `Subscriber<T>` реализует оба интерфейса: `Observer<T>` и `Subscription`. Следовательно, его можно использовать как для потребления уведомлений (данных, событий завершения и ошибки), так и для управления подпиской. В примере ниже мы подписываемся на все события, а затем сам подписчик решает отказаться от получения уведомлений, руководствуясь известным ему критерием. Обычно это делается с помощью оператора `takeUntil()`, но сейчас мы опишемся вручную:

```
Subscriber<Tweet> subscriber = new Subscriber<Tweet>() {  
    @Override  
    public void onNext(Tweet tweet) {  
        if (tweet.getText().contains("Java")) {  
            unsubscribe();  
        }  
    }  
  
    @Override  
    public void onComplete() {}  
  
    @Override  
    public void onError(Throwable e) {  
        e.printStackTrace();  
    }  
};  
tweets.subscribe(subscriber);
```

Когда `Subscriber` решает, что больше не хочет получать уведомления, он может отписать себя. В качестве упражнения реализуйте класс `Subscriber`, который полу-

чает только первые n событий, а затем отписывается. Класс `Subscriber` способен не только на это, но пока просто запомните, что он умеет отписываться от `Observable`.

Создание объектов `Observable`

Мы начали с подписки на объект `Observable` для получения проталкиваемых им событий. И это не случайно. При работе с RxJava вы, как правило, будете взаимодействовать с уже существующими объектами `Observable`, выполняя такие операции, как комбинирование, фильтрация и обертывание одного объекта другим. Однако, если внешний API не раскрывает никаких объектов `Observable`, то неплохо бы понять, откуда они берутся, как создать поток и обработать подписку на него. Прежде всего, существует несколько фабричных методов для создания фиксированных константных объектов `Observable`. Они полезны, если вы хотите единообразно применять RxJava во всем своем коде или если порождение значений обходится дешево, а сами значения известны заранее.

```
Observable.just(value)
```

Создать экземпляр `Observable`, который отдает ровно одно значение `value` всем будущим подписчикам, а затем завершается. Перегруженные варианты оператора `just()` могут принимать от двух до девяти отдаваемых значений.

```
Observable.from(values)
```

Аналогично `just()`, но принимает аргумент типа `Iterable<T>` или `T[]` и, следовательно, создает объект `Observable<T>`, порождающий столько значений, сколько есть элементов в коллекции `values`. Перегруженный вариант принимает аргумент типа `Future<T>` и порождает событие, когда объект `Future` завершается.

```
Observable.range(from, n)
```

Порождает n целых чисел, начиная с `from`. Например, метод `range(5, 3)` порождает числа 5, 6, 7, а затем нормально завершается. Любой подписчик получит один и тот же набор чисел.

```
Observable.empty()
```

Завершается сразу после подписки, не порождая никаких значений.

```
Observable.never()
```

Такой `Observable` не порождает вообще никаких уведомлений: ни значений, ни событий завершения или ошибки. Он может быть полезен для тестирования.

```
Observable.error()
```

Немедленно отправляет каждому подписчику уведомление `onError()`. Никакие значения не порождаются и, согласно контракту, уведомление `onCompleted()` тоже не может быть отправлено.

Подробнее о методе `Observable.create()`

Фабрики `empty()`, `never()` и `error()`, на первый взгляд, не особенно полезны, однако они очень удобны для композиции реальных объектов `Observable`. Интересно, что хотя в центре RxJava лежит идея асинхронной обработки потока событий, эти методы по умолчанию работают в потоке клиента. Взгляните на следующий код:

```
private static void log(Object msg) {
    System.out.println(
        Thread.currentThread().getName() + ": " + msg);
}

//...

log("До");
Observable
    .range(5, 3)
    .subscribe(i -> {
        log(i);
    });
log("После");
```

Нас интересует, в каком потоке выполняется каждый вызов `log`:

```
main: До
main: 5
main: 6
main: 7
main: После
```

Порядок вызовов `print` тоже важен. Не удивительно, что сообщения До и После печатаются в главном потоке клиента (`main`). Заметим, однако, что подписка также имела место в потоке клиента и что вызов `subscribe()` заблокировал этот поток на время до получения всех событий. Если не поступило соответствующего требования от какого-то оператора, то RxJava по умолчанию не использует пул потоков для выполнения вашего кода. Чтобы разобраться в этом поведении, изучим низкоуровневый оператор `create()`, служащий для создания объектов `Observable`:

```
Observable<Integer> ints = Observable
    .create(new Observable.OnSubscribe<Integer>() {
        @Override
        public void call(Subscriber<? super Integer> subscriber) {
            log("Create");
            subscriber.onNext(5);
            subscriber.onNext(6);
            subscriber.onNext(7);
            subscriber.onCompleted();
            log("Завершился");
        }
    });

log("Начало");
ints.subscribe(i -> log("Элемент: " + i));
log("Конец");
```

Мы сознательно включили в этот код подробное протоколирование. Вот что он печатает (включая имя потока, в котором выполнялась строка):

```
main: Начало
main: Create
main: Элемент: 5
main: Элемент: 6
main: Элемент: 7
main: Завершен
main: Конец
```

Чтобы понять, как работает `Observable.create()` и как RxJava относится к конкурентности, проанализируем выполнение этого кода шаг за шагом. Сначала мы создаем объект `ints` типа `Observable`, предоставив реализацию интерфейса `onSubscribe` методу `create()` (впоследствии мы почти всегда будем заменять такую конструкцию более простым лямбда-выражением). В этот момент еще не произошло ничего, кроме создания экземпляра `Observable`, поэтому первой напечатана строка `main: Начало`. Объект `Observable` по умолчанию откладывает порождение событий, т. е. оно начинается только после подписки со стороны какого-нибудь клиента, поэтому код, переданный методу `create()`, пока не выполняется. Далее мы подписываемся методом `ints.subscribe(...)`, что заставляет `Observable` начать порождение элементов. Это справедливо для так называемых *холодных* потоков. С другой стороны, существуют *горячие* потоки, которые порождают события, даже если на них никто не подписан. Это важное различие будет разъяснено в разделе «Горячие и холодные объекты `Observable`» ниже в этой главе.



Лямбда-выражение, получающее порожденные элементы (`i -> log("Элемент: " + i)`), внутри обертывается объектом `Subscriber<Integer>`. Этот объект почти напрямую передается в виде аргумента той функции, которая была задана при вызове `create()`. Поэтому всякий раз, как вы производите подписку на `Observable`, создается новый экземпляр `Subscriber` и передается вашему методу `create()`. Вызов `onNext()` или других методов `Subscriber` внутри `create()` приводит к косвенному вызову вашего же объекта `Subscriber`.

Метод `Observable.create()` настолько гибок, что позволяет имитировать поведение всех вышеупомянутых фабричных методов. Например, метод `Observable.just(x)`, создающий объект, который порождает единственное значение `x` и сразу после этого завершается, можно было бы написать так:

```
static <T> Observable<T> just(T x) {
    return Observable.create(subscriber -> {
        subscriber.onNext(x);
        subscriber.onCompleted();
    });
}
```

В качестве упражнения попробуйте реализовать методы `never()`, `empty()` и `range()`, пользуясь только `create()`.

Управление несколькими подписчиками

Порождение значений не начинается, пока нет подписчиков. Но при каждом обращении к `subscribe()` вызывается наш обработчик подписки внутри метода `create()`. Это нельзя назвать ни преимуществом, ни недостатком; это просто факт, о котором следует помнить. Иногда очень удобно, что с каждым подписчиком связан отдельный вызов обработчика. Например, при вызове `Observable.just(42)` значение 42 будет отправлено каждому подписчику, а не только первому. С другой стороны, если внутри `create()` находится запрос к базе данных или какое-то длительное вычисление, то хорошо бы выполнять его только один раз для всех подписчиков.

Чтобы лучше понять, как работает подписка, рассмотрим следующий пример, в которой мы дважды подписываемся на один и тот же объект `Observable`:

```
Observable<Integer> ints =
    Observable.create(subscriber -> {
        log("Create");
        subscriber.onNext(42);
        subscriber.onCompleted();
    })
    log("Начало");
ints.subscribe(i -> log("Элемент A: " + i));
ints.subscribe(i -> log("Элемент B: " + i));
log("Конец");
```

Как вы думаете, что произойдет? Напомним, что при каждой подписке на объект `Observable`, созданный фабричным методом `create()`, лямбда-выражение, переданное в качестве аргумента, по умолчанию выполняется в том же потоке, где была произведена подписка:

```
main: Начало
main: Create
main: Элемент A: 42
main: Create
main: Элемент B: 42
main: Конец
```

Если мы хотим не вызывать `create()` для каждого подписчика, а повторно использовать уже вычисленные события, то можем воспользоваться оператором `cache()`:

```
Observable<Integer> ints =
    Observable.<Integer>create(subscriber -> {
        //...
    })
    .cache();
```

`cache()` – первый встретившийся нам оператор. Операторы обертывают существующие объекты `Observable`, наделяя их новыми возможностями – обычно путем перехвата подписки. Конкретно оператор `cache()` располагается между методом `subscribe()` и нашим объектом `Observable`. При появлении первого подписчика `cache()` делегирует обработку подписки обернутому им экземпляру `Observable` и переправляет все события (данные, уведомления о завершении и ошибке) подписчику. Но вместе с тем он запоминает все отправленные события. Если появится еще один подписчик и захочет получить события, то `cache()` не делегирует обращение обернутому `Observable`, а сам отправляет кэшированные значения. При включенном кэшировании трасса вызовов выглядит иначе:

```
main: Начало
main: Create
main: Элемент A: 42
main: Элемент B: 42
main: Конец
```

Конечно, нужно помнить, что кэширование бесконечного потока – прямая дорога к катастрофе, именуемой `OutOfMemoryError`. Но об этом мы будем говорить гораздо позже, в разделе «Потребление памяти и утечки» главы 8.

Бесконечные потоки

О бесконечных структурах данных стоит поговорить особо. Память компьютера конечна, поэтому, на первый взгляд, бесконечный список или поток – вещь невозможная. Однако `RxJava` позволяет порождать и потреблять события на лету. Да и традиционную очередь можно рассматривать как бесконечный источник значений, хотя они не хранятся в памяти одновременно. Но как реализовать такой бесконечный поток с помощью метода `create()`? Для примера построим объект `Observable`, порождающий все натуральные числа:

```
// НЕПРАВИЛЬНО! Не делайте так
Observable<BigInteger> naturalNumbers = Observable.create(
    subscriber -> {
        BigInteger i = ZERO;
        while (true) { // не делайте так!
            subscriber.onNext(i);
            i = i.add(ONE);
        }
    });
naturalNumbers.subscribe(x -> log(x));
```

Наличие конструкции `while(true)` всегда должно вызывать подозрения. На первый взгляд выглядит нормально, но эта реализация некорректна – и вы очень скоро в этом убедитесь. Дело не в том, что цикл бесконечный – бесконечные потоки `Observable` вполне допустимы и весьма полезны. Если, конечно, реализованы правильно. При вызове `subscribe()` лямбда-выражение внутри `create()` вычисляется в контексте клиентского потока. А поскольку это вычисление никогда не за-

вершается, то `subscribe()` навечно блокирует работу программы. Но, скажете вы, ведь подписка должна быть асинхронной, а не выполняться в потоке клиента. Это правильное замечание, поэтому познакомимся с понятием явной конкурентности:

```
Observable<BigInteger> naturalNumbers = Observable.create(
    subscriber -> {
        Runnable r = () -> {
            BigInteger i = ZERO;
            while (!subscriber.isUnsubscribed()) {
                subscriber.onNext(i);
                i = i.add(ONE);
            }
        };
        new Thread(r).start();
    });
```

Вместо того чтобы выполнять блокирующий цикл в потоке клиента, мы создаем отдельный поток и порождаем события в нем. Теперь метод `subscribe()` не блокирует клиентский поток, поскольку не делает ничего, кроме запуска нового потока. А все вызовы `x -> log(x)` производятся в этом потоке. Теперь допустим, что нас интересуют не все натуральные числа (их все-таки многовато), а только несколько первых. Мы уже знаем, как остановить поток уведомлений от `Observable` — отписаться:

```
Subscription subscription = naturalNumbers.subscribe(x -> log(x));
// спустя некоторое время...
subscription.unsubscribe();
```

Если вы привыкли обращать внимание на мелкие детали, то, наверное, заметили, что подозрительный цикл `while(true)` заменен таким:

```
while (!subscriber.isUnsubscribed()) {
```

На каждой итерации мы должны удостовериться, что кто-то нас слушает. Если подписчик прекратил прослушивание, то проверка условия `subscriber.isUnsubscribed()` покажет, что мы можем без опаски завершить поток и вместе с ним работу `Runnable`, т. е. остановить поток. Очевидно, что для каждого подписчика имеется отдельный поток и отдельный цикл, так что после отписки одного остальные продолжат получать свои независимые потоки событий. Хотя самостоятельное создание потоков — не лучшее проектное решение и в RxJava есть куда более подходящие декларативные средства управления конкурентностью, этот пример показывает, как правильно обрабатывать события подписки.

Флаг `isUnsubscribed()` рекомендуется проверять как можно чаще, чтобы не посылать события после того, как подписчик прекратил их получать. Мало того что порождение событий обходится дорого, так еще и энергично отправлять их в никуда совершенно бессмысленно. Ничего принципиально порочного в запуске потоков из `create()` нет, но эта техника чревата ошибками и плохо масштабируется. В разделе «Многопоточность в RxJava» главы 4 мы рассмотрим декларативную

конкурентность и пользовательскую диспетчеризацию, что позволит писать конкурентный код без прямого взаимодействия с потоками.

Обработка отписки непосредственно перед попыткой отправить событие – вещь хорошая при условии, что события отправляются достаточно часто. Но представьте, что события возникают очень редко. `Observable` может определить, что подписчик отписался, только когда попытается отправить ему событие. Для иллюстрации рассмотрим полезный фабричный метод `delayed(x)`, создающий объект `Observable`, который порождает значение `x` с задержкой десять секунд. Мы уже знаем, что нужно использовать дополнительный поток, хотя эта техника программирования и не самая лучшая:

```
static <T> Observable<T> delayed(T x) {
    return Observable.create(
        subscriber -> {
            Runnable r = () -> {
                sleep(10, SECONDS);
                if (!subscriber.isUnsubscribed()) {
                    subscriber.onNext(x);
                    subscriber.onCompleted();
                }
            };
            new Thread(r).start();
        });
}

static void sleep(int timeout, TimeUnit unit) {
    try {
        unit.sleep(timeout);
    } catch (InterruptedException ignored) {
        // сознательно игнорируем
    }
}
```

При наивной реализации мы запускаем новый поток и спим 10 секунд. Было бы надежнее хотя бы воспользоваться классом `java.util.concurrent.ScheduledExecutorService`, но этот пример преследует только педагогические цели. Спустя 10 секунд мы проверяем, слушает нас кто-нибудь или нет, и, если слушает, то порождаем единственный элемент и завершаемся. Но что, если подписчик решит отписаться через секунду после подписки, задолго до того, как будет порождено событие? Тогда фоновый поток проспит оставшиеся девять секунд только для того, чтобы, проснувшись, обнаружить, что подписчика уже нет. И вот это нас как раз и беспокоит: удерживать ресурс лишние девять секунд как-то расточительно. Представьте, что ресурс – это дорогостоящее подключение к какому-то источнику данных, за которое мы платим посекундно, и что события в нем возникают очень редко. Ждать несколько секунд или даже минут и в итоге осознать, что уже давно никто не подписан и следовало бы разорвать соединение, мягко говоря, неоптимально. По счастью, мы можем получить извещение прямо в тот момент, когда экземпляр `subscriber` отпишется, а, значит, освободить ресурсы как можно раньше, а не тогда, когда придет следующее сообщение.


```
static <T> Observable<T> delayed(T x) {
    return Observable.create(
        subscriber -> {
            Runnable r = () -> { /* ... */ };
            final Thread thread = new Thread(r);
            thread.start();
            subscriber.add(Subscriptions.create(thread::interrupt));
        });
}
```

Важна последняя строка, а все остальное не изменилось. Фоновый поток уже работает, а точнее заснул на 10 секунд. Но сразу после запуска потока мы просим подписчика известить нас с помощью функции обратного вызова, когда он захочет отписаться, а также регистрируем подписчика, вызвав метод `Subscriber.add()`. У этой функции обратного вызова всего одна задача: прервать поток. Метод `Thread.interrupt()` возбуждает исключение `InterruptedException` во время выполнения `sleep()`, прерывая десятисекундную паузу. Это исключение игнорируется, и программа продолжает выполнение с предложения после `sleep()`. Но в этой точке `subscriber.isUnsubscribed()` возвращает `true`, так что событие не порождается. Поток сразу же останавливается, и никакие ресурсы не расходуются впустую. Такую технику можно использовать для любой очистки. Впрочем, если поток порождает стабильный поток частых событий, то можно обойтись и без явного обратного вызова.

Существует еще одна причина не создавать потоки явно внутри `create()`. В разделе 4.2 «Предполагайте, что экземпляры наблюдателя вызываются строго последовательно» руководства по проектированию с использованием Rx (Rx Design Guidelines) есть требование не допускать одновременного получения подписчиком нескольких уведомлений. Это требование легко нарушить, если потоки используются явно. Такое поведение схоже с поведением акторов, например в комплекте инструментов Akka (<http://akka.io/>), где каждый актор может обрабатывать не более одного сообщения в каждый момент времени. Благодаря этому предположению код `Observer` можно писать так, будто наблюдатели синхронизированы, т. е. не может быть одновременных обращений из разных потоков. Любая реализация `Observable` обязана соблюдать этот контракт. Памятуя об этом, рассмотрим следующий код, который, пренебрегая идиоматичностью, пытается параллелизировать загрузку нескольких порций данных `Data`:

```
Observable<Data> loadAll(Collection<Integer> ids) {
    return Observable.create(subscriber -> {
        ExecutorService pool = Executors.newFixedThreadPool(10);
        AtomicInteger countDown = new AtomicInteger(ids.size());
        // ОПАСНОСТЬ, нарушение контракта Rx. Не делайте так!
        ids.forEach(id -> pool.submit(() -> {
            final Data data = load(id);
            subscriber.onNext(data);
            if (countDown.decrementAndGet() == 0) {
                pool.shutdownNow();
                subscriber.onCompleted();
            }
        }));
    });
}
```

```

    }
  }));
}

```

Этот код не только неоправданно сложен, но и нарушает некоторые принципы Rx. Точнее, он допускает вызов метода `onNext()` объекта `subscriber` одновременно из нескольких потоков. Сложности можно избежать, просто воспользовавшись идиоматичными операторами RxJava, например `merge()` и `flatMap()`, но это мы отложим до раздела «Обращение с несколькими объектами Observable, как с одним, с помощью `merge()`» главы 3. Есть и хорошая новость: даже если кто-то некорректно реализовал интерфейс `Observable`, мы сможем это исправить, применив оператор `serialize()`, например: `loadAll(...).serialize()`. Этот оператор гарантирует, что события сериализованы и упорядочены. Кроме того, он отвергает любые события, поступившие после завершения или ошибки.

Последний аспект создания `Observable`, оставшийся не рассмотренным, — распространение ошибок. До сих пор мы видели, что объект типа `Observer<T>` может принимать значения типа `T`, за которыми, возможно, следует событие завершения или ошибки. Но как протолкнуть ошибку всем подписчикам? Рекомендуемая техника — заключать целые выражения внутри `create()` в блок `try-catch`. Объекты `Throwable` следует передавать в потоке событий, а не протоколировать и не возбуждать повторно. Вот так:

```

Observable<Data> rxLoad(int id) {
    return Observable.create(subscriber -> {
        try {
            subscriber.onNext(load(id));
            subscriber.onCompleted();
        } catch (Exception e) {
            subscriber.onError(e);
        }
    });
}

```

Этот дополнительный блок `try-catch` необходим для распространения возможного исключения, возбужденного, к примеру, в методе `load(id)`. В противном случае RxJava сделает все, что может, — по крайней мере, выведет информацию об исключении в стандартный вывод, но если мы хотим строить надежные потоки, то должны рассматривать исключения как полноправных граждан, а не как дурацкие языковые конструкции, которые никто толком не понимает.

Паттерн завершения `Observable` с одним значением и обертывания его блоком `try-catch` настолько часто встречается, что для него предусмотрен специальный оператор `fromCallable()`:

```

Observable<Data> rxLoad(int id) {
    return Observable.fromCallable(() -> load(id));
}

```

Он семантически эквивалентен `create()`, но гораздо короче и дает некоторые дополнительные преимущества, с которым мы познакомимся ниже.

Хронометраж: операторы `timer()` и `interval()`

Мы посвятили несколько страниц изучению объектов `Observable`, которые самостоятельно создают потоки, хотя такая практика в RxJava не поощряется. В следующих главах мы изучим диспетчеры, но сначала рассмотрим два очень полезных оператора, в реализации которых используются потоки: `timer()` и `interval()`. Первый создает `Observable`, который порождает нулевое значение типа `long` по истечении указанной задержки, а затем завершается:

```
Observable
    .timer(1, TimeUnit.SECONDS)
    .subscribe((Long zero) -> log(zero));
```

Как ни странно, оператор `timer()` чрезвычайно полезен. По существу, это асинхронный эквивалент метода `Thread.sleep()`. Вместо того чтобы блокировать текущий поток, мы создаем объект `Observable` и подписываемся на него. Эта техника обретет гораздо больший смысл, когда мы узнаем, как организовывать сложные вычисления с помощью композиции простых `Observable`. Фиксированное значение 0 (переменной `zero`) – не более чем соглашение, не имеющее особого смысла. Однако смысл появляется при рассмотрении оператора `interval()`, который генерирует последовательность чисел типа `long`, начинающуюся с нуля, с фиксированной задержкой между соседними элементами:

```
Observable
    .interval(1_000_000 / 60, MICROSECONDS)
    .subscribe((Long i) -> log(i));
```

В отличие от оператора `range()`, `Observable.interval()` ждет заданное время, прежде чем породить очередной элемент, в т. ч. первый. В нашем примере задержка равна примерно 16 666 мкс, т. е. приблизительно 60 Гц – типичная частота кадров в анимации. Это не случайное совпадение: `interval()` иногда применяется для управления анимацией или другими процессами, которые должны протекать с заданной частотой. В каком-то смысле `interval()` похож на метод `scheduleAtFixedRate()` из класса `ScheduledExecutorService`. Вы наверняка сможете придумать много ситуаций, где подойдет `interval()`, например: опрос данных, обновление пользовательского интерфейса или моделирование хода времени в имитационных моделях.

Горячие и холодные объекты `Observable`

Получив экземпляр `Observable`, важно понять, является поток *горячим* или *холодным*. API и семантика в обоих случаях одинаковы, но способ использования зависит от типа. Холодный объект `Observable` ленивый, он ни при каких обстоятельствах не начинает порождать события, пока кто-то не выразит в них заинтере-

сованность. Если наблюдателей нет, то `Observable` – просто статическая структура данных. Отсюда также следует, что каждый подписчик получает собственную копию потока, потому что события не только порождаются лениво, но и вряд ли как-нибудь кэшируются. Холодные объекты `Observable` обычно появляются на свет в результате выполнения `Observable.create()`, идиоматика которого состоит в том, чтобы отложить всякую работу до появления реального прослушивателя. Поэтому холодный `Observable` в какой-то мере зависит от подписчиков. Помимо метода `create()`, такие объекты создают операторы `Observable.just()`, `from()` и `range()`. Подписка на холодный `Observable` часто влечет за собой побочный эффект внутри метода `create()`, например отправку запроса базе данных или открытие соединения.

Горячие объекты `Observable` ведут себя иначе. Возможно, полученный вами экземпляр уже порождает события вне зависимости от того, сколько у него подписчиков. Такой объект отправляет события, даже если их никто не ожидает и они попросту теряются. Если над холодными `Observable` вы обычно имеете полный контроль, то горячие вообще не зависят от потребителей. Появившегося подписчика можно уподобить врезке в телефонный кабель²: он подслушивает все, что проходит мимо. Наличие объекта `Subscriber` не изменяет поведение `Observable`; они абсолютно независимы друг от друга.

Как ни странно, объект, создаваемый методом `Observable.interval()`, горячим не является. Можно было бы подумать, что он просто порождает тики таймера с определенным интервалом независимо от окружения, но на самом деле события таймера порождаются, только если кто-то подписался, и при этом каждый подписчик получает собственный поток. А это и есть определение холодного `Observable`.

Горячие `Observable` обычно возникают, когда мы никак не можем контролировать источник событий. Примерами могут служить перемещения мыши, ввод с клавиатуры или нажатия на кнопку мыши. До сих пор мы ни словом не обмолвились о пользовательском интерфейсе, но вообще-то RxJava отлично подходит для его реализации. Эту библиотеку особенно ценят в сообществе Android, где она помогает преобразовать вложенные обратные вызовы в композицию потоков. В разделе «Применение RxJava в разработке программ для Android» главы 8 мы рассмотрим, как используется RxJava в мобильных устройствах на платформе Android.

Различие между холодными и горячими потоками становится особенно существенным, когда поведение программы зависит от доставки событий. Если объект `Observable` холодный, то мы получим полный и согласованный набор событий, когда бы ни подписались – сразу или спустя несколько часов после создания. С другой стороны, в случае горячего `Observable` нет никакой уверенности в том, что получены все события, начиная с самого первого. Ниже в этой главе мы узнаем о некоторых приемах, позволяющих гарантировать, что каждый подписчик получает все события. Один из таких приемов уже упоминался: оператор `cache()`. Теоретически он может сохранить в буфере все события горячего `Observable`, дав

² Hohpe G., Woolf B. «Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions», Addison-Wesley Professional.

возможность подписчикам, присоединившимся позже, увидеть те же события, что все остальные. Но для этого нужен неограниченный объем памяти, так что будьте осторожны с кэшированием горячих Observable.

Еще одно интересное различие между холодными и горячими источниками – зависимость от времени. Холодный Observable порождает значения по запросу и, возможно, несколько раз, поэтому несущественно, в какой именно момент был создан элемент. Напротив, горячие observable отправляют события по мере поступления, обычно из внешнего источника. Это означает, что момент генерации значения очень важен, т. к. позволяет расположить события на временной оси.

Практический пример: от API обратных вызовов к потоку Observable

Большинство API, имеющихся в Java, например JDBC, java.io, сервлеты³, а также многие коммерческие API являются блокирующими. Это означает, что клиентский поток должен дожидаться результата или побочного эффекта. Но бывают ситуации, которым асинхронность присуща изначально, например, поступление событий из внешнего источника. Технически построить блокирующий API для работы с потоками можно следующим образом:

```
while(true) {  
    Event event = blockWaitingForNewEvent();  
    doSomethingWith(event);  
}
```

Но к счастью, когда предметная область столь очевидно асинхронна, скорее всего, существует какой-то API на основе обратных вызовов, как, например, в JavaScript. Такие API принимают функцию обратного вызова в том или ином виде; чаще всего это выглядит как интерфейс с набором методов, которые можно реализовать для получения уведомлений о различных событиях. Наиболее известные примеры такого API дает чуть ли не каждая графическая библиотека построения пользовательских интерфейсов, например Swing. При использовании таких прослушивателей, как `onClick()` или `onKeyUp()`, обратные вызовы неизбежны. Если вам доводилось работать в таком окружении, то термин *ад обратных вызовов* вам, безусловно, знаком. Обратные вызовы часто бывают вложенными, и управиться с большим числом таких вызовов практически невозможно. Вот пример вложенности с четырьмя уровнями:

```
button.setOnClickListener(view -> {  
    MyApi.asyncRequest(response -> {  
        Thread thread = new Thread() -> {  
            int year = datePicker.getYear();  
            runOnUiThread() -> {  
                button.setEnabled(false);  
                button.setText(" " + year);  
            }  
        }  
    }  
})
```

³ По крайней мере, до версии 3.0.

```
    });  
  });  
  thread.setDaemon(true);  
  thread.start();  
});  
});
```

Удовлетворение простейших требований, например выполнение некоторого действия, когда промежуток времени между двумя обратными вызовами достаточно мал, превращается в кошмар, дополнительно осложненный многопоточностью. В этом разделе мы переработаем API, основанный на обратных вызовах, в RxJava, получив по дороге такие преимущества, как контроль над потоками, управление жизненным циклом и очистку.

Один из моих любимых примеров потоков – обновления состояния, происходящие из Твиттера в виде так называемых *твитов*. Каждую секунду происходит несколько тысяч обновлений. Многие сопровождаются сведениями о геолокации, языке и другими метаданными. Мы воспользуемся библиотекой с открытым исходным кодом Twitter4J (<http://twitter4j.org/en/index.html>), которая умеет доставлять подмножество новых твитов с помощью API на основе обратных вызовов. Мы не собираемся здесь объяснять, как устроена Twitter4J, и не станем придавать значения надежности кода. Эта библиотека выбрана просто как хороший пример API с обратными вызовами в интересной предметной области. Простейшая программа чтения твитов в режиме реального времени могла бы выглядеть так:

```
import twitter4j.Status;  
import twitter4j.StatusDeletionNotice;  
import twitter4j.StatusListener;  
import twitter4j.TwitterStream;  
import twitter4j.TwitterStreamFactory;  
  
TwitterStream twitterStream = new TwitterStreamFactory().getInstance();  
twitterStream.addListener(new twitter4j.StatusListener() {  
    @Override  
    public void onStatus(Status status) {  
        log.info("Состояние: {}", status);  
    }  
  
    @Override  
    public void onException(Exception ex) {  
        log.error("Обратный вызов в случае ошибки", ex);  
    }  
  
    // другие обратные вызовы  
});  
twitterStream.sample();  
TimeUnit.SECONDS.sleep(10);  
twitterStream.shutdown();
```

Вызов метода `twitterStream.sample()` запускает фоновый поток, который подключается к Твиттеру и ожидает новых сообщений. При появлении твита выпол-

няется метод обратного вызова `onStatus`. Поскольку выполнение может происходить в разных потоках, мы не можем полагаться на возбуждение исключений. Вместо этого используется уведомление `onException()`. Проспав 10 секунд, мы вызываем метод `shutdown()`, чтобы остановить поток твитов и освободить захваченные ресурсы, например HTTP-подключения и потоки выполнения.

Все выглядит неплохо, но эта программа ведь ничего и не делает. В реальности мы, наверное, хотели бы как-то обрабатывать каждое сообщение типа `Status` (*твит*). Например, его можно было бы сохранить в базе данных или подать на вход алгоритма машинного обучения. Технически можно было бы поместить все это в метод обратного вызова, но тогда инфраструктурные вызовы оказываются тесно связанными с бизнес-логикой. Простое делегирование отдельному классу лучше, но, к сожалению, такое решение нельзя использовать повторно. В действительности нам хотелось бы четко разделить технические вопросы (получение данных по HTTP-подключению) от прикладных (интерпретация входных данных). Поэтому мы строим второй уровень обратных вызовов.

```
void consume(
    Consumer<Status> onStatus,
    Consumer<Exception> onException) {
    TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
    twitterStream.addListener(new StatusListener() {
        @Override
        public void onStatus(Status status) {
            onStatus.accept(status);
        }

        @Override
        public void onException(Exception ex) {
            onException.accept(ex);
        }

        // другие обратные вызовы
    });
    twitterStream.sample();
}
```

Добавив дополнительный уровень абстракции, мы теперь можем повторно использовать метод `consume()` разными способами. Например, вместо протоколирования можно было бы реализовать сохранение, анализ или обнаружение актов мошенничества:

```
consume(
    status -> log.info("Состояние: {}", status),
    ex -> log.error("Обратный вызов в случае ошибки", ex)
);
```

Однако мы просто переместили проблему на один уровень вверх по иерархии. Что, если требуется подсчитать число твитов в секунду? Или обработать только первые пять? А если мы хотим завести несколько прослушивателей? В каждой из

таких ситуаций придется открывать новое HTTP-подключение. Да и к тому же – что немаловажно – этот API не позволяет отписаться, что чревато утечкой ресурсов. Надеюсь, вы поняли, что мы движемся в сторону API на основе Rx. Вместо того чтобы передавать обратные вызовы в то место, где они могут быть выполнены, мы можем обратить наши взоры к `Observable<Status>` и позволить любому клиенту подписаться в удобный для него момент. Но помните, что показанная ниже реализация все же открывает новое сетевое соединение для каждого подписчика:

```
Observable<Status> observe() {
    return Observable.create(subscriber -> {
        TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
        twitterStream.addListener(new StatusListener() {
            @Override
            public void onStatus(Status status) {
                subscriber.onNext(status);
            }

            @Override
            public void onException(Exception ex) {
                subscriber.onError(ex);
            }

            // другие обратные вызовы
        });
        subscriber.add(Subscriptions.create(twitterStream::shutdown));
    });
}
```

В этот момент мы можем просто вызвать метод `observe()`, который всего лишь создает объект `Observable`, не обращаясь к внешнему серверу. Мы уже знаем, что пока кто-то не подпишется, код внутри `create()` не выполняется. Подписка выглядит почти так же, как и раньше:

```
observe().subscribe(
    status -> log.info("Состояние: {}", status),
    ex -> log.error("Обратный вызов в случае ошибки", ex)
);
```

Существенное отличие от `consume(...)` заключается в том, что нас никто не заставляет передавать обратные вызовы в виде аргументов `observe()`. Вместо этого можно вернуть объект `Observable<Status>`, передавать его другим методам, где-то сохранить и вообще использовать там и тогда, где мы сочтем нужным. Можно составлять композиции нескольких `Observable`, о чем мы поговорим в главе 3. Важный вопрос, который мы еще не затрагивали, – освобождение ресурсов. Когда клиент отписывается, мы должны закрыть поток `TwitterStream`, чтобы избежать утечки ресурсов. Нам уже известны две подходящие техники, воспользуемся сначала более простой:

```
@Override
public void onStatus(Status status) {
```



```
if (subscriber.isUnsubscribed()) {
    twitterStream.shutdown();
} else {
    subscriber.onNext(status);
}
}

@Override
public void onException(Exception ex) {
    if (subscriber.isUnsubscribed()) {
        twitterStream.shutdown();
    } else {
        subscriber.onError(ex);
    }
}
```

Если клиент подписывается, чтобы получить только малую часть потока, наш объект `Observable` позаботится об освобождении ресурсов. Нам известна и другая техника очистки, при которой не нужно дожидаться поступления очередного события. В тот момент, когда подписчик отписывается, мы сразу же вызываем метод `shutdown()`, не дожидаясь, когда придет следующий твит. Этот метод и выполняет очистку (последняя строка):

```
twitterStream.addListener(new StatusListener() {
    // обратные вызовы...
});
twitterStream.sample();

subscriber.add(Subscriptions.create(twitterStream::shutdown));
```

Интересно, что этот объект `Observable` стирает различие между горячими и холодными потоками. С одной стороны, он представляет внешние события, возникновение которых мы не контролируем (горячее поведение), а с другой, события не поступают в нашу систему (из-за отсутствия HTTP-соединения), пока мы не вызовем метод `subscribe()`. И еще один побочный эффект, о котором мы как-то забыли: каждый вызов `subscribe()` запускает новый фоновый поток и создает новое соединение с внешней системой. Один и тот же экземпляр `Observable<Status>` должен допускать использование многими подписчиками, а поскольку тип `Observable` ленивый, то технически должна быть возможность вызвать `observe()` один раз на этапе инициализации и сохранить его в каком-нибудь синглтоне. Однако текущая реализация просто открывает новое соединение, т. е. по сути дела многократно читает одни и те же данные из сети для каждого подписчика. Конечно, мы хотим регистрировать несколько подписчиков на этот поток, но не видно, почему каждый подписчик должен независимо читать одни и те же данные. В действительности нам нужно поведение типа *издатель-подписчик*, когда один издатель (внешняя система) доставляет данные многим подписчикам. Теоретически этого можно добиться с помощью оператора `cache()`, но мы не хотим хранить старые события в буфере вечно. Далее мы исследуем некоторые решения этой проблемы.

Управление подписчиками вручную

Вручную отслеживать всех подписчиков и закрывать соединение с внешней системой только после того, как все они отпишутся, – сизифов труд. Но мы, тем не менее, сделаем это, чтобы в полной мере оценить излагаемые далее идиоматические решения. Идея заключается в том, чтобы хранить подписчиков в коллекции типа `Set<Subscriber<Status>>` и открывать соединение с внешней системой, когда в коллекции появляется первый элемент, а закрывать, когда она оказывается пустой.

// НЕ ДЕЛАЙТЕ ТАК, это решение очень хрупкое и чревато ошибками

```
class LazyTwitterObservable {

    private final Set<Subscriber<? super Status>> subscribers =
        new CopyOnWriteArraySet<>();

    private final TwitterStream twitterStream;

    public LazyTwitterObservable() {
        this.twitterStream = new TwitterStreamFactory().getInstance();
        this.twitterStream.addListener(new StatusListener() {
            @Override
            public void onStatus(Status status) {
                subscribers.forEach(s -> s.onNext(status));
            }

            @Override
            public void onException(Exception ex) {
                subscribers.forEach(s -> s.onError(ex));
            }

            // другие обратные вызовы
        });
    }

    private final Observable<Status> observable = Observable.create(
        subscriber -> {
            register(subscriber);
            subscriber.add(Subscriptions.create(() ->
                this.deregister(subscriber)));
        });

    Observable<Status> observe() {
        return observable;
    }

    private synchronized void register(Subscriber<? super Status> subscriber) {
        if (subscribers.isEmpty()) {
            subscribers.add(subscriber);
            twitterStream.sample();
        } else {
            subscribers.add(subscriber);
        }
    }
}
```

```

    }

    private synchronized void deregister(Subscriber<? super Status> subscriber) {
        subscribers.remove(subscriber);
        if (subscribers.isEmpty()) {
            twitterStream.shutdown();
        }
    }
}

```

Множество subscribers – потокобезопасная коллекция подписавшихся объектов Subscriber. Как только появляется новый подписчик, мы добавляем его в множество и лениво соединяем с источником событий. Наоборот, когда исчезает последний подписчик, мы закрываем соединение с источником. Ключевой момент здесь – то, что имеется ровно одно соединение с внешней системой, а не по одному соединению для каждого подписчика. Это решение работает и вполне надежно, но реализация кажется слишком низкоуровневой и подверженной ошибкам. Доступ к множеству subscribers должен быть синхронизирован, но сама коллекция должна также поддерживать безопасный обход. Вызов register() *обязательно* должен предшествовать добавлению обратного вызова deregister(), в противном случае он будет вызван еще до регистрации подписчика. Наверняка, есть лучший способ реализовать подобное мультиплексирование единственного потока между несколькими подписчиками – и на самом деле подобных механизмов по крайней мере два. Весь смысл RxJava – уменьшить необходимость в опасном трафаретном коде и абстрагировать конкурентность.

Класс rx.subjects.Subject

Класс Subject интересен тем, что расширяет Observable и одновременно реализует интерфейс Observer. А это значит, что его можно рассматривать как Observable на стороне клиента (подписываться на входящие события) и как Observer на стороне поставщика (проталкивать события по запросу путем вызова onNext()). Как правило, мы сохраняем где-то у себя ссылку на Subject, чтобы можно было проталкивать события из любого источника по своему выбору, но для внешнего мира раскрываем этот объект Subject как Observable. Сейчас мы реализуем поток обновления состояния с помощью класса Subject. Чтобы еще упростить реализацию, будем подключаться к внешней системе энергично и не станем вести учет подписчиков. Это не только сделает код проще, но и уменьшит задержку, наблюдаемую первым подписчиком. События уже поступают, и нам не нужно дожидаться подключения к какому-то стороннему приложению.

```

class TwitterSubject {

    private final PublishSubject<Status> subject = PublishSubject.create();

    public TwitterSubject() {
        TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
    }
}

```

```

twitterStream.addListener(new StatusListener() {
    @Override
    public void onStatus(Status status) {
        subject.onNext(status);
    }

    @Override
    public void onException(Exception ex) {
        subject.onError(ex);
    }

    // другие обратные вызовы
});
twitterStream.sample();
}

public Observable<Status> observe() {
    return subject;
}
}

```

`PublishSubject` — один из подклассов `Subject`. Мы энергично начинаем принимать события от внешней системы и просто проталкиваем их (вызывая метод `subject.onNext(...)`) всем подписчикам. Объект `Subject` сам запоминает эти события, так что нам думать об этом не нужно. Отметим, что мы возвращаем `subject` из метода `observe()`, притворяясь, что это самый обычный `Observable`. Любой новый подписчик будет получать все последующие события сразу после вызова `onNext()` на стороне сервера — по крайней мере, до тех пока не отпишется. Поскольку `Subject` самостоятельно управляет жизненным циклом объектов `Subscriber`, нам остается только вызвать `onNext()`, не думая о том, сколько подписчиков нас слушает.



Распространение ошибок в объектах `Subject`

Темы (объекты `Subject`) полезны, но с ними связано много тонкостей, о которых нужно знать. Например, после вызова метода `subject.onError()` объект `Subject` молча игнорирует все последующие уведомления об ошибках `onError`, т. е. «проглатывает» их.

`Subject` — полезное средство создания экземпляров `Observable` в случае, когда метод `Observable.create(...)` кажется чрезмерно сложным. Перечислим другие подклассы `Subject`.

`AsyncSubject`

Запоминает последнее порожденное значение и отправляет его подписчикам при вызове `onComplete()`. Пока `AsyncSubject` не завершен, все события, кроме последнего, отбрасываются.

`BehaviorSubject`

Отправляет все события, порожденные после подписки, как и `PublishSubject`. Но сначала отправляет самое последнее событие, предшествовавшее под-

писке. Это позволяет подписчику сразу получить информацию о состоянии потока. Например, `Subject` может представлять текущую температуру, которая рассылается раз в минуту. Подписавшийся клиент сразу получит результат последнего измерения температуры, не дожидаясь следующего события. История показаний его не интересует, только последняя измеренная температура. Если еще не было отправлено ни одного события, то первым отправляется специальное событие по умолчанию (если таковое было задано).

ReplaySubject

Самый интересный тип `Subject`, который кэширует все прошедшие через него события. Новый подписчик сначала получает пачку прошлых (кэшированных событий), а затем начинает получать последующие события в реальном времени. По умолчанию кэшируются все события, случившиеся с момента создания `Subject`. Это опасно, если поток бесконечный или очень длинный. На этот случай предусмотрены подклассы `ReplaySubject` для хранения ограниченного множества событий:

- задается число хранящихся в памяти событий (`createWithSize()`);
- задается максимальное время хранения событий (`createWithTime()`);
- или одновременно максимальное число и максимальное время (что произойдет раньше) (`createWithTimeAndSize()`).

К объектам `Subject` следует подходить с осторожностью: иногда существуют и более идиоматические способы разделения ресурсов между подписчиками и кэширования событий (см., например, раздел «Тип `ConnectableObservable`» ниже). Пока остановитесь на относительно низкоуровневом методе `Observable.create()` или, что еще лучше, на стандартных фабричных методах типа `from()` и `just()`.

Следует также помнить о конкурентности. По умолчанию вызов метода `onNext()` объекта `Subject` приводит к вызову метода `onNext()` всех наблюдателей `Observer`. Неудивительно, что эти методы называются одинаково. В каком-то смысле обращение к методу `onNext()` объекта `Subject` косвенно вызывает `onNext()` каждого подписчика `Subscriber`. Но следует помнить, что согласно руководству «*Rx Design Guidelines*» все вызовы `onNext()` объекта `Observer` должны быть сериализованы, т. е. `onNext()` нельзя одновременно вызывать из двух разных потоков. Однако при некоторых способах работы с `Subject` это правило легко нарушить – например, вызвать `Subject.onNext()` из нескольких потоков в пуле. К счастью, если вы опасаетесь такой ситуации, просто вызовите `Subject.toSerialized()` – полный аналог метода `Observable.serialize()`. Этот оператор гарантирует, что события будут отправляться в правильном порядке.

Тип ConnectableObservable

Тип `ConnectableObservable` дает интересный способ координации нескольких подписчиков и совместного использования одной подписки. Помните нашу первую попытку создать одиночное/ленивое соединение с внешним источником в виде

класса `LazyTwitterObservable`? Нам пришлось вручную отслеживать всех подписчиков, открывать соединение при появлении первого и закрывать при исчезновении последнего. Тип `ConnectableObservable` наследует `Observable` и гарантирует, что в любой момент времени существует не более одного подписчика, хотя в действительности их может много и все они разделяют один внешний ресурс.

У типа `ConnectableObservable` много применений, например, если нужно гарантировать, что все подписчики получают одну и ту же последовательность событий вне зависимости от того, когда подписались. Кроме того, `ConnectableObservable` может принудительно имитировать подписку, если это приводит к важным побочным эффектам, даже в случае, когда ни одного «реального» подписчика еще нет. Скоро мы рассмотрим эти сценарии. Класс `Subject` дает императивные способы создания `Observable`, тогда как `ConnectableObservable` экранирует истинный `Observable` и гарантирует, что с ним сможет взаимодействовать максимум один подписчик. Сколько бы подписчиков ни установили соединение с `ConnectableObservable`, он открывает только одну подписку на объект `Observable`, по которому был создан.

Реализация единственной подписки с помощью `publish().refCount()`

Напомним: у нас имеется один описатель внешнего ресурса, например, HTTP-подключение к потоку обновлений состояния в Твиттере. Однако объект `Observable`, рассылающий эти события, должен сообщать использоваться несколькими подписчиками. Созданная ранее наивная реализация `Observable` не умела это контролировать, поэтому каждый подписчик получал свое собственное подключение. Это очень расточительно:

```
Observable<Status> observable = Observable.create(subscriber -> {
    System.out.println("Открывается соединение");
    TwitterStream twitterStream = new TwitterStreamFactory().getInstance();
    //...
    subscriber.add(Subscriptions.create(() -> {
        System.out.println("Разрывается соединение");
        twitterStream.shutdown();
    }));
    twitterStream.sample();
});
```

При использовании этого `Observable` каждый подписчик создает новое соединение. Выглядит это так:

```
Subscription sub1 = observable.subscribe();
System.out.println("Подписка 1");
Subscription sub2 = observable.subscribe();
System.out.println("Подписка 2");
sub1.unsubscribe();
System.out.println("Отписка 1");
sub2.unsubscribe();
System.out.println("Отписка 2");
```

И вот что печатается:

```
Открывается соединение
Подписка 1
Открывается соединение
Подписка 2
Разрывается соединение
Отписка 1
Разрывается соединение
Отписка 2
```

На этот раз мы для простоты воспользовались методом `subscribe()` без параметров, который активизирует подписку, но игнорирует все события и уведомления. Уделив почти половину главы сражению с этой проблемой, мы теперь знаем о многих возможностях RxJava и можем, наконец, предложить самое масштабируемое и простое решение: композицию `publish().refCount()`:

```
lazy = observable.publish().refCount();
//...
System.out.println("До всех подписчиков");
Subscription sub1 = lazy.subscribe();
System.out.println("Подписка 1");
Subscription sub2 = lazy.subscribe();
System.out.println("Подписка 2");
sub1.unsubscribe();
System.out.println("Отписка 1");
sub2.unsubscribe();
System.out.println("Отписка 2");
```

Вот теперь печатается то, чего мы так долго добивались:

```
До всех подписчиков
Открывается соединение
Подписка 1
Подписка 2
Отписка 1
Разрывается соединение
Отписка 2
```

Соединение не устанавливается, пока не появится первый подписчик. Важнее, однако, то, что при появлении второго подписчика новое соединение не создается, до исходного объекта `observable` даже дело не доходит. Тандем `publish().refCount()` обортывает этот объект и перехватывает все подписки. Позже мы объясним, зачем нужны два метода и что означает использование метода `publish()` в одиночестве. Ну а пока сосредоточимся на `refCount()`. Этот оператор подсчитывает, сколько имеется активных подписчиков, — прямо как при подсчете ссылок в старых алгоритмах сборки мусора. Когда это число меняется с нуля на единицу, производится подписка на объект `Observable`. Любое значение, кроме 1, игнорируется, и один и тот же обернутый объект `Subscriber` совместно используется всеми обращающимися подписчиками. Но когда последний подписчик отписывается, счетчик обращается в нуль и `refCount()` понимает, что пора отписаться.

Таким образом, `refCount()` делает ровно то, что мы реализовали вручную в классе `LazyTwitterObservable`. Дует `publish().refCount()` позволяет организовать разделение единственного объекта `Subscriber`, не жертвуя ленивостью. Эта пара операторов используется настолько часто, что для нее даже введен псевдоним `share()`. Имейте в виду, что если между последней отпиской и новой подпиской проходит даже очень краткое время, `share()` все равно переподключается, как если бы кэширования не было вовсе.

Жизненный цикл *ConnectableObservable*

Еще одно полезное применение оператора `publish()` – принудительная имитация подписки в отсутствие подписчиков. Возьмем, к примеру, наш объект `Observable<Status>`. Прежде чем делать его доступным клиентам, мы хотим сохранить все события в базе данных, даже если еще нет ни одного подписчика. Наивный подход не годится:

```
Observable<Status> tweets = //...
return tweets
    .doOnNext(this::saveStatus);
```

Мы воспользовались оператором `doOnNext()`, который просматривает все значения, проходящие в потоке, и выполняет некоторое действие, например `saveStatus()`. Вспомним, однако, что тип `Observable` ленивый, а это значит, что пока кто-то не подписался, метод `doOnNext()` не вызывается. Нам нужен некий фиктивный `observer`, который не прослушивает события, но заставляет объекты `observable` порождать события. И надо же – существует перегруженный вариант `subscribe()`, который именно это и делает:

```
Observable<Status> tweets = //...
tweets
    .doOnNext(this::saveStatus)
    .subscribe();
```

Этот метод `subscribe()` без параметров вызывает метод `Observable.create()` и устанавливает соединение с источником событий. На первый взгляд, проблема решена, но мы опять забыли защититься от нескольких подписчиков. Если открыть доступ к объекту `tweets`, то второй подписчик может попытаться подключиться к внешнему ресурсу, например, создать второе HTTP-подключение. Идиоматичный способ решения – воспользоваться парой `publish().connect()`, которая сразу создает искусственный объект `Subscriber`, и это будет единственный подписчик на входной поток. Проще пояснить на примере. И заодно разберемся, как работает `publish()` без сопровождения:

```
ConnectableObservable<Status> published = tweets.publish();
published.connect();
```

Наконец-то мы видим `ConnectableObservable` во всем блеске. Мы можем вызвать метод `Observable.publish()` любого объекта `Observable` и получить в ответ объ-

ект `ConnectableObservable`. Можно и дальше использовать исходный `Observable` (в данном случае `tweets`); `publish()` на него никак не влияет. Но нас больше интересует возвращенный объект `ConnectableObservable`. Любой клиент, подписавшийся на `ConnectableObservable`, запоминается в множестве подписчиков. Пока метод `connect()` не вызван, эти подписчики прозябают втуне, ни один из них не подписан напрямую на обернутый `Observable`. Но стоит вызвать `connect()`, как специальный подписчик-посредник подписывается на `Observable` (`tweets`) вне зависимости от того, сколько подписчиков запомнено в множестве – пусть даже ни одного. И если в множестве, которое хранит `ConnectableObservable`, есть подписчики, то все они будут получать одну и ту же последовательность уведомлений.

У такого механизма есть ряд преимуществ. Допустим, в нашем приложении имеется `Observable`, в котором заинтересовано несколько подписчиков. На этапе инициализации какие-то компоненты (например, бран-объекты Spring или EJB) подписываются на этот `Observable` и начинают прослушивать его. Не будь `ConnectableObservable`, вполне могло бы случиться, что горячий `Observable` сразу начнет отправлять события. Тогда первый подписчик получит все, а те, что подключились чуть позже, могут несколько первых событий пропустить. Если мы хотим, чтобы все подписчики имели одинаковое представление о мире, то это может оказаться проблемой. Все будут получать события в одном и том же порядке, но опоздавшие начальные события не увидят. Решение проблемы состоит в том, чтобы сначала опубликовать `Observable` методом `publish()` и разрешить всем компонентам системы подписываться на него методом `subscribe()`, например, на этапе инициализации приложения. После того как все подписчики, которым требуется получать одну и ту же последовательность событий (в т. ч. самые первые), получили шанс подписаться, приложение подключает объект `ConnectableObservable` методом `connect()`. В результате создается единственный объект `Subscriber`, имеющий доступ к обернутому `Observable`, и именно он начинает рассылать события всем подписчикам. В примере ниже используется каркас Spring (<http://projects.spring.io/spring-framework/>), но вообще решение не привязано ни к какому конкретному каркасу:

```
import org.springframework.context.ApplicationListener;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.event.ContextRefreshedEvent;
import rx.Observable;
import rx.observables.ConnectableObservable;

@Configuration
class Config implements ApplicationListener<ContextRefreshedEvent> {

    private final ConnectableObservable<Status> observable =
        Observable.<Status>create(subscriber -> {
            log.info("Начало");
            //...
        }).publish();

    @Bean
```

```
public Observable<Status> observable() {
    return observable;
}

@Override
public void onApplicationEvent(ContextRefreshedEvent event) {
    log.info("Подключение");
    observable.connect();
}
}

@Component
class Foo {

    @Autowired
    public Foo(Observable<Status> tweets) {
        tweets.subscribe(status -> {
            log.info(status.getText());
        });
        log.info("Подписка");
    }
}

@Component
class Bar {

    @Autowired
    public Bar(Observable<Status> tweets) {
        tweets.subscribe(status -> {
            log.info(status.getText());
        });
        log.info("Подписка");
    }
}
```

В этом простом приложении первым делом создается объект типа `Observable` (на самом деле, производного от него типа `ConnectableObservable`). Поскольку тип `Observable` ленивый, такие объекты можно создавать даже статически. Это объект публикуется методом `publish()`, поэтому все подписывающиеся на него объекты `Subscriber` будут запомнены, но не получат уведомлений, пока мы не вызовем метод `connect()`. Далее идут два класса, помеченные аннотацией `@Component`, которым необходим этот `Observable`. Механизм внедрения зависимостей подставляет наш объект типа `ConnectableObservable` и позволяет подписаться на него любому желающему. Однако даже в случае горячего `Observable` события не начнут поступать до полного окончания этапа инициализации. После того как все компоненты будут созданы и связаны между собой, каркас отправит событие `ContextRefreshedEvent`. Гарантируется, что в этот момент каждый компонент имел возможность запросить `Observable` и подписаться на него методом `subscribe()`. Когда приложение готово начать работу, мы вызываем `connect()`. В результате однократно производится подписка на обернутый `Observable` и каждому компоненту отправляется в точности одна и та же последовательность событий. Ниже показан фрагмент журнала (имена компонентов заключены в квадратные скобки):

```
[Foo ] Подписка
[Bar ] Подписка
[Config] Подключение
[Config] Начало
[Foo ] Msg 1
[Bar ] Msg 1
[Foo ] Msg 2
[Bar ] Msg 2
```

Обратите внимание, что компоненты `Foo` и `Bar` сообщают, что подписались, хотя еще не получили ни одного события. И лишь после окончания инициализации приложения метод `connect()` производит подписку на обернутый `Observable` и начинается доставка сообщений `Msg 1` и `Msg 2` всем компонентам. Сравните с поведением обычного `Observable` в той же ситуации, когда `ConnectableObservable` не используется и каждому компоненту разрешено подписываться немедленно:

```
[Config] Начало
[Foo ] Подписка
[Foo ] Msg 1
[Config] Начало
[Bar ] Подписка
[Foo ] Msg 2
[Bar ] Msg 2
```

Есть два различия, которые следует иметь в виду. Самое главное: сразу после подписки компонента `Foo` открывается соединение с внешним ресурсом – не дожидаясь завершения инициализации приложения. Хуже того, компонент `Bar` вызывает открытие второго соединения (сообщение `Начало` печатается дважды). И во-вторых – вы заметили, что компонент `Bar` получил событие `Msg 2`, а событие `Msg 1` до него так и не дошло, доставшись только компоненту `Foo`? Является ли такая несогласованность при потреблении горячего потока `Observable` проблемой или нет, зависит от обстоятельств, но знать о ней в любом случае необходимо.

Резюме

Создание объектов `Observable` и подписка на них – важнейшая особенность `RxJava`. Начинающие часто забывают о подписке и удивляются, почему не порождается никаких событий. Многие разработчики с восторгом принимают замечательные операторы, предоставляемые библиотекой (см. главу 3), но не понимают, как эти операторы под капотом производят подписку, а это может стать причиной тонких ошибок.

Кроме того, асинхронную природу `RxJava` часто рассматривают как нечто само собой разумеющееся, хотя это не всегда так. На самом деле, большинство операторов в `RxJava` не пользуются никаким пулом потоков. Точнее, по умолчанию никакой конкурентности не предполагается и все происходит в потоке клиента. Это еще один важный урок, который следует вынести из прочитанной главы. Теперь, понимая принципы подписки и конкурентности, вы можете начать пользоваться `RxJava` эффективно и без досадных разочарований.

В главе 3 мы рассмотрим встроенные в библиотеку операторы и способы их комбинирования. Декларативные преобразования и композиция потоков – это именно то, что составляет очарование RxJava.

Глава 3.

Операторы и преобразования

Цель этой главы – познакомиться с основами операторов в RxJava и способами их комбинирования для построения высокоуровневых, легко обозримых конвейеров. Одна из причин популярности RxJava – богатый набор встроенных операторов и возможность создания новых. Оператор – это функция, которая принимает предыдущий объект `Observable<T>` и возвращает следующий объект `Observable<R>`, причем типы `T` и `R` необязательно совпадают. Операторы допускают композицию простых преобразований в сложные графы обработки.

Например, оператор `Observable.filter()` получает элементы из объекта `Observable`, но пропускает дальше только те, что удовлетворяют заданному предикату. С другой стороны, оператор `Observable.map()` преобразует получаемые элементы на лету. Это позволяет извлекать, обогащать и обертывать исходные события. Есть и не столь привычные операторы. Например, `Observable.delay()` передает события без изменения, но добавляет перед каждым фиксированную задержку. Наконец, существуют операторы (например, `Observable.buffer()`), которые потребляют несколько входных событий перед тем, как отправить их дальше, возможно, собрав в пакет.

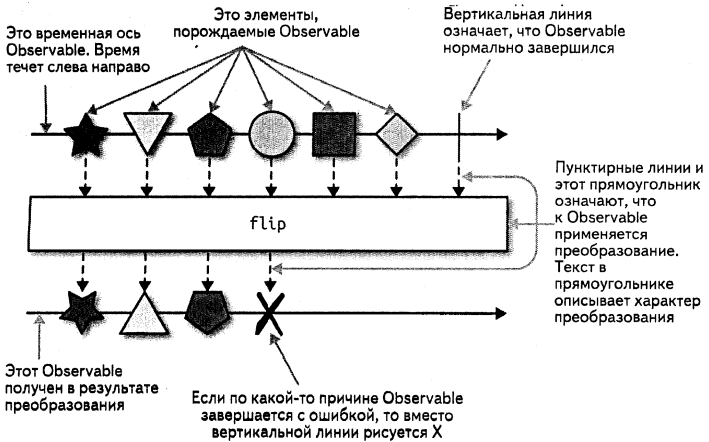
Но осознать красоту операторов Rx мало, их истинная мощь раскрывается, когда мы начинаем их комбинировать. Сцепление нескольких операторов, разветвление потока на несколько подпотоков и последующее их слияние – для всего этого есть идиомы, которыми вы должны свободно владеть.

Базовые операторы: отображение и фильтрация

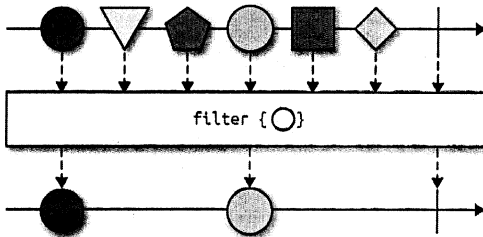
Большинство операторов реализовано в виде методов экземпляра типа `Observable`, они тем или иным способом изменяют поведение исходного `Observable`, видимое следующим за ним объектам `Observable` или подписчикам. Звучит запутанно, но на самом деле механизм очень гибкий и осваивается без особого труда. Один из самых простых операторов – `filter()` – принимает предикат и либо передает события дальше, либо отбрасывает их:

```
Observable<String> strings = //...
Observable<String> filtered = strings.filter(s -> s.startsWith("#"));
```

Настало время познакомиться с так называемыми *камышковыми диаграммами* (marble¹ diagram), повсеместно используемыми в документации по RxJava для наглядности. Камышковые диаграммы иллюстрируют работу операторов. Как правило, диаграмма содержит две горизонтальные временные оси, направленные слева направо. Фигурки на диаграммах (те самые «камышки») служат для визуализации событий. Между осями располагается оператор, который так или иначе изменяет последовательность событий, поступающих от исходного Observable и передаваемых в результирующий. На рисунке ниже приведен пример.



Ниже показана диаграмма, описывающая оператор `filter()`. Метод `observable.filter()` возвращает те же события (поэтому камышки на верхней и нижней оси одинаковы), но некоторые события пропускаются, потому что не удовлетворяют предикату.



¹ Marbles – это те самые шарики, в которые любил играть Том Сойер. «Джим, я дам тебе шарик. Я дам тебе мой белый алебастровый шарик». – Прим. перевод.

Для некоторых объектов `Observable` нас могут интересовать не все события, например, если объем данных очень велик. Кроме того, часто к одному и тому же `Observable` применяют несколько операторов `filter()` с разными предикатами. Выглядит это так: `(filter(p1).filter(p2).filter(p3))` – и эквивалентно логической конъюнкции `(filter(p1 && p2 && p3))`. У свертывания соседних операторов в один (это относится не только к `filter()`) есть свои плюсы и минусы. Наличие нескольких мелких преобразований (например, нескольких фильтров) предпочтительно, если эти преобразования можно использовать повторно или включить в другую композицию. С другой стороны, чем больше операторов, тем выше накладные расходы² и глубже стек. Какой стиль выбрать, зависит от требований и вашего стиля кодирования:

```
Observable<String> strings = someFileSource.lines();
Observable<String> comments = strings.filter(s -> s.startsWith("#"));
Observable<String> instructions = strings.filter(s -> s.startsWith(">"));
Observable<String> empty = strings.filter(String::isBlank);
```

Возникает вопрос: «Что происходит с исходным источником `strings`?». Читатели, знакомые с объектно-ориентированным программированием, возможно, помнят, что методы типа `java.util.List.sort()` переупорядочивают элементы списка на месте и ничего не возвращают. Тип `List<T>` в Java изменяемый (хорошо это или плохо), поэтому переупорядочивать его содержимое разрешается. Можно также вообразить себе гипотетический метод `void List.filter()`, который принимает предикат и удаляет из внутреннего представления списка не удовлетворяющие ему элементы. Но в RxJava следует забыть об изменении внутренних структур данных: модификация переменных вне потока считается крайне неидиоматичной и опасной практикой. Каждый оператор возвращает *новый* объект `Observable`, оставляя исходный без изменения.

Это значительно упрощает рассуждения о потоке событий. Поток может разветвляться на несколько независимых источников, каждый со своими характеристиками. Одна из сильных сторон RxJava заключается в возможности использовать один объект `Observable` в разных местах, никак не затрагивая интересы других потребителей. Передавая `Observable` какой-то неизвестной функции, можно быть уверенным, что она его никак не повредит. Того же не скажешь об объекте изменяемого типа `java.util.Date` – его может модифицировать любой код, имеющий ссылку на объект. Именно поэтому новый API `java.time` сделан абсолютно неизменяемым.

Взаимно однозначные преобразования с помощью `map()`

Допустим, что имеется поток событий и нам нужно применить к каждому событию некоторое преобразование. Это может быть создание Java-объекта по его

² Ведутся исследования по технологии слияния операторов (*fusion*), которая прозрачно свертывает несколько операторов в один.

представлению в формате JSON (или наоборот), обогащение, обертывание, выделение из события и т. д. На помощь приходит бесценный оператор `map()`:

```
import rx.functions.Func1;

Observable<Status> tweets = //...
Observable<Date> dates = tweets.map(new Func1<Status, Date>() {
    @Override
    public Date call(Status status) {
        return status.getCreatedAt();
    }
});

Observable<Date> dates =
    tweets.map((Status status) -> status.getCreatedAt());

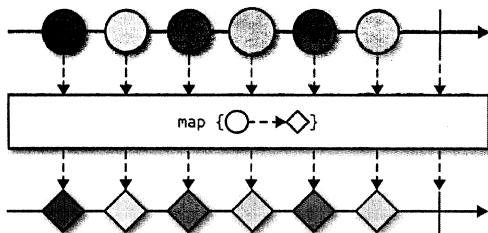
Observable<Date> dates =
    tweets.map((status) -> status.getCreatedAt());

Observable<Date> dates =
    tweets.map(Status::getCreatedAt);
```

Все показанные выше способы определения объекта `dates` эквивалентны, начиная с самого громоздкого с использованием `Func1<T, R>` и кончая компактным синтаксисом (появившимся в Java 8), в котором используется ссылка на метод и выведение типа. Но взгляните внимательно! Исходный объект `tweets` порождает события типа `Status`. Затем мы вызываем метод `map()`, передавая ему функцию, которая принимает одно событие (`Status s`) и возвращает значение типа `Date`. Кстати говоря, использование изменяемых событий (таких, как `java.util.Date`) составляет проблему, потому что любой оператор или подписчик может случайно изменить событие, потребляемое другими подписчиками. Мы можем легко исправить это, применив еще один оператор `map()`:

```
Observable<Instant> instants = tweets
    .map(Status::getCreatedAt)
    .map((Date d) -> d.toInstant());
```

Камешковая диаграмма `map()` выглядит следующим образом:



Оператор `map()` принимает функцию, которая может изменить форму входного события с круглой на квадратную. Это преобразование применяется к каждому проходящему событию. Теперь самое время задать контрольный вопрос на понимание работы `Observable`. Взгляните на следующий код и попробуйте догадаться, какое значение получит подписавшийся на этот объект клиент:

```
Observable
    .just(8, 9, 10)
    .filter(i -> i % 3 > 0)
    .map(i -> "#" + i * 10)
    .filter(s -> s.length() < 4);
```

Объекты `Observable` *ленивые*, т. е. не начинают порождать события, пока на них кто-то не подпишется. Мы можем создать бесконечный поток, который тратит несколько часов на вычисление первого значения, но пока мы явно не выразим желание получать уведомления о событиях, `Observable` остается пассивной, ничего не делающей структурой данных для порождения значений некоторого типа `t`. Это относится даже к горячим `Observable` — хотя источник без усталости генерирует события, ни один оператор, в частности `map()` и `filter()` не вычисляется, пока какой-то клиент не проявит интерес. В противном случае мы просто выбрасывали бы результаты вычислений, а это совсем уж бессмысленно. Всякий раз, используя оператор, в том числе и те, которые еще не рассматривались, мы создаем обертку вокруг исходного объекта `Observable`. Эта обертка может перехватывать проходящие события, но обычно не подписывается сама:

```
Observable
    .just(8, 9, 10)
    .doOnNext(i -> System.out.println("A: " + i))
    .filter(i -> i % 3 > 0)
    .doOnNext(i -> System.out.println("B: " + i))
    .map(i -> "#" + i * 10)
    .doOnNext(s -> System.out.println("C: " + s))
    .filter(s -> s.length() < 4)
    .subscribe(s -> System.out.println("D: " + s));
```

Протоколирование или иное обследование проходящих сообщений — вещь настолько полезная, что существует специальный *нечистый* оператор `doOnNext()`, который позволяет заглянуть в сообщение, не изменяя его. Нечистым он является, потому что может иметь побочные эффекты, например, протоколирование или доступ к глобальному состоянию. Оператор `doOnNext()` просто получает любое событие, отправленное предыдущим `Observable`, и передает его следующему без какой-либо модификации. Можно считать его зондом, который безопасно вставляется в любое место конвейера объектов `Observable`, чтобы посмотреть, что по нему проходит. Это прямолинейная реализация паттерна Подслушивание (*Wiretap*) (<http://www.enterpriseintegrationpatterns.com/patterns/messaging/WireTap.html>), описанного в книге Hohpe, Woolf «Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions» (Addison-Wesley)³. Технически `doOnNext()`

³ Г. Хоп, Б. Вульф «Шаблоны интеграции корпоративных приложений», Вильямс, 2000.

может изменить событие. Однако изменение событий, управляемых объектом `Observable`, — прямой путь к катастрофе. Вскоре мы научимся обрабатывать события конкурентно, разветвлять потоки и т. д. Гарантировать потокобезопасное изменение события в таких условиях было бы очень трудно. Поэтому на практике все типы, обернутые `Observable`, следует считать неизменяемыми.

Сначала пройдем по пути, который выбирает `RxJava`. В каждой строке показанного выше кода создается новый объект `Observable`, который в некотором смысле обертывает предыдущий. Например, первый оператор `filter()` не удаляет число 9 из потока `Observable.just(8, 9, 10)`, а создает новый поток, который отправит подписчику значения 8 и 10. Этот принцип относится к большинству операторов: они не модифицируют содержимое или поведение существующего `Observable`, а создают новые. Однако говорить, что `filter()` или `map()` *создает* новый объект `Observable`, не вполне точно. Большинство операторов ленивые и ничего не делают, пока кто-то не подпишется. Так что же происходит, когда `Rx` видит вызов `subscribe()` в конце цепочки? Понимание внутренних механизмов работы поможет разобраться в том, как потоки обрабатываются под капотом. Будем анализировать код снизу вверх.

- Сначала `subscribe()` информирует предыдущий `Observable`, что хочет получать значения.
- У предыдущего `Observable (filter(s -> s.length() < 4))` своих значений нет, это всего лишь декоратор, обертывающий другой объект `Observable`. Поэтому он сам подписывается на обернутый объект.
- Оператор `map(i -> "#" + i * 10)`, как и `filter()`, сам ничем не владеет. Он просто преобразует то, что получает, а значит, должен подписаться на предыдущий `Observable`, как и все остальные.
- Все меняется, когда мы доходим до оператора `just(8, 9, 10)`. Этот `Observable` является истинным источником событий. Как только оператор `filter(i -> i % 3 > 0)` подписывается на него (в результате явного вызова `subscribe()` далеко внизу), он начинает отправлять события в поток.
- Теперь мы можем понаблюдать за тем, как события проходят по всем этапам конвейера. `filter()` получает от соседа число 8 и передает его дальше (поскольку предикат `i % 3 > 0` возвращает `true`). Затем `map()` преобразует 8 в строку `"#80"` и пробуждает следующий оператор `filter()`.
- Предикат `s.length() < 4` удовлетворяется, поэтому мы наконец печатаем преобразованное значение на `System.out`.

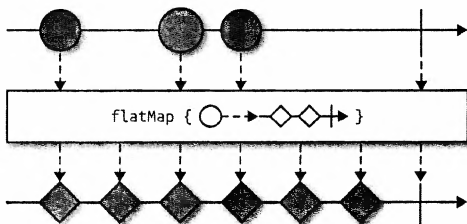
Проследите за тем, как отбрасываются числа 9 и 10. В итоге операторы `doOnNext()` формируют следующую распечатку:

```
A: 8
B: 8
C: #80
D: #80
A: 9
```

A: 10
B: 10
C: #100

Обертывание с помощью flatMap()

`flatMap()` — один из самых важных операторов в RxJava. На первый взгляд, он напоминает `map()`, но в результате преобразования каждого элемента может быть возвращен еще один (вложенный, внутренний) объект `Observable`. Вспомнив, что `Observable` может представлять асинхронную операцию, мы приходим к выводу, что `flatMap()` можно использовать для запуска асинхронного вычисления для каждого входного события (разветвление) и последующего объединения результатов. Оператор `flatMap()` принимает объект типа `Observable<T>` и функцию, преобразующую `T` в `Observable<R>`. Сначала `flatMap()` конструирует объект `Observable<Observable<R>>`, заменяя все значения типа `T` объектами типа `Observable<R>` (как `map()`). Но на этом дело не кончается: оператор автоматически подписывается на эти внутренние потоки `Observable<R>`, порождая в результате один поток значений типа `R`, содержащий все значения из внутренних потоков в порядке их поступления. Принцип работы показан на следующей камешковой диаграмме:



На этой диаграмме отражен один важный аспект `flatMap()`. Каждое входящее событие (круг) преобразуется в объект `Observable`, содержащий два ромба, разделенных некоторой задержкой. Если интервал между двумя входящими событиями мал, то `flatMap()` автоматически применит преобразования и превратит их в два потока ромбов. Но поскольку RxJava одновременно подписывается на оба и объединяет их, то события из одного внутреннего `Observable` могут чередоваться с событиями из другого. Мы исследуем это поведение ниже.

`flatMap()` — самый фундаментальный оператор в RxJava, с его помощью легко⁴ реализовать `map()` и `filter()`:

```
import static rx.Observable.empty;
import static rx.Observable.just;

numbers.map(x -> x * 2);
```

⁴ Однако из соображений производительности в RxJava имеются отдельные реализации `map()` и `filter()`.

```
numbers.filter(x -> x != 10);

// эквивалентно
numbers.flatMap(x -> just(x * 2));
numbers.flatMap(x -> (x != 10) ? just(x) : empty());
```

Но приведем более реалистичный пример `flatMap()`. Допустим, мы получаем поток фотографий машин, въезжающих на скоростную автостраду. Для каждой машины выполняется довольно сложный алгоритм распознавания символов, который возвращает номер машины. Понятно, что распознавание может завершиться неудачно, и тогда алгоритм не возвращает ничего. Он может также завершиться с ошибкой или по какой-то странной причине вернуть два номера для одной машины. Все это легко смоделировать с помощью объектов `Observable`:

```
Observable<CarPhoto> cars() {
    //...
}

Observable<LicensePlate> recognize(CarPhoto photo) {
    //...
}
```

Взяв `Observable<LicensePlate>` за основу потока данных, мы сможем построить модель, учитывающую следующие случаи:

- на фотографии машины нет номера (пустой поток);
- фатальная внутренняя ошибка (обратный вызов `onError()`); например, когда модуль распознавания полностью и необратимо вышел из строя;
- распознан один или несколько номеров, после чего следует событие `onComplete()`.

Или того лучше — `recognize()` может всякий раз возвращать несколько прогрессивно улучшающихся результатов, например, начинать с грубой оценки или одновременно выполнять два алгоритма. Вот как можно было бы воспользоваться показанными выше методами:

```
Observable<CarPhoto> cars = cars();

Observable<Observable<LicensePlate>> plates =
    cars.map(this::recognize);

Observable<LicensePlate> plates2 =
    cars.flatMap(this::recognize);
```

Любой результат, возвращенный функцией, которая передана `map()`, снова обернется объектом `Observable`. Это означает, что если функция возвращает `Observable<LicensePlate>`, то на выходе `map()` мы получим `Observable<Observable<LicensePlate>>`. Но мало того что с вложенными объектами `Observable` неудобно работать, так еще нужно сначала подписаться на каждый внутренний `Observable`, чтобы получить хоть какие-то результаты. К тому же, придется каким-то образом синхронизировать внутренние результаты, сформировав из них единый поток, а это очень трудно.

`flatMap()` решает эти проблемы путем *линеаризации* результата, так что на выходе получается простой поток объектов `LicensePlate`. Позже, в разделе «Многопоточность в RxJava» главы 4 мы научимся распараллеливать работу с помощью `flatMap()`. В общем, оператор `flatMap()` стоит использовать в следующих ситуациях:

- результат преобразования оператором `map()` должен иметь тип `Observable`, например, если к каждому элементу потока применяется длительная асинхронная операция без блокировки;
- преобразование имеет вид один-ко-многим, т. е. одно входящее событие порождает несколько подсобытий. Например, поток заказчиков преобразуется в поток их заказов, причем каждому заказчику может соответствовать произвольное количество заказов.

Предположим теперь, что мы хотели бы использовать метод, возвращающий объект типа `Iterable` (например, `List` или `Set`). Так, если в классе `Customer` имеется простой метод `List<Order> getOrders()`, то мы будем вынуждены написать несколько операторов, чтобы воспользоваться им в конвейере `Observable`:

```
Observable<Customer> customers = //...
Observable<Order> orders = customers
    .flatMap(customer ->
        Observable.from(customer.getOrders()));
```

Или – эквивалентно и столь же многословно:

```
Observable<Order> orders = customers
    .map(Customer::getOrders)
    .flatMap(Observable::from);
```

Потребность отображать один элемент на `Iterable` возникает так часто, что специально для этого преобразования был создан оператор `flatMapIterable()`:

```
Observable<Order> orders = customers
    .flatMapIterable(Customer::getOrders);
```

Обертывая методы объектами `Observable`, следует проявлять осторожность. Если бы `getOrders()` был не простым методом доступа, а длительной операцией, то было бы лучше реализовать его так, чтобы явно возвращался объект `Observable<Order>`.

Еще один интересный вариант `flatMap()` может реагировать не только на данные, также на уведомления об ошибках и завершении. Ниже описана сигнатура этого перегруженного варианта. Для типа `Observable<T>` мы должны предоставить:

- функцию `T -> Observable<R>`;
- функцию, отображающую уведомление об ошибке на `Observable<R>`;
- функцию без аргументов, которая реагирует на завершение предыдущего потока и возвращает `Observable<R>`.

Вот как это выглядит в виде кода:

```
<R> Observable<R> flatMap(
    Func1<T, Observable<R>> onNext,
    Func1<Throwable, Observable<R>> onError,
    Func0<Observable<R>> onCompleted)
```

Допустим, мы создаем службу загрузки видео на сервер. Она принимает `UUID` и возвращает ход загрузки в виде потока типа `Observable<Long>` – сколько байтов уже передано. Мы можем распорядиться этой информацией как угодно – например, отобразить ее в пользовательском интерфейсе. Но по-настоящему нас интересует завершение потока, т. е. окончание загрузки. Лишь после успешной загрузки можно начать рейтингование видео. Наивная реализация просто подписалась бы на поток и игнорировала все события, кроме завершения (оно всегда должно быть последним):

```
void store(UUID id) {
    upload(id).subscribe(
        bytes -> {}, // игнорируем
        e -> log.error("Ошибка", e),
        () -> rate(id)
    );
}

Observable<Long> upload(UUID id) {
    //...
}

Observable<Rating> rate(UUID id) {
    //...
}
```

Отметим, однако, что метод `rate()` возвращает объект `Observable<Rating>`, который отбрасывается. На самом деле нам нужно, чтобы метод `store()` вернул этот второй `Observable<Rating>`. Но мы не можем вызвать `upload()` и `rate()` конкурентно, потому что второй завершится ошибкой, если первый еще не закончился. Решение снова дает оператор `flatMap()`, но в своей самой сложной форме:

```
upload(id)
    .flatMap(
        bytes -> Observable.empty(),
        e -> Observable.error(e),
        () -> rate(id)
    );
```

Это код следует переписать. У нас имеется `Observable<Long>`, возвращенный методом `upload()`. Для каждого прочитанного из потока числа типа `Long` мы возвращаем `Observable.empty()`, т. е. по существу отбрасываем эти события – индикация хода выполнения нас не интересует. Ошибки нас тоже не интересуют, но вместо протоколирования мы передаем их дальше подписчику. Отметим, что в наивной

реализации мы просто протоколировали ошибки, а, значит, по сути дела скрывали их. Эвристическое правило гласит: не знаешь, как обработать исключение, пусть решает начальник (т. е. вызывающий метод, родительская задача или следующий объект `Observable`). Наконец, лямбда-выражение `() -> rate(id)` реагирует на завершение потока. В этот момент мы подменяем уведомление о завершении другим объектом `Observable<Rating>`. Таким образом, даже если исходный `Observable` захочет завершиться, мы это дело проигнорируем и попросту добавим в конец еще один `Observable`. Помните, что все три обратных вызова должны возвращать объект типа `Observable<R>` с одним и тем же типом `R`.

На практике никто не заменяет `map()` и `filter()` эквивалентным `flatMap()`, чтобы не утратить понятность и производительность кода. Но чтобы вы лучше освоили синтаксис `flatMap()`, ниже приведен пример преобразования последовательности символов в код Морзе:

```
import static rx.Observable.empty;
import static rx.Observable.just;

Observable<Sound> toMorseCode(char ch) {
    switch(ch) {
        case 'a': return just(DI, DAH);
        case 'b': return just(DAH, DI, DI, DI);
        case 'c': return just(DAH, DI, DAH, DI);
        //...
        case 'p': return just(DI, DAH, DAH, DI);
        case 'r': return just(DI, DAH, DI);
        case 's': return just(DI, DI, DI);
        case 't': return just(DAH);
        //...
        default:
            return empty();
    }
}

enum Sound { DI, DAH }

//...

just('S', 'p', 'a', 'r', 't', 'a')
    .map(Character::toLowerCase)
    .flatMap(this::toMorseCode)
```

Как легко видеть, каждый символ заменяется последовательностью точек (`DI`) и тире (`DAH`). Если символ неизвестен, то возвращается пустая последовательность. Оператор `flatMap()` гарантирует, что мы получаем плоский поток знаков, а не объект типа `Observable<Observable<Sound>>`, который вернул бы простой `map()`. И попутно мы столкнулись еще с одним важным аспектом `flatMap()`: порядком событий. Лучше всего объяснить это на примере, который станет намного забавнее, если добавить оператор `delay()`.

Откладывание событий с помощью оператора `delay()`

По существу, оператор `delay()` сдвигает все полученные из предыдущего потока `Observable` события вперед во времени. Стало быть, такая простая конструкция:

```
import java.util.concurrent.TimeUnit;

just(x, y, z).delay(1, TimeUnit.SECONDS);
```

порождает `x`, `y` и `z` не сразу после подписки, а с заданной задержкой.

Мы уже встречались с оператором `timer()` в главе 2, а `delay()` очень на него похож. Мы можем заменить `delay()` комбинацией `timer()` и (сюрприз!) `flatMap()`:

```
Observable
    .timer(1, TimeUnit.SECONDS)
    .flatMap(i -> Observable.just(x, y, z))
```

Надеюсь, это понятно: `timer()` генерирует искусственное событие, которое нам совершенно не интересно. Однако `flatMap()` заменяет искусственное событие (0 в аргументе `i`) тремя порожденными значениями: `x`, `y`, `z`. В данном конкретном случае это дает тот же результат, что `just(x, y, z).delay(1, SECONDS)`, но в общем случае это не так. Оператор `delay()` полнее, чем `timer()`, поскольку сдвигает все последующие события на заданное время, тогда как `timer()` просто «спит» и порождает специальное событие спустя указанное время. Для полноты картины упомянем еще перегруженный вариант `delay()`, который может вычислять задержку для каждого события, а не глобально для всех событий. В следующем примере порождение каждой строки задерживается на время, зависящее от длины строки:

```
import static rx.Observable.timer;
import static java.util.concurrent.TimeUnit.SECONDS;

Observable
    .just("Lorem", "ipsum", "dolor", "sit", "amet",
        "consectetur", "adipiscing", "elit")
    .delay(word -> timer(word.length(), SECONDS))
    .subscribe(System.out::println);

TimeUnit.SECONDS.sleep(15);
```

Если выполнить эту программу до подписки включительно, то она немедленно завершится, не выведя никаких результатов, потому что порождение событий происходит в фоновом потоке. В главе 4 мы узнаем о типе `BlockingObservable`, который упрощает такие проверки. А пока просто поставим `sleep()` в конце. Тогда первым появится слово *sit*, а спустя секунду — *amet* и *elit*. Помните, что `delay()` можно переписать как `timer()` плюс `flatMap()`? Сами не хотите попробовать? Решение приведено ниже:

```
Observable
    .just("Lorem", "ipsum", "dolor", "sit", "amet",
```



```
"consectetur", "adipiscing", "elit")
.flatMap(word ->
  timer(word.length(), SECONDS).map(x -> word))
```

Примеры выше демонстрируют интересную особенность `flatMap()`: порядок событий не сохраняется. Зная, как работает `delay()`, мы, наконец, можем заняться этой проблемой.

Порядок событий после `flatMap()`

Что в действительности делает оператор `flatMap()`? Он принимает *главную* последовательность (поток `Observable`) значений, распределенных во времени (событий), и заменяет каждое событие независимой подпоследовательностью. В общем случае эти подпоследовательности никак не связаны между собой и с событиями, которые стали причиной их порождения из главной последовательности. Таким образом, у нас теперь нет главной последовательности, а есть набор объектов `Observable`, каждый из которых ведет свою жизнь — появляется и исчезает. Поэтому `flatMap()` не может дать *никаких* гарантий относительно того, в каком порядке эти подсобытия дойдут до следующего оператора или подписчика. Рассмотрим простой пример:

```
just(10L, 1L)
  .flatMap(x ->
    just(x).delay(x, TimeUnit.SECONDS))
  .subscribe(System.out::println);
```

В этом примере мы задерживаем событие `10L` на 10 секунд, а событие `1L` (хронологически появляющееся позже) — на 1 секунду. В результате мы увидим число 1 через секунду, а число 10 — спустя девять секунд после этого. Порядок событий в начале и в конце изменился! А может быть еще хуже: представьте, что преобразование `flatMap()` порождает несколько событий (или даже бесконечно много), распределенных в широком временном диапазоне:

```
Observable
  .just(DayOfWeek.SUNDAY, DayOfWeek.MONDAY)
  .flatMap(this::loadRecordsFor);
```

Метод `loadRecordsFor()` возвращает разные потоки в зависимости от дня недели:

```
Observable<String> loadRecordsFor(DayOfWeek dow) {
  switch(dow) {
    case SUNDAY:
      return Observable
        .interval(90, MILLISECONDS)
        .take(5)
        .map(i -> "Sun-" + i);
    case MONDAY:
      return Observable
        .interval(65, MILLISECONDS)
```

```

        .take(5)
        .map(i -> "Mon-" + i);
    //...
}
}

```

Дублирование в `loadRecordsFor()` допущено сознательно, чтобы не усложнять восприятие примера, и без того уже достаточно сложного. Посмотрим шаг за шагом, что делает этот оператор `flatMap()`. Имеется простой `Observable`, порождающий дни недели: воскресенье (`Sunday`) и сразу за ним понедельник (`Monday`). Далее мы преобразуем каждое из двух значений в подпоследовательность, которую генерирует оператор `interval()`. Напомним, что `interval()` генерирует последовательность целых чисел, начиная с нуля, с фиксированной задержкой. В данном случае задержка зависит от дня недели: 65 миллисекунд для воскресенья и 90 для понедельника. Из обеих последовательностей мы берем только первые пять элементов (`take(5)`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» ниже). В результате у нас образовалось два потока `Observable`, которые одновременно отсчитывают числа с разной частотой. И чего ждать на выходе? Самый очевидный ответ такой:

```
Sun-0, Sun-1, Sun-2, Sun-3, Sun-4, Mon-0, Mon-1, Mon-2, Mon-3, Mon-4
```

Но на самом деле у нас есть два независимо работающих потока, а их результаты надо каким образом *объединить* в один поток. Когда `flatMap()` видит на входе `Sunday`, он сразу вызывает `loadRecordsFor(Sunday)` и переправляет все порожденные этой функцией события (типа `Observable<String>`) дальше. Тем временем на входе появляется `Monday` и `flatMap()` вызывает функцию `loadRecordsFor(Monday)`. События из второго подпотока также передаются дальше, чередуясь с событиями из первого. Если бы от `flatMap()` требовалось предотвратить такое чередование, то ему пришлось бы либо буферизовать последующие подпотоки до завершения первого, либо подписываться на второй подпоток только после того, как первый завершится. И такое поведение действительно реализовано в операторе `concatMap()` (см. раздел «Сохранение порядка с помощью `concatMap()`» ниже). Но `flatMap()` поступает иначе: сразу подписывается на все подпотоки и объединяет их, отправляя события дальше по мере того, как внутренние потоки их порождают. Все подпоследовательности, возвращаемые `flatMap()` объединены и равноправны, т. е. `RxJava` передает их следующему объекту `Observable`, не отдавая предпочтения ни одной:

```
Mon-0, Sun-0, Mon-1, Sun-1, Mon-2, Mon-3, Sun-2, Mon-4, Sun-3, Sun-4
```

Внимательно просчитав задержки, мы убедимся, что этот порядок правилен. Например, хотя в исходном `Observable` первым было событие `Sunday`, на выходе первым появляется `Mon-0`, потому что подпоток, порожденный из `Monday`, начинает отправлять события раньше. По той же причине `Mon-4` появляется до `Sun-3` и `Sun-4`.

Сохранение порядка с помощью `concatMap()`

Как быть, если абсолютно необходимо сохранить исходный порядок событий? То есть события, появившиеся в результате обработки события n , должны предшествовать тем, что появились в результате обработки события $n+1$. Оказывается, есть удобный оператор `concatMap()`, который имеет точно такой же синтаксис, как `flatMap()`, но работает совершенно по-другому:

```
Observable
    .just(DayOfWeek.SUNDAY, DayOfWeek.MONDAY)
    .concatMap(this::loadRecordsFor);
```

На этот раз будет напечатано именно то, что мы и предполагали:

```
Sun-0, Sun-1, Sun-2, Sun-3, Sun-4, Mon-0, Mon-1, Mon-2, Mon-3, Mon-4
```

А что происходит под капотом? Когда в исходном потоке появляется первое событие (Sunday), `concatMap()` подписывается на объект `Observable`, возвращенный функцией `loadRecordsFor()`, и передает все порожденные им события дальше. По завершении этого внутреннего потока `concatMap()` дожидается следующего события на входе (Monday) и продолжает в том же духе. Оператор `concatMap()` не вносит никакой конкурентности, а сохраняет порядок входящих событий, избегая чередования.



Внутри `flatMap()` использует оператор `merge()`, который одновременно подписывается на все внутренние `Observable`, никак не различая их (см. раздел «Обращение с несколькими объектами `Observable`, как с одним, с помощью `merge()`» ниже). Именно поэтому исходящие события чередуются. С другой стороны, `concatMap()` технически мог бы использовать оператор `concat()` (см. раздел «Способы комбинирования потоков: `concat()`, `merge()` и `switchOnNext()`» ниже). Оператор `concat()` подписывается только на первый `Observable` и, лишь когда он завершится, переходит ко второму.

Управление уровнем конкурентности `flatMap()`

Предположим, что имеется большой список пользователей, обернутый объектом `Observable`. В классе `User` имеется метод `loadProfile()`, который загружает экземпляр `Observable<Profile>`, отправляя HTTP-запрос. Наша цель – как можно быстрее загрузить профили всех пользователей. Оператор `flatMap()` как раз для того и задумывался, чтобы производить конкурентные вычисления над входящими значениями:

```
class User {
    Observable<Profile> loadProfile() {
        // Отправить HTTP-запрос...
```

```

    }
}

class Profile { /* ... */}

//...

List<User> veryLargeList = //...
Observable<Profile> profiles = Observable
    .from(veryLargeList)
    .flatMap(User::loadProfile);

```

На первый взгляд, все отлично. Поток `Observable<User>` конструируется из фиксированного списка `List` с помощью оператора `from()`, следовательно, после подписки он почти мгновенно порождает всех пользователей. Для каждого объекта `User` вызывается оператор `flatMap()`, в результате чего `loadProfile()` возвращает `Observable<Profile>`. Затем `flatMap()` самостоятельно подписывается на все полученные объекты `Observable<Profile>` и переправляет события `Profile` дальше. Подписка на внутренний `Observable<Profile>`, скорее всего, влечет за собой создание нового HTTP-подключения. Таким образом, если имеется 10 000 пользователей, то мы чуть ли не одновременно создадим 10 000 одновременных HTTP-подключений. Если все они адресованы одному и тому же серверу, то может произойти следующее:

- запрос на подключение будет отклонен;
- длительное ожидание, завершающееся таймаутом;
- крах сервера;
- ограничение скорости передачи или занесение в черный список;
- увеличение общей задержки;
- проблемы на стороне клиента: превышение максимального числа открытых сокетов, превышение максимального числа потоков, чрезмерный расход памяти и т. д.

Повышение уровня конкурентности окупается лишь до определенного момента. Попытка запустить слишком много одновременных операций, скорее всего, приведет к чрезмерно большому числу контекстных переключений, высокому потреблению памяти и процессора и общему снижению производительности. Одно из возможных решений – замедлить работу `Observable<User>`, так чтобы он не порождал все объекты `User` сразу. Однако настройка задержки для достижения оптимального уровня конкурентности – дело неблагодарное. Есть другой путь: воспользоваться очень простым перегруженным вариантом `flatMap()`, который ограничивает общее число конкурентных подписок на внутренние потоки:

```
flatMap(User::loadProfile, 10);
```

Параметр `maxConcurrent` ограничивает число активных внутренних `Observable`. На практике это означает, что для первых 10 полученных пользователей оператор `flatMap()` вызывает метод `loadProfile()`. Но при появлении одиннадцатого⁵

⁵ На самом деле, `flatMap()` в этот момент даже не запрашивает дополнительных пользователей, подробнее этот нюанс мы объясним в разделе «Учет запрошенного объема данных» главы 6.

`flatMap()` не вызывает `loadProfile()`, а ждет завершения хотя бы одного из активных внутренних потоков. Таким образом, параметр `maxConcurrent` ограничивает число фоновых задач, запускаемых оператором `flatMap()`.

Вероятно, вы заметили, что `concatMap(f)` семантически эквивалентен вызову `flatMap(f, 1)`, т. е. `flatMap()` с параметром `maxConcurrent`, равным 1. Можно было бы посвятить еще пару страниц особенностям `flatMap()`, но нас ждут еще более занимательные операторы.

Более одного объекта Observable

Преобразовывать один `Observable` интересно, но что, если имеется несколько кооперативных объектов `Observable`? Если у вас есть опыт традиционного конкурентного программирования на Java, т. е. использования классов `Thread` и `Executor`, то вы знаете, как трудно обеспечить синхронизацию доступа к разделяемому изменяемому состоянию. По счастью, RxJava работает лучше и при таких условиях. Кроме того, библиотека предлагает единообразный способ обработки ошибок во всех операторах, работающих с несколькими потоками. Если какой-нибудь источник входящих событий порождает уведомление об ошибке, то это уведомление отправляется дальше, так что весь составной поток также завершается с ошибкой. Если ошибки порождают сразу несколько входящих `Observable`, то принимается во внимание первая, а остальные отбрасываются (любой `Observable` может порождать `onError` только один раз, см. раздел «Анатомия `rx.Observable`» главы 2). Наконец, на случай, если мы хотим продолжить обработку и порождать ошибки только после порождения всех нормальных событий, у большинства операторов имеется вариант с именем вида `*DelayError`.

Обращение с несколькими объектами *Observable*, как с одним, с помощью `merge()`

Помните, как в разделе «Обертывание с помощью `flatMap()`» выше мы говорили о методе `Observable<LicensePlate> recognize(CarPhoto photo)`, который пытался асинхронно распознать номер машины `LicensePlate` на ее фотографии `CarPhoto`? Тогда мы мельком заметили, что в таком потоке можно было бы одновременно выполнять несколько алгоритмов – одни быстрее, другие точнее. Но мы не хотим раскрывать детали алгоритмов внешнему миру, нужно лишь помочь в поток прогрессивно улучшающиеся результаты: от самого быстрого до самого точного.

Допустим, что имеется три алгоритма, написанных с учетом требований RxJava, так что каждый из них обернут объектом `Observable`. Разумеется, каждый алгоритм сам по себе может порождать от нуля до потенциально бесконечного множества результатов:

```
Observable<LicensePlate> fastAlgo(CarPhoto photo) {  
    // Быстрый, но низкого качества
```

```

}

Observable<LicensePlate> preciseAlgo(CarPhoto photo) {
    // Точный, но иногда накладный
}

Observable<LicensePlate> experimentalAlgo(CarPhoto photo) {
    // Непредсказуемый, может не завершаться
}

```

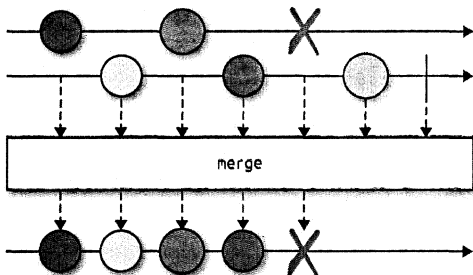
Мы хотим выполнить эти алгоритмы одновременно (подробнее о том, как в RxJava обрабатывается конкурентность, см. раздел «Декларативная подписка с помощью `subscribeOn()`» главы 4) и получить результаты как можно скорее. Нам не важно, какой именно алгоритм породил событие, мы хотим получить все и агрегировать их в единый поток. Именно это и делает оператор `merge()`:

```

Observable<LicensePlate> all = Observable.merge(
    preciseAlgo(photo),
    fastAlgo(photo),
    experimentalAlgo(photo)
);

```

Я намеренно поместил `preciseAlgo()` (предположительно, самый медленный) первым, желая подчеркнуть, что порядок объектов `Observable`, передаваемых `merge()`, в общем-то произволен. Оператор `merge()` хранит ссылки на все переданные ему `Observable` и, как только клиент подпишется на `Observable<LicensePlate> all`, он автоматически произведет подписку на все `Observable` сразу. И неважно, какой из них породит значение первым, — оно все равно будет переправлено подписчику на `all`. Конечно, оператор `merge()` соблюдает контракт Rx, гарантируя, что все события сериализованы (не перекрываются), даже если исходные потоки порождают значения одновременно. Работу `merge()` иллюстрирует следующая кашековая диаграмма:



Оператор `merge()` часто используется, когда нужно обращаться с несколькими источниками однотипных событий, как с одним⁶. Кстати, если имеется всего два

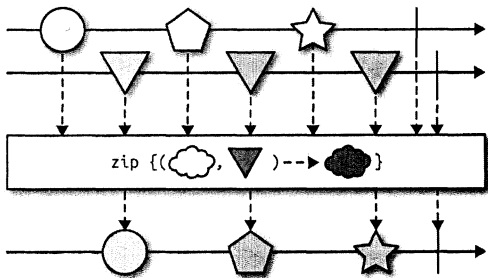
⁶ В некоторых типах вычислений эта стадия называется *соединением* (`join`).

объекта `Observable`, к которым нужно применить `merge()`, то можно воспользоваться методом экземпляра `obs1.mergeWith(obs2)`.

Имейте в виду, что ошибки, порождаемые любым из исходных `Observable`, энергично передаются подписчикам. Если нужно отложить доставку уведомлений об ошибках до завершения всех остальных потоков, то можно воспользоваться вариантом `mergeDelayError()`; этот оператор запомнит все ошибки, а не только первую, и инкапсулирует их в объект `rx.exceptions.CompositeException`.

Попарная композиция с помощью `zip()` и `zipWith()`

Под «скреплением»⁷ (*zipping*) – понимается комбинирование двух (или более) потоков таким образом, что каждое событие одного потока объединяется в пару с соответствующим ему событием другого потока. Выходной поток содержит композиции первых событий из каждого входного потока, вторых событий и т. д. Поэтому n -ое событие в выходном потоке появляется не раньше, чем каждый входной поток породит n событий. Это полезно, когда нужно объединить результаты нескольких потоков, как-то связанных между собой. Или, напротив, когда значения порождают два независимых потока, но смысл они обретают только в сочетании друг с другом. Механизм иллюстрирует следующая камешковая диаграмма:



Операторы `zip()` и `zipWith()` эквивалентны. Вторая форма используется в текущем интерфейсе для скрепления двух потоков: `s1.zipWith(s2, ...)`. Если же потоков больше двух, то имеются перегруженные варианты статического метода `zip()` в типе `Observable`, принимающие от двух до девяти аргументов:

```
Observable.zip(s1, s2, s3...)
```

У многих других операторов также есть два варианта: статический метод и метод экземпляра, например: `merge()` и `mergeWith()`. Чтобы лучше понять механизм `zip()`, представьте, что есть два независимых, но синхронизированных потока. Так,

⁷ От слова *zip* – застежка-молния. Имеется в виду метафора соединения противоположных зубцов.
Прим. перев.

в API метеорологической станции `WeatherStation` могут быть методы, которые каждую минуту возвращают результаты измерений температуры и скорости ветра, произведенных строго в одно и то же время:

```
interface WeatherStation {
    Observable<Temperature> temperature();
    Observable<Wind> wind();
}
```

Мы предполагаем, что оба объекта `Observable` порождают события в одно и то же время и, стало быть, с одинаковой частотой. При таком ограничении можно безопасно объединить оба потока, отбирая соответственные события из каждого. Это означает, что если в одном потоке появляется событие, то его нужно придержать, пока не появится событие в другом, и наоборот. Название `zip` подразумевает, что события в «левом» и «правом» потоке возникают парами. В более общем случае `zip()` может принимать до девяти потоков и порождать исходящее событие, только когда увидит все входящие.

На первый взгляд, в качестве типа значения, возвращаемого `zip()` идеально смотрелся бы кортеж (пара в частном случае двух потоков). К сожалению, в Java нет встроенных структур для пар, а у RxJava нет никаких внешних зависимостей. Можно воспользоваться реализацией класса `Pair` из библиотеки Apache Commons Lang (<http://commons.apache.org/proper/commons-lang/apidocs/org/apache/commons/lang3/tuple/Pair.html>), `Javaslang` (<http://static.javadoc.io/io.javaslang/javaslang/2.0.1/javaslang/Tuple1.html>) или Android SDK (<https://developer.android.com/reference/android/util/Pair.html>). Или сами напишите функцию или структуру данных для комбинирования пар событий:

```
class Weather {
    public Weather(Temperature temperature, Wind wind) {
        //...
    }
}

//...

Observable<Temperature> temperatureMeasurements = station.temperature();
Observable<Wind> windMeasurements = station.wind();

temperatureMeasurements
    .zipWith(windMeasurements,
        (temperature, wind) -> new Weather(temperature, wind));
```

Когда появляется новое событие `Temperature`, `zipWith()` ждет (без блокировки!) события `Wind` и наоборот. Два события передаются нашему лямбда-выражению⁸, которое составляет из них объект `Weather`. Затем все повторяется. Оператор `zip()` был описан в терминах потоков, в т. ч. бесконечных. Но часто возникает необходимость применить `zipWith()` или `zip()` к объектам `Observable`, порождающим ровно

⁸ Здесь возможен также более лаконичный синтаксис `Weather::new`.

один элемент. Обычно такие объекты возникают в результате асинхронного ответа на какой-то запрос или действие. В главе 4 мы подробно поговорим о том, как использовать RxJava в реальных приложениях.

Ну а пока изучим пример. Требуется породить *декартово произведение* множеств значений из двух потоков. Пусть имеются два объекта Observable: один соответствует горизонталям шахматной доски (*ranks*, от 1 до 8), а другой – вертикалям (*files*, от *a* до *h*). Мы хотим получить все 64 клетки на доске:

```
Observable<Integer> oneToEight = Observable.range(1, 8);
Observable<String> ranks = oneToEight
    .map(Object::toString);
Observable<String> files = oneToEight
    .map(x -> 'a' + x - 1)
    .map(ascii -> (char)ascii.intValue())
    .map(ch -> Character.toString(ch));
Observable<String> squares = files
    .flatMap(file -> ranks.map(rank -> file + rank));
```

Объект *squares* типа Observable порождает ровно 64 события: для 1 – *a1*, *a2*, ... *a8*, затем *b1*, *b2*, ... и последними – *h7* и *h8*. Это интересный пример применения *flatMap()* – для каждой вертикали (*file*) генерируются все находящиеся на ней клетки. Рассмотрим еще один, более практический пример декартова произведения. Предположим, что вы планируете провести один выходной в каком-то городе, когда на улице солнечная погода, а перелет и отели дешевы. Для этого мы скомбинируем несколько потоков и получим все возможные результаты:

```
import java.time.LocalDate;

Observable<LocalDate> nextTenDays =
    Observable
        .range(1, 10)
        .map(i -> LocalDate.now().plusDays(i));

Observable<Vacation> possibleVacations = Observable
    .just(City.Warsaw, City.London, City.Paris)
    .flatMap(city -> nextTenDays.map(date -> new Vacation(city, date)))
    .flatMap(vacation ->
        Observable.zip(
            vacation.weather().filter(Weather::isSunny),
            vacation.cheapFlightFrom(City.NewYork),
            vacation.cheapHotel(),
            (w, f, h) -> vacation
        ));
```

Вот определение класса *Vacation*:

```
class Vacation {
    private final City where;
    private final LocalDate when;

    Vacation(City where, LocalDate when) {
        this.where = where;
        this.when = when;
    }
}
```

```

    }

    public Observable<Weather> weather() {
        //...
    }

    public Observable<Flight> cheapFlightFrom(City from) {
        //...
    }

    public Observable<Hotel> cheapHotel() {
        //...
    }
}

```

Код весьма насыщенный. Сначала мы с помощью комбинации `range()` и `map()` генерируем все даты, начиная с завтрашнего дня на 10 дней вперед. Затем с помощью `flatMap()` результат комбинируется с тремя городами; оператор `zip()` нам здесь не нужен, т. к. мы хотим получить все пары (дата, город). Для каждой такой пары создается инкапсулирующий ее класс `Vacation`. Вот теперь начинается самое интересное: мы применяем `zip` к трем объектам `Observable`: `Observable<Weather>`, `Observable<Flight>` и `Observable<Hotel>`. Предполагается, что два последних возвращают единицу или ноль в зависимости от того, найден ли дешевый рейс или отель для данной комбинации города и даты. Хотя `Observable<Weather>` всегда возвращает что-то, мы применяем оператор `filter(Weather::sunny)`, чтобы отфильтровать пасмурные дни. В итоге получается операция `zip()` с тремя потоками, каждый из которых порождает ноль или один элемент. `zip()` завершается, как только завершится один из входных потоков, оставшиеся потоки при этом игнорируются. Благодаря этому свойству, если для какой-то пары данные о погоде, рейсе или отеле отсутствуют, в результирующем потоке, созданном `zip()`, данных для нее тоже не будет. Следовательно, мы получим поток всех возможных планов на выходной, удовлетворяющих требованиям.

Не удивляйтесь, что указанная в операторе `zip` функция игнорирует свои аргументы: `(w, f, h) -> vacation`. Внешний поток объектов `Vacation` содержит все возможные планы провести выходной в доступные дни. Однако для реализации плана необходимо, чтобы сложились условия: погода, дешевый рейс и свободный номер в отеле. Если все это в наличии, то мы возвращаем экземпляр `vacation`, в противном случае `zip` вообще не вызовет наше лямбда-выражение.

Когда потоки не синхронизированы: *combineLatest(), withLatestFrom() и amb()*

В разделе «Попарная композиция с помощью `zip()` и `zipWith()`» выше мы высказали смелое предположение о том, что оба объекта `Observable` порождают события с одинаковой частотой и в близкие моменты времени. Но если один поток хотя бы ненамного опережает другой, то событиям из более быстрого потока придется все дольше и дольше дожидаться отстающего. Чтобы проиллюстрировать это явление, сначала скрепим два потока, производящих события в одном темпе:

```
Observable<Long> red = Observable.interval(10, TimeUnit.MILLISECONDS);
Observable<Long> green = Observable.interval(10, TimeUnit.MILLISECONDS);

Observable.zip(
    red.timestamp(),
    green.timestamp(),
    (r, g) -> r.getTimestampMillis() - g.getTimestampMillis()
).forEach(System.out::println);
```

Объекты `red` и `green` порождают элементы с одинаковой частотой. К каждому объекту мы присоединили временную метку `timestamp()`, чтобы знать когда он был порожден.

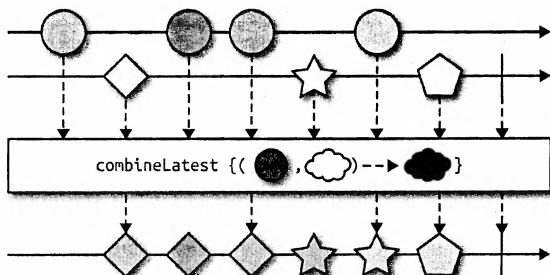


timestamp()

Оператор `timestamp()` обортывает событие произвольного типа `T` объектом класса `rx.schedulers.Timestamped<T>` с двумя атрибутами: исходное значение типа `T` и временная метка типа `long`, соответствующая моменту создания этого значения.

Преобразование, указанное в `zip()`, просто вычисляет разность моментов создания событий из каждого потока. Если потоки синхронизированы, то эта разность колеблется вокруг нуля. Но если слегка замедлить один поток, например присвоить `green` значение `Observable.interval(11, MILLISECONDS)`, то ситуация принципиально изменится. Различие во времени между `red` и `green` будет нарастать: `red` потребляется в темпе поступления, но должен ждать отстающий поток, причем время ожидания увеличивается. Со временем накапливающееся запаздывание может привести к устареванию данных и даже к утечке памяти (см. раздел «Потребление памяти и утечки» главы 8). На практике `zip()` следует использовать очень осторожно.

Что нам на самом деле нужно, так это порождение пары всякий раз, как *хотя бы один* входной поток порождает событие, при этом вторым элементом пары должно быть последнее известное значение из другого потока. Для этого полезен оператор `combineLatest()`, работу которого иллюстрирует следующая диаграмма:



Рассмотрим искусственный пример. Один поток порождает значения s_0, s_1, s_2 с интервалом 17 миллисекунд, а другой – значения f_0, f_1, f_2 с интервалом 10 миллисекунд (заметно быстрее):

```
import static java.util.concurrent.TimeUnit.MILLISECONDS;
import static rx.Observable.interval;

Observable.combineLatest(
    interval(17, MILLISECONDS).map(x -> "S" + x),
    interval(10, MILLISECONDS).map(x -> "F" + x),
    (s, f) -> f + ":" + s
).forEach(System.out::println);
```

Мы объединяем оба потока: порождаем новое значение всякий раз, как приходит событие от какого-то потока. Синхронизация быстро утрачивается, но, по крайней мере, значения потребляются в режиме реального времени, и более быстрый поток не должен дожидаться отставшего напарника:

```
F0:S0
F1:S0
F2:S0
F2:S1
F3:S1
F4:S1
F4:S2
F5:S2
F5:S3
...
F998:S586
F998:S587
F999:S587
F1000:S587
F1000:S588
F1001:S588
```

Отметим, что при каждом появлении нового события f в выходной поток отправляется очередное событие: $f_0:s_0, f_1:s_0, f_2:s_0$. RxJava замечает новое событие в более быстром потоке, присоединяет к нему последнее событие, прочитанное из медленного потока (но по-прежнему должна дожидаться появления какого-то события!), – в данном случае s_0 – и порождает новую пару. Но ни одному потоку не отдается особое предпочтение: когда появляется событие s_1 из медленного потока, к нему точно также присоединяется последнее известное событие из быстрого потока (f_2). Спустя примерно 10 секунд мы видим событие $F1000:S588$. Все сходится: за 10 секунд быстрый поток породил примерно 1000 событий, а медленный – только 588 (10 секунд, поделенные на 17 миллисекунд).

Оператор withLatestFrom()

Оператор `combineLatest` симметричен, т. е. не различает комбинируемые потоки. Но иногда требуется порождать выходное событие всякий раз, как что-то появляется в одном потоке (присоединив к нему последнее известное значение из другого), и не порождать, если событие появляется в другом потоке. Иными

словами, события из второго потока не приводят к отправке события в выходной поток, а используются лишь в качестве дополнения к событиям из первого потока. Такое поведение обеспечивает оператор `withLatestFrom()`. Продемонстрируем его на тех же потоках `slow` и `fast`:

```
Observable<String> fast = interval(10, MILLISECONDS).map(x -> "F" + x);
Observable<String> slow = interval(17, MILLISECONDS).map(x -> "S" + x);
slow
    .withLatestFrom(fast, (s, f) -> s + ":" + f)
    .forEach(System.out::println);
```

Здесь `slow` — главный поток, т. е. результирующий Observable порождает событие при *каждом* появлении события в `slow` при условии, что `fast` породил к этому моменту хотя бы одно событие. Наоборот, `fast` играет лишь вспомогательную роль и используется лишь в том случае, когда что-то пришло из потока `slow`. Функция, переданная в качестве второго аргумента `withLatestFrom()`, комбинирует каждое новое значение из `slow` с последним значением из `fast`. Однако новые значения из `fast` дальше не передаются, они лишь запоминаются внутри и добавляются к событиям из `slow`, когда те появляются. В распечатке мы видим, что каждое событие из `slow` встречается ровно один раз, тогда как некоторые события из `fast` пропущены:

```
S0:F1
S1:F2
S2:F4
S3:F5
S4:F7
S5:F9
S6:F11
...
```

Все события из потока `slow`, поступившие до первого события из `fast`, молча отбрасываются, потому что их не с чем комбинировать. Так и задумано, но если вам остро необходимо сохранить все события из главного потока, то позаботьтесь о том, чтобы другой поток породил хотя бы фиктивное событие как можно скорее. В примере ниже поток `fast` искусственно замедлен путем сдвига всех событий на 100 миллисекунд вперед (см. раздел «Откладывание событий с помощью оператора `delay()`» выше). Не будь фиктивного события, мы потеряли бы несколько событий из потока `slow`, но благодаря оператору `startWith()` мы создаем новый объект observable на основе `fast`. Он сразу же порождает событие "FX", а затем передает все события исходного потока `fast`:

```
Observable<String> fast = interval(10, MILLISECONDS)
    .map(x -> "F" + x)
    .delay(100, MILLISECONDS)
    .startWith("FX");

Observable<String> slow = interval(17, MILLISECONDS).map(x -> "S" + x);
slow
    .withLatestFrom(fast, (s, f) -> s + ":" + f)
    .forEach(System.out::println);
```

Как показывает распечатка, ни одно событие из `slow` не пропало. Но в начале мы видим несколько фиктивных событий "FX", и лишь через 100 миллисекунд появляется первое реальное событие "F1":

```
S0:FX
S1:FX
S2:FX
S3:FX
S4:FX
S5:FX
S6:F1
S7:F3
S8:F4
S9:F6
...
```

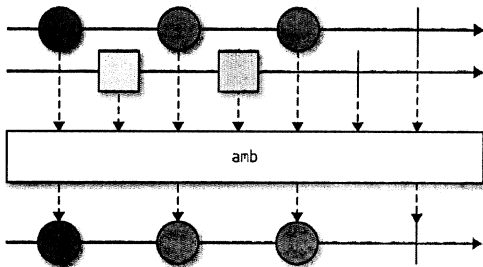
Оператор `startWith()` возвращает объект `Observable`, который сразу после подписки порождает некоторое постоянное значение (например, "FX"), а за ним – все события из исходного `Observable`. Например, показанный ниже код порождает значений 0, 1 и 2 именно в таком порядке:

```
Observable
    .just(1, 2)
    .startWith(0)
    .subscribe(System.out::println);
```

В разделе «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» ниже приведены примеры использования похожего оператора `concat()`.

Оператор `amb()`

И последний простенький оператор, который иногда оказывается полезным, – `amb()` (и родственный ему `ambWith()`). Он подписывается на все входные объекты `Observable`, которые контролирует, и ждет, когда появится самый первый элемент. Как только какой-то `Observable` породит первое событие, `amb()` отбрасывает все остальные потоки и переправляет дальше только события из того первого `Observable`, который пробудил его к жизни (см. диаграмму ниже).



В следующем примере показана работа `amb()` с двумя потоками. Обратите внимание на параметр `initialDelay`, который управляет тем, какой `Observable` первым начинает порождать события:

```
Observable<String> stream(int initialDelay, int interval, String name) {
    return Observable
        .interval(initialDelay, interval, MILLISECONDS)
        .map(x -> name + x)
        .doOnSubscribe(() ->
            log.info("Подписка на " + name))
        .doOnUnsubscribe(() ->
            log.info("Отписка от " + name));
}

//...

Observable.amb(
    stream(100, 17, "S"),
    stream(200, 10, "F")
).subscribe(log::info);
```

Можно написать эквивалентную программу с использованием нестатического метода `ambWith()`, но она менее понятна, потому что скрывает симметрию `amb()`. Кажется, будто второй каким-то образом обертывает первый, хотя на самом деле оба они равноправны:

```
stream(100, 17, "S")
    .ambWith(stream(200, 10, "F"))
    .subscribe(log::info);
```

Так или иначе, оба варианта дают одинаковые результаты. Поток `slow` порождает события реже, но первое событие в нем появляется через 100 мс, тогда как первое событие в потоке `fast` порождается спустя 200 мс. Оператор `amb()` первым делом подписывается на оба потока, а обнаружив первое событие в потоке `slow`, сразу же отписывается от второго потока и переправляет все события только из первого:

```
14:46:13.334: Подписка на S
14:46:13.341: Подписка на F
14:46:13.439: Отписка от F
14:46:13.442: S0
14:46:13.456: S1
14:46:13.473: S2
14:46:13.490: S3
14:46:13.507: S4
14:46:13.525: S5
```

Обратные вызовы `doOnSubscribe()` и `doOnUnsubscribe()` полезны для отладки. Обратите внимание, что отписка от `F` происходит спустя примерно 100 мс после подписки на `S`; это именно тот момент, когда появилось первое событие из потока `S`. После этого прослушивать события из `F` уже не имеет смысла.

Более сложные операторы: `collect()`, `reduce()`, `scan()`, `distinct()` и `groupBy()`

Некоторые операторы производят более сложные преобразования, например, рассматривают последовательность и попутно выполняют агрегирование, скажем, вычисляют скользящее среднее. Есть даже операторы с внутренним состоянием, которое изменяется по мере продвижения вдоль последовательности. Именно так работает оператор `distinct`, который запоминает встретившиеся значения и отбрасывает повторяющиеся.

Просмотр последовательности с помощью `Scan` и `Reduce`

Все операторы, которые мы рассматривали до сих пор, применялись к одному событию (например, фильтрация отображение и скрепление). Но иногда требуется агрегировать события, чтобы сократить или упростить исходный поток. Например, рассмотрим объект `Observable<Long>`, который отслеживает ход передачи данных. При отправке каждого блока данных порождается значение типа `Long`, содержащее размер блока. Вещь полезная, но на самом-то деле нам хочется знать, сколько всего передано байтов. Ни в коем случае не следует хранить и модифицировать глобальное состояние внутри оператора:

```
import java.util.concurrent.atomic.LongAdder;

// НЕПРАВИЛЬНО!
Observable<Long> progress = transferFile();

LongAdder total = new LongAdder();
progress.subscribe(total::add);
```

Этот код может приводить к очень неприятным ошибкам из-за конкурентности, как всегда в случае разделяемого состояния. Лямбда-выражения внутри операторов могут выполняться в произвольном потоке, поэтому глобальное состояние должно быть потокобезопасным. И, кроме того, нужно принять во внимание ленивость. RxJava стремится минимизировать использование глобального изменяемого состояния, предоставляя операторы, допускающие композицию. Модификация глобального состояния – сложное дело, даже при наличии гарантий Rx. Более того, мы больше не можем использовать операторы Rx для последующей композиции `total` – например, для периодического обновления пользовательского интерфейса. Уведомление о завершении передачи также усложняется. На самом деле, нам нужно суммировать размеры блоков данных и сообщать текущий итог при появлении каждого нового блока. Вот как должен выглядеть гипотетический поток:

```
Observable<Long> progress = // [10, 14, 12, 13, 14, 16]
Observable<Long> totalProgress = /* [10, 24, 36, 49, 63, 79]
```



```
10
10+14=24
  24+12=36
    36+13=49
      49+14=63
        63+16=79
```

*/

Первый элемент передается как есть (10). Но прежде чем передавать второй элемент (14) дальше, мы складываем его с ранее переданным элементом (10) и передаем сумму (24). Третий элемент (12) снова складывается с предыдущим результатом (24), поэтому дальше передается 36. Этот процесс продолжается, пока не завершится входной поток. Последний элемент, отправленный в выходной поток, равен сумме всех полученных на входе. Этот сравнительно сложный алгоритм легко реализовать с помощью оператора `scan()`:

```
Observable<Long> totalProgress = progress
    .scan((total, chunk) -> total + chunk);
```

`scan()` принимает два параметра: последнее сгенерированное значение (*аккумулятор*) и текущее значение из исходного `Observable`. На первой итерации `total` совпадает с первым элементом из `progress`, а на второй оказывается равным результату предыдущего вызова `scan()`. Это показано в табл. 3.1.

Таблица 3.1. В каждой строке представлена одна итерация `scan()`

progress	total	chunk	totalProgress
10	—	—	—
14	10	14	24
12	24	12	36
13	36	13	49
14	49	14	63
16	63	16	79

`scan()` работает, как бульдозер, который продвигается по входному потоку `Observable` и сгребает все элементы. В перегруженном варианте `scan()` можно указать начальное значение (если оно отличается от первого элемента):

```
Observable<BigInteger> factorials = Observable
    .range(2, 100)
    .scan(BigInteger.ONE, (big, cur) ->
        big.multiply(BigInteger.valueOf(cur)));
```

Объект `factorials` генерирует числа 1, 2, 6, 24, 120, 720 и т. д. Отметим, что входящий `Observable` начинается с 2, а исходящий — с 1, т. е. с заданного нами начального значения (`BigInteger.ONE`). Как правило, тип результирующего `Observable` совпадает с типом аккумулятора. Поэтому, если явно не задано начальное значение аккумулятора, то тип `T`, параметризующий выходной `Observable`, не изменя-

ется. В противном случае (как в примере `factorials` выше) результат имеет тип `Observable<BigInteger>`, поскольку начальное значение имело тип `BigInteger`. По-настоящему, что этот тип не должен изменяться в процессе просмотра.

Иногда промежуточные результаты нас не интересуют, а нужен только конечный. Например, мы хотим знать общее число переданных байтов, а не изменение в процессе передачи. Или собрать все значения в какой-то изменяемой структуре данных, например `ArrayList`, добавляя их по одному. Для этой цели предназначен оператор `reduce()`. Но есть очевидный подвод: если последовательность бесконечна, то `scan()` будет продолжать порождение исходящих событий для каждого входящего, тогда как `reduce()` не породит вообще ни одного события. Допустим, имеется источник объектов `CashTransfer` (перевод наличных) с методом `getAmount()`, который возвращает значение типа `BigDecimal`. Требуется вычислить сумму всех переводов. Следующие два преобразования эквивалентны. Они перебирают все переводы и суммируют их величины, считая `ZERO` начальным значением суммы:

```
Observable<CashTransfer> transfers = //...;

Observable<BigDecimal> total1 = transfers
    .reduce(BigDecimal.ZERO,
        (totalSoFar, transfer) ->
            totalSoFar.add(transfer.getAmount()));

Observable<BigDecimal> total2 = transfers
    .map(CashTransfer::getAmount)
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```

Оба преобразования дают одинаковый результат, но второе выглядит проще, несмотря на то, что в нем два шага. Это еще одна причина предпочесть композицию нескольких небольших преобразований одному, но громоздкому. Кроме того, вы, вероятно, заметили, что `reduce()` – это, по существу `scan()`, который оставляет только последний элемент. И реализовать этот оператор можно было бы так:

```
public <R> Observable<R> reduce(
    R initialValue,
    Func2<R, T, R> accumulator) {
    return scan(initialValue, accumulator).takeLast(1);
}
```

Как видим, `reduce()` просто сканирует `Observable`, но отбрасывает все элементы, кроме последнего (см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» ниже).

Редукция с помощью изменяемого аккумулятора: `collect()`

Теперь давайте преобразуем конечный поток событий типа `T` в поток, содержащий единственное событие типа `List<T>`. Разумеется, это событие порождается, когда входной поток `Observable<T>` завершается:

```
Observable<List<Integer>> all = Observable
    .range(10, 20)
    .reduce(new ArrayList<>(), (list, item) -> {
        list.add(item);
        return list;
    });
```

В этом примере в начале выполнения `reduce()` имеется пустой массив `ArrayList<Integer>` (аккумулятор), а на каждой итерации в него добавляется порожденный элемент. Лямбда-выражение, отвечающее за редукцию (аккумуляцию) должно вернуть новую версию аккумулятора. Увы, метод `List.add()` возвращает не список `List`, а значение типа `boolean`, поэтому требуется явное предложение `return`. Чтобы избавиться от многословия, можно воспользоваться оператором `collect()`. Работает он почти так же, как `reduce()`, но предполагает, что для каждого события используется один и тот же изменяемый аккумулятор, а не создается новый неизменяемый (сравните с неизменяемым типом `BigInteger` в примере выше):

```
Observable<List<Integer>> all = Observable
    .range(10, 20)
    .collect(ArrayList::new, List::add);
```

Еще одно полезное применение `collect()` — агрегирование всех событий в один объект `StringBuilder`. В этом случае аккумулятором является пустой `StringBuilder`, а операция добавляет к нему элемент:

```
Observable<String> str = Observable
    .range(1, 10)
    .collect(
        StringBuilder::new,
        (sb, x) -> sb.append(x).append(", ")
    )
    .map(StringBuilder::toString);
```

Как и любой оператор над `Observable`, `reduce()` и `collect()` неблокирующие, поэтому итоговый список `List<Integer>`, содержащий все числа, порожденные объектом `Observable.range(10, 20)`, появится только после уведомления о том, что входной поток завершился; исключения распространяются, как обычно. Преобразование `Observable<T>` в `Observable<List<T>>` встречается настолько часто, что для него имеется встроенный оператор `toList()`. Практические примеры см. в разделе «`BlockingObservable`: выход из реактивного мира» главы 4.

Проверка того, что `Observable` содержит ровно один элемент, с помощью `single()`

Кстати говоря, некоторые объекты `Observable`, по определению, должны порождать только одно значение. Например, в примере выше всегда порождается один список `List<Integer>`, пусть даже пустой. В таком случае имеет смысл применить оператор `single()`. Он не изменяет входной `observable`, но проверяет, что тот по-

родил ровно одно событие. Если это не так, то вместо ожидаемого результата мы получим исключение.

Устранение дубликатов с помощью `distinct()` и `distinctUntilChanged()`

Иногда бывает полезен бесконечный поток случайных величин, обычно он комбинируется с другими потоками. Следующий `Observable` порождает псевдослучайные числа типа `Integer` в диапазоне от 0 до 1000, не включая верхнюю границу:

```
Observable<Integer> randomInts = Observable.create(subscriber -> {
    Random random = new Random();
    while (!subscriber.isUnsubscribed()) {
        subscriber.onNext(random.nextInt(1000));
    }
});
```

Ясно, что иногда встречаются дубликаты, а если применить оператор `take(1001)`, то хотя бы один дубликат будет наверняка⁹. Но что, если мы хотим взять только 10 наименьших *уникальных* случайных чисел? Встроенный оператор `distinct()` отбрасывает из входного потока уже встречавшиеся ранее значения, передавая дальше только уникальные:

```
Observable<Integer> uniqueRandomInts = randomInts
    .distinct()
    .take(10);
```

Всякий раз как из входного потока `Observable (randomInts)` поступает новое значение, `distinct()` проверяет, встречалось ли оно раньше. Для сравнения используются методы `equals()` и `hashCode()`, поэтому позаботьтесь об их реализации в соответствии с рекомендациями по программированию на Java (у двух равных объектов хэш-коды *обязательно* должны совпадать). Интересно, что `take(1001)` в конечном итоге породит все значения от 0 до 999 в случайном порядке, но никогда не завершится, потому что между 0 и 999 не существует 1001-го целого числа, которое не совпадало бы ни с одним из предыдущих.

В разделе «Практический пример: от API обратных вызовов к потоку `Observable`» главы 2 мы рассматривали объект `Observable<twitter4j.Status>`, который порождает обновления состояния в социальной сети Твиттер. Всякий раз как пользователь отправляет твит, этот `Observable` порождает новое событие. У объекта `Status` имеется несколько методов доступа, например `getText()`, `getUser()` и т. д. Оператор `distinct()` не имеет смысла для событий `Status`, поскольку дубликаты в этом случае попросту невозможны. Но что, если нас интересует только текст самого первого твита для каждого пользователя (идентификатор пользователя, имеющий тип `long`, возвращает метод `status.getUser().getId()`)? Очевидно, что можно извлечь нужное свойство и выполнить для него `distinct()`:

⁹ `nextInt(1000)` порождает лишь 1000 различных результатов.

```
Observable<Status> tweets = //...

Observable<Long> distinctUserIds = tweets
    .map(status -> status.getUser().getId())
    .distinct();
```

Увы, к тому моменту, когда вызывается `distinct()`, исходного объекта `Status` уже нет. Нам необходимо как-то извлечь свойство события, служащее для установления уникальности. Два события считаются равными (и второе отбрасывается), если это свойство (будем называть его *ключом*) уже встречалось ранее:

```
Observable<Status> distinctUserIds = tweets
    .distinct(status -> status.getUser().getId());
```

Ключи (что бы ни использовалось в этом качестве) сравниваются с помощью методов `equals()` и `hashCode()` с ранее встречавшимися ключами. Не забывайте, что `distinct()` должен вечно хранить все события (или ключи), которые когда-либо видел (см. раздел «Потребление памяти и утечки» главы 8).

Оператор `distinct()` полезен, если мы хотим обработать уникальные события только один раз.

На практике нередко разумнее пользоваться оператором `distinctUntilChanged()`. Он отбрасывает событие, только если оно совпадает с предыдущим (по умолчанию для сравнения используется метод `equals()`). `distinctUntilChanged()` особенно удобен, когда подается стационарный поток измерений, а мы хотим получать уведомление только тогда, когда результат измерения изменился. В разделе «Попарная композиция с помощью `zip()` и `zipWith()`» выше мы экспериментировали с потоком `Observable<Weather>`, где класс `Weather` имеет два атрибута: `Temperature` и `Wind`. Новое событие `Weather` появляется раз в минуту, но погода изменяется не так часто, поэтому хорошо бы отбросить повторяющиеся события и сосредоточиться только на изменениях:

```
Observable<Weather> measurements = //...

Observable<Weather> tempChanges = measurements
    .distinctUntilChanged(Weather::getTemperature);
```

Здесь событие `Weather` порождается только при изменении температуры (изменения скорости ветра в расчет не принимаются). Понятно, что если мы хотим порождать событие всякий раз, как изменяется либо `Temperature`, либо `Wind`, то прекрасно подошел бы оператор `distinctUntilChanged()` без параметров в предположении, что в классе `Weather` реализован метод `equals()`. Важное различие между `distinct()` и `distinctUntilChanged()` состоит в том, что последний может порождать дубликаты, но только если они разделены каким-то другим значением. Так, в течение суток может неоднократно встречаться одно и то же значение температуры, но в промежутках будет то теплее, то холоднее. Поэтому `distinctUntilChanged()` должен помнить только последнее встретившееся значение — в отличие от `distinct()`, который должен хранить все уникальные значения, встретившиеся с начала по-

тока. Это означает, что `distinctUntilChanged()` потребляет предсказуемый постоянный объем памяти, чего нельзя сказать о `distinct()`.

Выборка с помощью операторов `skip()`, `takeWhile()` и прочих

Никто не заставляет нас читать поток целиком, особенно если речь идет о горячих бесконечных `Observable` (хотя ими дело не ограничивается). На практике очень часто производят выборку из потока, содержащую лишь его малое подмножество. Для большинства операторов в этом разделе приводятся примеры, только если они не следуют принципу наименьшего удивления. Но операторы типа `take` и `last` настолько полезны, что опустить их просто невозможно. Ниже приведен неполный перечень таких операторов:

`take(n)` и `skip(n)`

Оператор `take(n)` обрывает выходной поток, получив первые `n` событий из входного потока, после чего отписывается от него (он может завершиться и раньше, если во входном потоке оказалось меньше `n` событий). Поведение `skip(n)` в точности противоположно; он отбрасывает первые `n` событий и начинает повторять события из входного потока, начиная с `(n+1)`-го. Оба оператора либерально относятся к своим аргументам: отрицательные числа рассматриваются как 0, а превышение размера `Observable` не считается ошибкой:

```
Observable.range(1, 5).take(3); // [1, 2, 3]
Observable.range(1, 5).skip(3); // [4, 5]
Observable.range(1, 5).skip(5); // []
```

`takeLast(n)` и `skipLast(n)`

Еще одна пара операторов, не требующая особых пояснений. `takeLast(n)` отдает только последние `n` значений из входного потока. Этот оператор хранит во внутреннем буфере последние `n` полученных значений и, получив уведомление о завершении, сразу отправляет весь буфер. Применять `takeLast()` к бесконечному потоку не имеет смысла, потому что никогда ничего не отдаст – раз поток не заканчивается, то у него нет *последних событий*. С другой стороны, оператор `skipLast(n)` отдает все значения из входного потока, кроме `n` последних. Это означает, что он отдает первое значение из входного потока только после получения `n+1` значений, второе – после получения `n+2` значений и т. д.

```
Observable.range(1, 5).takeLast(2); // [4, 5]
Observable.range(1, 5).skipLast(2); // [1, 2, 3]
```

`first()` и `last()`

Операторы `first()` и `last()` без параметров можно реализовать в виде `take(1).single()` и `takeLast(1).single()` соответственно, и это объясняет их поведение. Дополнительный оператор `single()` гарантирует, что выход-

ной поток содержит ровно одно событие или завершается исключением. У операторов `first()` и `last()` имеются также перегруженные варианты, принимающие предикаты. Тогда возвращается не самое первое (последнее) значение, а первое (последнее) из тех, что удовлетворяют заданному условию.

`takeFirst(predicate)`

Оператор `takeFirst(predicate)` можно выразить в виде `filter(predicate).take(1)`. Единственное его отличие от `first(predicate)` состоит в том, что в случае отсутствия подходящих значений он не завершается исключением `NoSuchElementException`.

`takeUntil(predicate)` и `takeWhile(predicate)`

Операторы `takeUntil(predicate)` и `takeWhile(predicate)` являются близкими родственниками. `takeUntil()` переправляет значения из исходного потока, но завершается и отписывается после отправки первого же значения, *удовлетворяющего* заданному предикату. Наоборот, `takeWhile()` продолжает переправлять значения, пока они удовлетворяют предикату. То есть разница в том, что `takeUntil()` передает первое не удовлетворяющее предикату значение, а `takeWhile()` — нет. Эти операторы важны тем, что позволяют условно отписаться от `Observable` в зависимости от порождаемых им событий. Иначе оператору пришлось бы как-то взаимодействовать с экземпляром `Subscription` (см. раздел «Управление прослушивателями с помощью типов `Subscription` и `Subscriber<T>`» главы 2), который недоступен в момент вызова оператора.

```
Observable.range(1, 5).takeUntil(x -> x == 3); // [1, 2, 3]
Observable.range(1, 5).takeWhile(x -> x != 3); // [1, 2]
```

`elementAt(n)`

Хотя выборка элемента по индексу встречается нечасто, для нее имеется встроенный оператор `elementAt(n)`. Он может завершиться исключением `IndexOutOfBoundsException`, если во входном потоке нет достаточного числа элементов или если задан отрицательный индекс. Разумеется, оператор возвращает объект `Observable<T>` с тем же типом `T`, что и у входного потока.

Операторы ...OrDefault()

Многие рассмотренные в этом разделе операторы могут возбуждать исключения. Например, `first()` поступает так, если входной поток пуст. На такой случай предусмотрены операторы вида `...OrDefault`, которые заменяют исключение значением по умолчанию. Ни один из них не нуждается в объяснениях: `elementAtOrDefault()`, `firstOrDefault()`, `lastOrDefault()`, `singleOrDefault()`.

`count()`

Оператор `count()` вычисляет, сколько событий породил входной `Observable`. Кстати, узнать, сколько порождено событий, удовлетворяющих заданному

предикату, можно с помощью идиомы `filter(predicate).count()`. Не волнуйтесь, все операторы ленивые, поэтому эта конструкция работает даже для очень больших потоков. Понятно, что `count()` не порождает никакого значения для бесконечного потока. Реализовать `count()` легко с помощью `reduce()`:

```
Observable<Integer> size = Observable
    .just('A', 'B', 'C', 'D')
    .reduce(0, (sizeSoFar, ch) -> sizeSoFar + 1);
```

```
all(predicate), exists(predicate), contains(value)
```

Иногда важно знать, что все события, порожденные данным `Observable`, удовлетворяют некоторому предикату. Оператор `all(predicate)` порождает `true`, когда входной поток завершается и все значения удовлетворяют предикату. А `false` порождается, как только обнаружено первое не удовлетворяющее предикату значение. Оператор `exists(predicate)` – прямая противоположность `all()`; он порождает `true`, как только обнаружено первое подходящее значение, и `false` – если поток завершился, и ни одного подходящего значения не встретилось. Часто предикат, передаваемый `exists()`, просто сравнивает входящие значения с константами. На такой случай имеется оператор `contains()`:

```
Observable<Integer> numbers = Observable.range(1, 5);
```

```
numbers.all(x -> x != 4); // [false]
numbers.exists(x -> x == 4); // [true]
numbers.contains(4); // [true]
```

Способы комбинирования потоков: `concat()`, `merge()` и `switchOnNext()`

Оператор `concat()` (и метод экземпляра `concatWith()`) позволяет соединить два объекта `Observable`: когда первый завершается, `concat()` подписывается на второй. Важно, что `concat()` подписывается на второй `Observable` тогда и только тогда, когда первый завершился (см. также раздел «Сохранение порядка с помощью `concatMap()`» выше). Оператор `concat()` может даже работать с одним и тем же входным `Observable`, к которому применяются разные операторы. Например, чтобы получить только несколько первых и несколько последних элементов из очень длинного потока, можно поступить так:

```
Observable<Data> veryLong = //...
final Observable<Data> ends = Observable.concat(
    veryLong.take(5),
    veryLong.takeLast(5)
);
```

Имейте в виду, что этот код подписывается на `veryLong` дважды, что может быть нежелательно. Другой пример применения `concat()` – предоставление значения по умолчанию в случае, когда первый поток не порождает ничего:


```
Observable<Car> fromCache = loadFromCache();
Observable<Car> fromDb = loadFromDb();

Observable<Car> found = Observable
    .concat(fromCache, fromDb)
    .first();
```

Объекты `Observable` ленивые, поэтому ни `loadFromCache()`, ни `loadFromDb()` еще не загружают данные. Метод `loadFromCache()` может завершиться без порождения событий, если кэш пуст, но `loadFromDb()` всегда порождает один объект `Car`. Композиция `concat()` и `first()` сначала подписывается `fromCache`, и если тот порождает один элемент, то `concat()` не подписывается на `fromDb`. Если же поток `fromCache` пуст, то `concat()` подписывается на `fromDb` и загружает данные из базы.

Оператор `concat()` тесно связан с `merge()` и `switchMap()`. Он работает как конкатенация обычных списков `List<T>`: сначала выбирает все элементы из первого потока, а когда тот завершается, переходит ко второму потоку. Конечно, как и все рассмотренные до сих пор операторы, `concat()` является неблокирующим, он порождает события только тогда, когда что-то порождает входной поток. Сравним теперь `concat()` с `merge()` (см. раздел «Обращение с несколькими объектами `Observable`, как с одним, с помощью `merge()`» выше) и с еще одним оператором — `switchOnNext()`.

Рассмотрим группу людей, у каждого из которых есть микрофон. Микрофоны моделируются объектом `Observable<String>`, а каждое событие представляет одно слово. Понятно, что события упорядочены во времени и возникают, когда люди произносят слова. Чтобы смоделировать это поведение, сконструируем простой объект `observable`, который, хоть и служит для демонстрации, интересен сам по себе:

```
Observable<String> speak(String quote, long millisPerChar) {
    String[] tokens = quote.replaceAll("[: ,]", "").split(" ");
    Observable<String> words = Observable.from(tokens);
    Observable<Long> absoluteDelay = words
        .map(String::length)
        .map(len -> len * millisPerChar)
        .scan((total, current) -> total + current);
    return words
        .zipWith(absoluteDelay.startWith(0L), Pair::of)
        .flatMap(pair -> just(pair.getLeft()))
        .delay(pair.getRight(), MILLISECONDS));
}
```

Это довольно сложный код, поэтому разберем его строчка за строчкой. Мы получаем произвольный текст в строке `String` и с помощью регулярного выражения разбиваем его на слова, удаляя знаки препинания. Затем для каждого слова вычисляем, сколько времени требуется на его произнесения, для чего просто умножаем длину слова на `millisPerChar`. Далее требуется разнести слова во времени, так чтобы каждое появлялось в результирующем потоке с задержкой, вычисленной ранее. Очевидно, что простого оператора `from` тут недостаточно:

```
Observable<String> words = Observable.from(tokens);
```

Мы хотим, чтобы слову предшествовала задержка, зависящая от длины предыдущего слова. В первой, наивной, попытке задержка просто формируется для каждого слова, исходя из его длины:

```
words.flatMap(word -> Observable
    .just(word)
    .delay(word.length() * millisPerChar, MILLISECONDS));
```

Это неправильное решение. Такой `Observable` сначала породит все однобуквенные слова одновременно. Затем, спустя некоторое время, все двухбуквенные слова, затем все трехбуквенные. Мы же хотим, чтобы сразу было порождено первое слово, а затем второе – с задержкой, зависящей от длины первого слова. Кажется ужасно сложно, а оказывается на удивление изящно. Сначала создадим по `words` вспомогательный поток, который содержит только относительные задержки, индуцированные каждым словом:

```
words
    .map(String::length)
    .map(len -> len * millisPerChar);
```

В предположении, что `millisPerChar` равно 100 и что `words` состоит из слов *Though this be madness*¹⁰, получаем такой поток: 600, 400, 200, 700. Если просто задержать каждое слово на это время, то «*be*» появится первым, а остальные слова тоже будут переставлены. Нам же нужны частичные суммы задержек: 600, 600 + 400 = 1000, 1000 + 200 = 1200, 1200 + 700 = 1900. Это легко сделать с помощью оператора `scan()` (см. раздел «Просмотр последовательности с помощью `Scan` и `Reduce`» выше).

```
Observable<Long> absoluteDelay = words
    .map(String::length)
    .map(len -> len * millisPerChar)
    .scan((total, current) -> total + current);
```

Теперь, имея последовательность слов и последовательность абсолютных задержек для каждого слова, оба потока можно скрепить. Это именно та ситуация, в которой `zip()` проявляет себя во всем блеске:

```
words
    .zipWith(absoluteDelay.startWith(0L), Pair::of)
    .flatMap(pair -> just(pair.getLeft()))
```

Здесь применение `zip()` вполне оправдано, потому что потоки заведомо одного размера и точно синхронизированы. Э-э-э... почти. Мы не хотим никакой задержки перед первым словом. Длина первого слова должна влиять на задержку второго, сумма длин первого и второго слова – на задержку третьего и т. д. Такого сдвига легко добиться, просто добавив в начало потока `absoluteDelay` значение 0:

¹⁰ Начало известной фразы Полония из «Гамлета»: «Если это безумие, то в нем есть система». – Прим. перевод.

```
import org.apache.commons.lang3.tuple.Pair;

words
  .zipWith(absoluteDelay.startWith(0L), Pair::of)
  .flatMap(pair -> just(pair.getLeft()))
  .delay(pair.getRight(), MILLISECONDS));
```

Мы строим последовательность пар (слово, абсолютная задержка) с гарантией отсутствия задержки для первого слова. Пары могут выглядеть так:

```
(Though, 0)
(this, 600)
(be, 1000)
(madness, 1200)
...
```

Это временная разметка речи, где каждое слово сопровождается точкой на временной оси. Осталось преобразовать каждую пару в одноэлементный поток `Observable`, сдвинутый во времени:

```
flatMap(pair -> just(pair.getLeft()))
  .delay(pair.getRight(), MILLISECONDS));
```

После такой длительной подготовки можно, наконец, разобраться, чем отличаются `concat()`, `merge()` и `switchOnNext()`. Предположим, что три человека цитируют шекспировского «Гамлета»:

```
Observable<String> alice = speak(
  "To be, or not to be: that is the question", 110);

Observable<String> bob = speak(
  "Though this be madness, yet there is method in't", 90);

Observable<String> jane = speak(
  "There are more things in Heaven and Earth, " +
  "Horatio, than are dreamt of in your philosophy", 100);
```

Как видим, для каждого человека задан свой темп речи, измеряемый в миллисекундах на одну букву (`millisPerChar`). Что случится, если все трое будут говорить одновременно? RxJava может ответить на этот вопрос:

```
Observable
  .merge(
    alice.map(w -> "Alice: " + w),
    bob.map(w -> "Bob: " + w),
    jane.map(w -> "Jane: " + w)
  )
  .subscribe(System.out::println);
```

Результат выглядит хаотически, слова, произносимые разными людьми, перемежаются. Мы услышим только шум, который почти невозможно разобрать:

Alice: To
Bob: Though
Jane: There
Alice: be
Alice: or
Jane: are
Alice: not
Bob: this
Jane: more
Alice: to
Jane: things
Alice: be
Bob: be
Alice: that
Bob: madness
Jane: in
Alice: is
Jane: Heaven
Alice: the
Bob: yet
Alice: question
Jane: and
Bob: there
Jane: Earth
Bob: is
Jane: Horatio
Bob: method
Jane: than
Bob: in't
Jane: are
Jane: dreamt
Jane: of
Jane: in
Jane: your
Jane: philosophy

Так работает оператор `merge()`: он подписывается на слова каждого человека сразу и переправляет их дальше, не разбирая, кто говорит. Если два потока порождают события примерно в одно и то же время, то оба отправляются дальше немедленно. Внутри оператора нет ни буферизации, ни придерживания событий.

Ситуация выглядит совершенно иначе, если заменить `merge()` на `concat()`:

Alice: To
Alice: be
Alice: or
Alice: not
Alice: to
Alice: be
Alice: that
Alice: is
Alice: the
Alice: question
Bob: Though
Bob: this

```
Bob:    be
Bob:    madness
Bob:    yet
Bob:    there
Bob:    is
Bob:    method
Bob:    in't
Jane:   There
Jane:   are
Jane:   more
Jane:   things
Jane:   in
Jane:   Heaven
Jane:   and
Jane:   Earth
Jane:   Horatio
Jane:   than
Jane:   are
Jane:   dreamt
Jane:   of
Jane:   in
Jane:   your
Jane:   philosophy
```

Теперь порядок сохранен. Оператор `concat(alice, bob, jane)` сначала подписывается на поток `alice` и продолжает переправлять события из него, пока в нем еще что-то есть. Затем `concat()` переключается на поток `bob`. Вспомните о горячих и холодных `Observable`. В случае `merge()` переправляются все события из всех потоков, потому что `merge()` подписывается на них энергично и незамедлительно. Но `concat()` подписывается только на первый поток, поэтому в случае горячего `Observable` можно ожидать, что результат будет иным. К тому моменту, как первый `Observable` завершится, второй уже мог послать совершенно другую последовательность событий. Помните, что `concat()` не буферизует второй `Observable` в ожидании завершения первого, он просто подписывается на него лениво.

Оператор `switchOnNext()` дает совершенно другой способ комбинирования операторов. Пусть имеется объект типа `Observable<Observable<T>>`, т. е. поток событий, в котором каждое событие само является потоком¹¹. Так можно, например, смоделировать множество мобильных телефонов, подключающихся к сети и отключающихся от нее (внешний поток). При этом каждое подключение – это событие, но каждое такое событие само является потоком независимых контрольных сообщений (`Observable<Ping>`). В нашем же случае имеется поток типа `Observable<Observable<String>>`, где каждый внутренний поток – цитата, произносимая одним человеком: `alice`, `bob` или `jane`:

```
import java.util.Random;

Random rnd = new Random();
Observable<Observable<String>> quotes = just(
    alice.map(w -> "Alice: " + w),
```

¹¹ Нужно идти глубже. (Фраза из фильма «Начало»).

```
bob.map(w -> "Bob: " + w),
jane.map(w -> "Jane: " + w));
```

Сначала мы обертываем объекты `alice`, `bob` и `jane` объектом типа `Observable<Observable<String>>`. Повторим еще раз: объект `quotes` сразу порождает три события, каждое типа `Observable<String>`. Каждый внутренний объект типа `Observable<String>` представляет слова, произносимые одним человеком. Чтобы продемонстрировать работу `switchOnNext()`, задержим порождение внутренних `Observable` — не каждого слова внутри `Observable` (вариант *A*), а сразу всего `Observable` (вариант *B* имеет тонкое отличие):

```
//A
map(innerObs ->
    innerObs.delay(rnd.nextInt(5), SECONDS))

//B
flatMap(innerObs -> just(innerObs)
    .delay(rnd.nextInt(5), SECONDS))
```

В варианте *A* объект `Observable` сразу появляется во внешнем потоке, но начинает порождать события с некоторой задержкой, В варианте *B* весь `Observable` сдвигается вперед во времени и потому появляется во внешнем `Observable` гораздо позже. Спрашивается, зачем такие сложности? Дело в том, что оба статических оператора `concat()` и `merge()` могут работать либо с фиксированным списком `Observable`, либо с объектом `Observable`, состоящим из объектов `Observable`. В случае `switchOnNext()` имеет место лесенка.

Первым делом `switchOnNext()` подписывается на внешний `Observable<Observable<T>>`, который порождает внутренние объекты `Observable<T>`. Как только появится первый внутренний `Observable<T>`, этот оператор подписывается на него и начинает переправлять события типа `T` дальше. Но что происходит, когда появляется следующий внутренний `Observable<T>`? `switchOnNext()` отписывается от первого `Observable<T>` и переключается на следующий (отсюда и название оператора). Другими словами, если имеется поток потоков, то `switchOnNext()` всегда переправляет события из последнего внутреннего потока, даже если предыдущие продолжают порождать события.

Вот как это выглядит в нашем примере с цитированием «Гамлета»:

```
Random rnd = new Random();
Observable<Observable<String>> quotes = just(
    alice.map(w -> "Alice: " + w),
    bob.map(w -> "Bob: " + w),
    jane.map(w -> "Jane: " + w))
    .flatMap(innerObs -> just(innerObs)
        .delay(rnd.nextInt(5), SECONDS));

Observable
    .switchOnNext(quotes)
    .subscribe(System.out::println);
```

Из-за случайно природы примера результаты могут получаться разные, но один из возможных показан ниже:

```
Jane: There
Jane: are
Jane: more
Alice: To
Alice: be
Alice: or
Alice: not
Alice: to
Bob:  Though
Bob:  this
Bob:  be
Bob:  madness
Bob:  yet
Bob:  there
Bob:  is
Bob:  method
Bob:  in't
```

Все начинают говорить со случайной задержкой от 0 до 4 секунд. В данном случае первой оказалась Джейн, но не успела она произнести несколько слов, как во внешнем Observable появился Observable<String> Алисы. В этот момент `switchOnNext()` отписывается от потока jane, так что дослушать ее до конца нам не суждено. Ее Observable в дальнейшем игнорируется, а `switchOnNext()` слушает только поток alice. Но спустя некоторое время внутренний Observable снова прерывается, потому что в хор вступает Боб. Теоретически `switchOnNext()` мог бы породить все события из внутренних потоков Observable, если бы они не перекрывались, т. е. каждый поток завершался бы до появления следующего.

А что, если задерживать только события в каждом внутреннем потоке Observable (вариант А), а не сами внутренние потоки? Тогда все три внутренних потока появляться во внешнем одновременно, и `switchOnNext()` подпишется только на один из них.

Расщепление потока по условию с помощью `groupBy()`

В предметно-ориентированном проектировании (подробнее см. книгу Vaughn Vernon «Implementing Domain-Driven Design»¹², Addison-Wesley Professional) часто применяется техника «истории событий» (event sourcing). В этом случае хранится не текущее состояние, модифицируемое на месте SQL-командой UPDATE, а последовательность имевших место событий предметной области (фактов). Эти данные не подлежат изменению и добавляются в хранилище, допускающее только дписывание в конец. В результате мы никогда не перезаписываем данные и заодно бесплатно получаем контрольный журнал. А единственный способ просмотреть данные – представить всю последовательность фактов, начиная с пустого состояния.

¹² Вон Вернон «Реализация методов предметно-ориентированного проектирования», Вильямс, 2016.

Процесс применения событий к первоначально пустому состоянию называется *проекцией*. Один источник фактов может порождать несколько разных проекций. Например, может существовать поток фактов, относящихся к системе бронирования билетов: `TicketReserved` (билет забронирован), `ReservationConfirmed` (бронирование подтверждено), `TicketBought` (билет выкуплен). Прошедшее время здесь важно, потому что факты всегда отражают уже произошедшие действия и события. Из одного потока фактов (который заодно является и единственным источником истины) мы можем вывести несколько проекций, например:

- список всех подтвержденных бронирований;
- список бронирований, отмененных сегодня;
- совокупная выручка за неделю.

По мере эволюции системы мы можем отбрасывать старые проекции и строить новые, пользуясь данными, собранными в виде фактов. Предположим, что требуется построить проекцию, содержащую все бронирования вместе с состоянием. Для этого нужно отобрать все события типа `ReservationEvent` и применить их к соответствующим бронированиям. У типа `ReservationEvent` имеются подклассы, относящиеся к событиям разного типа, например: `TicketBought`. Кроме того, в каждом событии хранится `UUID` бронирования, к которому оно относится:

```
FactStore factStore = new CassandraFactStore();
Observable<ReservationEvent> facts = factStore.observe();
facts.subscribe(this::updateProjection);

//...

void updateProjection(ReservationEvent event) {
    UUID uuid = event.getReservationUuid();
    Reservation res = loadBy(uuid)
        .orElseGet(() -> new Reservation(uuid));
    res.consume(event);
    store(event.getUuid(), res);
}

private void store(UUID id, Reservation modified) {
    //...
}

Optional<Reservation> loadBy(UUID uuid) {
    //...
}

class Reservation {

    Reservation consume(ReservationEvent event) {
        // изменение данных
        return this;
    }
}
```


Понятно, что поток фактов `facts` представлен в виде `Observable`. Какая-то часть системы реагирует на вызовы API и веб-запросы (например, списывает средства с карточного счета клиента) и сохраняет факты (события предметной области), описывающие, что произошло. Другие части системы (или даже другие системы!) потребляют эти факты, подписываясь на поток, и строят снимок текущего состояния системы в любом разрезе. Наша программа очень простая: обработчик события `ReservationEvent` загружает объект `Reservation` из хранилища данных проекции. Если объект `Reservation` не найден, значит, это было первое событие с данным `uuid`, поэтому мы начинаем с пустого `Reservation`. Затем событие `ReservationEvent` передается объекту `Reservation`. Он умеет обновлять себя в соответствии с фактом любого типа. Обновленный объект `Reservation` записывается обратно в хранилище.

Напомним, что проекции не зависят от фактов, для них можно использовать любой другой механизм постоянного хранения или даже хранить состояние в памяти. Может быть даже так, что несколько проекций потребляют один и тот же поток фактов, но строят разные снимки. Например, объект `Accounting` может выделять из потока фактов только сведения о приходе и расходе денежных средств. А другая проекция может интересоваться фактами `FraudDetected` и строить сводную картину попыток мошенничества.

Это краткое введение в технику истории событий поможет нам понять, почему полезен оператор `groupBy()`. Спустя некоторое время мы обнаруживаем, что обновления проекции `Reservation` отстают, потому что мы не успеваем за темпом генерации фактов. Хранилище данных легко справляется с конкурентными операциями чтения и записи, поэтому попытаемся распараллелить обработку фактов.

```
Observable<ReservationEvent> facts = factStore.observe();

facts
    .flatMap(this::updateProjectionAsync)
    .subscribe();

//...

Observable<ReservationEvent> updateProjectionAsync(ReservationEvent event) {
    // возможно, асинхронно
}
```

В данном случае мы потребляем поток `facts` параллельно или, если быть точным, получение фактов последовательное, но обработка (в методе `updateProjectionAsync()`) может быть асинхронной. Метод `updateProjectionAsync()` изменяет состояние поданного на вход объекта `Reservation`. Но взглянув на его реализацию, мы сразу обнаруживаем потенциальное состояние гонки: два потока выполнения могут прочитать различные события, модифицировать один и тот же объект `Reservation` и попытаться сохранить его. Однако первое обновление перезаписывается и, по существу, теряется. Технически можно было бы попробовать оптимистическую блокировку, но остается другая проблема: правильный порядок применения фактов больше не гарантируется. Это не проблема, когда обновля-

ются два разных объекта `Reservation` (с разными `uuid`). Но применение фактов к одному и тому же объекту не в том порядке, в каком они имели место, может привести к катастрофе.

Тут-то и приходит на помощь оператор `groupBy()`. Он разбивает поток на несколько параллельных потоков, каждый из которых характеризуется одним и тем же значением некоторого ключа. В данном случае мы хотим разбить гигантский поток всех фактов о бронировании на много меньших потоков, каждый из которых порождает события с одним и тем же `uuid`:

```
Observable<ReservationEvent> facts = factStore.observe();

Observable<GroupedObservable<UUID, ReservationEvent>> grouped =
    facts.groupBy(ReservationEvent::getReservationUuid);

grouped.subscribe(byUuid -> {
    byUuid.subscribe(this::updateProjection);
});
```

В этом примере встречается сразу несколько новых конструкций. Сначала мы берем входной поток `Observable<ReservationEvent>` и группируем его по `uuid` (`ReservationEvent::getReservationUuid`). Вы, наверное, ожидаете, что `groupBy()` вернет `List<Observable<ReservationEvent>>` – в конце концов, мы ведь хотим преобразовать один поток в несколько. Но от этого предположения придется отказаться, как только вы осознаете, что `groupBy()` не может заранее знать, сколько различных ключей (`uuid`) есть во входном потоке. Следовательно, он должен создавать потоки на лету: как только встречается новый `uuid`, порождается новый объект `GroupedObservable<UUID, ReservationEvent>`, в который отправляются события с данным `uuid`. Таким образом, становится понятно, что внешняя структура данных должна иметь тип `Observable`.

Но что это за тип такой: `GroupedObservable<UUID, ReservationEvent>`? `GroupedObservable` – это подкласс `Observable`, который, помимо стандартного контракта `Observable`, возвращает ключ, общий для всех событий в данном потоке (в нашем случае `uuid`). Количество порожденных объектов `GroupedObservable` может варьироваться от 1 (т. е. у всех событий одинаковые ключи), до общего числа событий (если у каждого события уникальный ключ). Это один из тех случаев, когда к вложенности объектов `Observable` нет никаких претензий. Подписавшись на внешний `Observable`, мы получаем в качестве событий другие объекты `Observable` (типа `GroupedObservable`), на которые тоже можем подписаться. Например, в одном внутреннем потоке могут находиться взаимосвязанные события (скажем, с одинаковым корреляционным идентификатором), а разные внутренние потоки никак не связаны между собой и могут обрабатываться по отдельности.

Что дальше?

В `RxJava` есть еще десятки встроенных операторов. Многие из них будут рассмотрены в главе 6, но описывать весь API не слишком разумно и заняло бы много места. К тому же, исчерпывающее описание неизбежно устарело бы уже с выходом

следующей версии. Однако необходимо иметь общее представление о том, что могут операторы и как они работают. Логично, что следующим шагом будет написание собственных операторов.

Написание пользовательских операторов

Пока что мы видели лишь верхушку айсберга операторов в RxJava, но дальше встретим еще немало. Истинная мощь операторов – в композиции. Следуя философии UNIX – «*небольшие специализированные инструменты*»¹³ – каждый оператор выполняет одно небольшое преобразование. В этом разделе мы сначала рассмотрим оператор `compose()`, который обеспечивает возможность текучей композиции меньших операторов, а затем познакомимся с оператором `lift()`, помогающим писать совершенно новые операторы.

Повторное использование операторов с помощью `compose()`

Начнем с примера. По какой-то причине мы хотим преобразовать входной `Observable`, так чтобы каждый второй элемент отбрасывался и оставались только четные элементы. В разделе «Управление потоком» главы 6 мы узнаем об операторе `buffer()`, благодаря которому эта задача решается тривиально (`buffer(1, 2)` делает почти то, что нам нужно). Но пока сделаем вид, что ничего о нем не знаем. И тем не менее, мы легко сможем получить желаемое поведение с помощью композиции нескольких операторов:

```
import org.apache.commons.lang3.tuple.Pair;

//...

Observable<Boolean> trueFalse = Observable.just(true, false).repeat();
Observable<T>.upstream = //...
Observable<T> downstream = upstream
    .zipWith(trueFalse, Pair::of)
    .filter(Pair::getRight)
    .map(Pair::getLeft);
```

Сначала мы генерируем бесконечный поток `Observable<Boolean>`, который периодически порождает значения `true` и `false`. Это легко сделать, создав фиксированный поток `[true, false]` из двух элементов и повторив его бесконечно с помощью оператора `repeat()`. Оператор `repeat()` просто перехватывает уведомление о завершении входного потока и вместо того, чтобы передать его дальше, подписывается заново. Вообще говоря, не гарантируется, что будет повторяться одна и та

¹³ Hunt A., Thomas, D. «The Pragmatic Programmer: From Journeyman to Master» (Addison-Wesley Professional).

же последовательность событий, но это так, если входной поток фиксированный. В разделе «Повторение после сбоя» главы 7 описан похожий оператор `retry()`.

Затем мы применяем оператор `zipWith()` к нашему входному потоку `Observable` и этому бесконечному потоку `true` и `false`. Однако для этого нужна функция, объединяющая два элемента. В других языках это делается проще, ну а в Java нас выручит библиотека Apache Commons Lang (<https://commons.apache.org/proper/commons-lang/>), в которой имеется простой класс `Pair`. Итак, мы имеем поток значений типа `Pair<T, Boolean>`, в каждом из которых справа находится `true` или `false` (пара состоит из двух компонент: *left* и *right*). На следующем шаге мы производим фильтрацию с помощью оператора `filter()`, оставляя только пары, в которых справа находится `true`. То есть каждую вторую пару мы попросту отбрасываем. Последний шаг – убрать из пары вторую компоненту типа `Boolean` и оставить только компоненту типа `T` (`getLeft()`). Если не хотите пользоваться сторонней библиотекой, то вот вам альтернативная реализация:

```
import static rx.Observable.empty;
import static rx.Observable.just;

//...

upstream.zipWith(trueFalse, (t, bool) ->
    bool ? just(t) : empty())
    .flatMap(obs -> obs)
```

На первый взгляд, оператор `flatMap()` выглядит странно, не похоже, чтобы он делал что-то полезное. Но дело в том, что преобразование в `zipWith()` возвращает объект `Observable` (с одним элементом или пустой), т. е. весь оператор порождает `Observable<Observable<T>>`. А `flatMap()` позволяет избавиться от лишнего уровня вложенности – ведь лямбда-выражение в нем возвращает `Observable` для каждого входного элемента, который, по стечению обстоятельств, тоже является объектом типа `Observable`.

Но и та, и другая реализация плохо приспособлена для повторного использования. Если бы понадобилась последовательность операторов, оставляющая каждый нечетный элемент, то пришлось бы либо заняться копированием и вставкой, либо создать примерно такой вспомогательный метод:

```
static <T> Observable<T> odd(Observable<T> upstream) {
    Observable<Boolean> trueFalse = just(true, false).repeat();
    return upstream
        .zipWith(trueFalse, Pair::of)
        .filter(Pair::getRight)
        .map(Pair::getLeft)
}
```

Но при этом теряется возможность сцеплять операторы, т. е. мы больше не можем написать: `obs.op1().odd().op2()`. В отличие от языков C# (в котором реактивные расширения первоначально и были реализованы) и Scala¹⁴ (благодаря

¹⁴ Обертка RxJava для Scala находится по адресу <http://reactivex.io/rxscala>.

ря неявным преобразованиям), в Java нет методов расширения. Но встроенный оператор `compose()` дает максимально близкое приближение к ним. Он принимает в качестве аргумента функцию, назначение которой – преобразовать входной `Observable` путем применения последовательности операторов. Вот как это выглядит на практике:

```
private <T> Observable.Transformer<T, T> odd() {
    Observable<Boolean> trueFalse = just(true, false).repeat();
    return upstream -> upstream
        .zipWith(trueFalse, Pair::of)
        .filter(Pair::getRight)
        .map(Pair::getLeft);
}

//...

//[A, B, C, D, E...]
Observable<Character> alphabet =
    Observable
        .range(0, 'Z' - 'A' + 1)
        .map(c -> (char) ('A' + c));

//[A, C, E, G, I...]
alphabet
    .compose(odd())
    .forEach(System.out::println);
```

Функция `odd()` возвращает объект типа `Transformer<T, T>`, описывающий преобразование `Observable<T>` в `Observable<T>` (конечно, типы могут и различаться). Но раз `Transformer` – просто функция, то мы можем заменить ее лямбда-выражением (`upstream -> upstream...`). Отметим, что функция `odd()` выполняется энергично в момент сборки `Observable`, а не лениво в момент подписки. Любопытно, что если мы захотим породить четные значения (второе, четвертое и т. д.) вместо нечетных (первое, третье и т. д.), то нужно будет просто заменить `trueFalse` на `trueFalse.skip(1)`.

Реализация более сложных операторов с помощью `lift()`

Реализация пользовательских операторов осложняется необходимостью учитывать противодействие (см. раздел «Противодействие» главы 6) и механизм подписки. Поэтому прежде чем изобретать собственный оператор, постарайтесь обойтись существующими. Встроенные операторы гораздо лучше протестированы и проверены на практике. Но если ни один из имеющихся операторов все же не подходит, то на помощь придет метаоператор `lift()`. Оператор `compose()` полезен только для группировки существующих операторов. А `lift()` позволяет реализовать почти любой оператор путем изменения потока входящих событий.

Если `compose()` преобразовывает объекты `Observable`, то `lift()` позволяет преобразовать объекты `Subscriber`. Вспомним, что мы узнали в разделе «Подробнее

о методе `Observable.create()`» главы 2. Когда вызывается метод `subscribe()` для подписки на объект `Observable`, экземпляр `Subscriber`, обертывающий обратный вызов, добирается до этого объекта и приводит к вызову его метода `create()`, которому подписчик передается в качестве аргумента (это сильно упрощенное описание). Таким образом, при любой подписке объект `Subscriber` поднимается по цепочке операторов, пока не дойдет до того `Observable`, на который мы подписались. Очевидно, что между `Observable` и `subscribe()` может быть сколько угодно операторов, изменяющих события, движущиеся вниз по цепочке, например:

```
Observable
    .range(1, 1000)
    .filter(x -> x % 3 == 0)
    .distinct()
    .reduce((a, x) -> a + x)
    .map(Integer::toHexString)
    .subscribe(System.out::println);
```

Но вот что интересно: если заглянуть в исходный код RxJava и заменить вызов каждого оператора его телом, то эта кажущаяся весьма сложной последовательность становится вполне регулярной (обратите внимание, что `reduce()` реализован как `scan().takeLast(1).single()`):

```
Observable
    .range(1, 1000)
    .lift(new OperatorFilter<>(x -> x % 3 == 0))
    .lift(OperatorDistinct.<Integer>instance())
    .lift(new OperatorScan<>((Integer a, Integer x) -> a + x))
    .lift(OperatorTakeLastOne.<Integer>instance())
    .lift(OperatorSingle.<Integer>instance())
    .lift(new OperatorMap<>(Integer::toHexString))
    .subscribe(System.out::println);
```

Почти все операторы, кроме тех, что работают сразу с несколькими потоками (как `flatMap()`), реализованы с помощью `lift()`. Когда мы вызываем `subscribe()` в самом конце цепочки, создается экземпляр `Subscriber<String>` и передается непосредственному предшественнику. Это может быть «настоящий» `Observable<String>`, который порождает события, или просто результат какого-то оператора, в нашем случае `map(Integer::toHexString)`. Оператор `map()` сам не порождает событий, но все же получает объект `Subscriber`, желающий получать события. `map()` просто прозрачно для нас подписывается (с помощью вспомогательного оператора `lift()`) на своего родителя (в данном случае оператор `reduce()`). Однако он не может передать тот же экземпляр `Subscriber`, который получил, поскольку метод `subscribe()` ожидает аргумент типа `Subscriber<String>`, а метод `reduce()` — типа `Subscriber<Integer>`. Собственно, именно в этом — преобразовании `Integer` в `String` — и состоит назначение этого оператора `map()`. Поэтому `map()` создает искусственный объект типа `Subscriber<Integer>` и всякий раз как этот специальный `Subscriber` получает какое-то событие, `map()` применяет к этому событию функцию `Integer::toHexString` и уведомляет следующий далее объект `Subscriber<String>`.

Под капотом оператора map()

Именно это и делает класс `OperatorMap`: обеспечивает преобразование из выходного типа `Subscriber<R>` (`child`) во входной тип `Subscriber<T>`. Ниже приведена его реализация в исходном коде RxJava с несущественными упрощениями для большей понятности:

```
public final class OperatorMap<T, R> implements Operator<R, T> {

    private final Func1<T, R> transformer;

    public OperatorMap(Func1<T, R> transformer) {
        this.transformer = transformer;
    }

    @Override
    public Subscriber<T> call(final Subscriber<R> child) {
        return new Subscriber<T>(child) {

            @Override
            public void onCompleted() {
                child.onCompleted();
            }

            @Override
            public void onError(Throwable e) {
                child.onError(e);
            }

            @Override
            public void onNext(T t) {
                try {
                    child.onNext(transformer.call(t));
                } catch (Exception e) {
                    onError(e);
                }
            }
        };
    }
}
```

Одна необычная деталь — измененный порядок параметрических типов `T` и `R`. Оператор `map()` преобразует входящие значения типа `T` в исходящие типа `R`. Однако в обязанности оператора входит преобразование `Subscriber<R>` (поступающего от следующей далее подписки) в `Subscriber<T>` (передаваемого находящемуся выше `Observable`). Мы ожидаем подписки от `Subscriber<R>`, тогда как `map()` применяется к `Observable<T>`, требующему `Subscriber<T>`.

Постарайтесь хотя бы в общих чертах понять этот фрагмент исходного кода RxJava. Поняв, как реализован оператор `map()` (вообще-то, один из самых простых), вы сможете написать собственный оператор. Применяя `map()` к потоку, мы фактически вызываем `lift()`, передавая ему новый экземпляр класса `OperatorMap`,

который предоставляет функцию `transformer`. Эта функция получает входящие события типа `T` и возвращает исходящие события типа `R`. Всякий раз как пользователь передает вашему оператору какую-то функцию или преобразование, обязательно перехватывайте все возникающие в ней исключения и передавайте их дальше методом `onError()`. Это заодно гарантирует отписку от потока, так что он больше не будет порождать события.

Помните, что пока кто-то не подписался, мы всего лишь создали новый объект `Observable` (`lift()`, как и любой другой оператор, создает новый `Observable`), хранящий ссылку на экземпляр `OperatorMap`, который, в свою очередь, хранит ссылку на нашу функцию. Но стоит клиенту подписаться, как вызывается метод `call()` объекта `OperatorMap`. Эта функция получает наш объект `Subscriber<String>` (например, обортывает его вызовом метода `System.out::println`) и возвращает объект `Subscriber<Integer>`. Именно этот объект передается вверх по цепочке предшествующим операторам.

Примерно так работают все операторы – встроенные и пользовательские. Оператор получает один объект `Subscriber` и возвращает другой, который как-то преобразует и передает то, что считает нужным, объекту `Subscriber`, следующему за ним.

Наш первый оператор

Теперь мы хотели бы реализовать оператор, который будет порождать результат применения `toString()` к каждому нечетному элементу. Объяснить это проще всего на примере кода:

```
Observable<String> odd = Observable
    .range(1, 9)
    .lift(toStringOfOdd())
// Порождает строки: "1", "3", "5", "7" и "9"
```

Того же результата можно следующим образом добиться с помощью встроенных операторов, так что мы приводим пользовательский оператор только в педагогических целях:

```
Observable
    .range(1, 9)
    .buffer(1, 2)
    .concatMapIterable(x -> x)
    .map(Object::toString);
```

Оператор `buffer()` будет описан в разделе «Буферизация событий в списке» главы 6, а пока достаточно знать, что `buffer(1, 2)` преобразует `Observable<T>` в `Observable<List<T>>`, так что каждый внутренний список `List` содержит ровно один нечетный элемент, а четные пропускаются. Имея поток списков `List(1)`, `List(3)` и т. д., мы реконструируем плоский поток оператором `concatMapIterable()`. Но обучения ради реализуем пользовательский оператор, который делает все это за один шаг. Оператор может находиться в одном из двух состояний:

- из входного потока получено нечетное событие, которое передается в выходной поток после применения метода `toString()`;
- получено четное событие, оно просто отбрасывается.

Затем цикл повторяется. Код оператора мог бы выглядеть так:

```
<T> Observable.Operator<String, T> toStringOfOdd() {
    return new Observable.Operator<String, T>() {

        private boolean odd = true;

        @Override
        public Subscriber<? super T> call(Subscriber<? super String> child) {
            return new Subscriber<T>(child) {
                @Override
                public void onCompleted() {
                    child.onCompleted();
                }

                @Override
                public void onError(Throwable e) {
                    child.onError(e);
                }

                @Override
                public void onNext(T t) {
                    if(odd) {
                        child.onNext(t.toString());
                    } else {
                        request(1);
                    }
                    odd = !odd;
                }
            };
        }
    };
}
```

Подробнее о вызове `request(1)` речь пойдет в разделе «Учет запрошенного объема данных» главы 6. А пока можно представлять его следующим образом: если подписчик запрашивает лишь подмножество событий, например только первые два (`take(2)`), то RxJava понимает это и запрашивает именно столько данных, вызывая метод `request(2)`. Этот запрос передается вверх, и мы получаем только значения 1 и 2. Но четное значение 2 отбрасывается, а дальше мы, тем не менее, обязаны передать два события. Поэтому следует запросить одно дополнительное событие (`request(1)`), т. е. мы получаем еще и 3. В RxJava реализован довольно сложный механизм *противодавления*, который позволяет подписчикам запрашивать ровно столько событий, сколько они могут обработать; это защищает потребителей от чрезмерно «прытких» производителей. Этой теме посвящен раздел «Противодавление» главы 6.



К сожалению, хорошо это или плохо, `null` считается в RxJava допустимым событием, т. е. поток `Observable.just("A", null, "B")` ничем не хуже любого другого. Об этом следует помнить как при проектировании пользовательских операторов, так и при применении операторов к потоку. Но, вообще говоря, передача `null` считается неидиоматической практикой, рекомендуется использовать обертывающие значащие типы.

Еще один любопытный подвох поджидает нас в случае, если новому `Subscriber` не передан дочерний в качестве аргумента:

```
<T> Observable.Operator<String, T> toStringOfOdd() {
    // НЕПРАВИЛЬНО
    return child -> new Subscriber<T>() {
        //...
    }
}
```

Конструктор `subscriber` без параметров вполне допустим, и, на первый взгляд, наш оператор работает. Но посмотрим, что происходит, если поток бесконечен:

```
Observable
    .range(1, 4)
    .repeat()
    .lift(toStringOfOdd())
    .take(3)
    .subscribe(
        System.out::println,
        Throwable::printStackTrace,
        () -> System.out.println("Завершен")
    );
```

Мы строим бесконечный поток чисел (1, 2, 3, 4, 1, 2, 3...), применяем наш оператор («1», «3», «1», «3»...) и отбираем только первые три значения. Все вполне допустимо и никаких ошибок не предвидится – ведь потоки ленивые. Но стоит убрать `child` из конструктора `new Subscriber(child)`, как наш `Observable` перестает уведомлять о завершении после получения чисел 1, 3, 1. Что случилось?

Оператор `take(3)` запросил только первые три значения и захотел отписаться, вызвав метод `unsubscribe()`. К несчастью, запрошенная отписка так и не добралась до исходного потока, который продолжает порождать значения. Хуже того, эти значения обрабатываются нашим оператором и передаются следующему за ним подписчику (`take(3)`), который уже не слушает. Не будем вдаваться в детали реализации, а просто постулируем: в собственных операторах передавайте дочерний объект `Subscriber` конструктору нового `Subscriber`. Конструктор без параметров используется редко и крайне маловероятно, что он потребуется вам при написании простых операторов.

Это лишь малая толика тех проблем, с которыми можно столкнуться при создании пользовательских операторов. По счастью, очень редко бывает так, что желаемого нельзя достичь с помощью встроенных механизмов.

Резюме

Истинная мощь RxJava – в операторах. Декларативные преобразования потоков данных – безопасный и вместе с тем выразительный и гибкий механизм. Имея основательный фундамент в функциональном программировании, операторы играют решающую роль во внедрении RxJava. Свободное владение встроенными операторами – ключ к успешной работе с этой библиотекой. Помните, что мы видели еще не все операторы; дополнительные описаны, например, в разделе «Управление потоком» главы 6. Но уже сейчас вы имеете общее представление о том, что позволяет делать RxJava и как расширить ее возможности, если их не хватает.



Глава 4.

Применение реактивного программирования в существующих приложениях

Решение о включении новой библиотеки, технологии или парадигмы в приложение – унаследованное или разрабатываемое с нуля – требует тщательного обдумывания. RxJava – не исключение. В этой главе мы обсудим некоторые паттерны и архитектуры, встречающиеся в типичных приложениях Java, и посмотрим, чем может помочь Rx. Процесс это не простой, требующий значительных умственных усилий и смены угла зрения, поэтому мы будем переходить от императивного стиля к функциональному и реактивному постепенно. Многие современные библиотеки в проектах на Java лишь раздувают код, не давая ничего взамен. Однако, как мы увидим, RxJava не только позволяет упростить традиционные проекты, но и несет определенные выгоды унаследованным.

Я почти уверен, что вы уже находитесь под впечатлением RxJava. Встроенные операторы и простота их применения делают Rx на удивление мощным инструментом для преобразования потоков событий. Но завтра, придя на работу, вы осознаете, что в ваших-то приложениях потоков нет, как нет и событий реального времени от фондовой биржи. Там вообще-то событий днем с огнем не сыщешь – сплошные веб-запросы, базы данных да внешние API. Вам бы так хотелось опробовать эту новую штуку на чем-то, кроме «Hello world». Но похоже, в реальной жизни просто нет ситуаций, оправдывающих применение Rx. И тем не менее, RxJava может стать значительным шагом вперед с точки зрения архитектурного единообразия и эксплуатационной надежности. Вовсе не обязательно переходить на реактивный стиль сверху донизу – это рискованно и в начале потребует слишком много работы. Но Rx можно внедрить на любом уровне, не переделывая приложение целиком.

Мы рассмотрим некоторые типичные примеры приложений и покажем, как их можно улучшить путем ненавязчивого внедрения RxJava. Особое внимание будет уделено запросам к базе данных, кэшированию, обработке ошибок и периодическим задачам. Чем больше будет мест, куда вы внедрите RxJava, тем более единообразной станет архитектура.

От коллекций к Observable

Если ваше приложение не было разработано относительно недавно на базе таких каркасов, как Play (<https://www.playframework.com/>), акторы Akka (<http://akka.io/>) или, быть может, Vert.x (<http://vertx.io/>), то, наверное, вы имеете дело с контейнером сервлетов, с одной стороны, и JDBC или веб-сервисами – с другой. Между ними находится несколько уровней, на которых реализована бизнес-логика. Мы не будем перерабатывать их одновременно, а начнем с простого примера. Следующий класс представляет тривиальный репозиторий, абстрагирующий базу данных:

```
class PersonDao {  
  
    List<Person> listPeople() {  
        return query("SELECT * FROM PEOPLE");  
    }  
  
    private List<Person> query(String sql) {  
        //...  
    }  
  
}
```

Оставляя в стороне детали реализации, зададимся вопросом, какое отношение это имеет к Rx? До сих пор мы говорили об асинхронных событиях, поступающих от внешних систем, когда мы на них подписываемся. И при чем здесь этот прозаический Dao-объект? Дело в том, что Observable – не только канал для проталкивания событий. Объект Observable<T> можно рассматривать как структуру данных, двойственную к Iterable<T>. То и другое – контейнер элементов типа T, хотя их интерфейсы принципиально различны. Поэтому не должно вызывать удивления, что можно просто подменить один объект другим:

```
Observable<Person> listPeople() {  
    final List<Person> people = query("SELECT * FROM PEOPLE");  
    return Observable.from(people);  
}
```

Но это изменение несовместимо с существующим API. Если ваша система велика, то такая несовместимость может оказаться серьезной проблемой. Поэтому важно включать RxJava в API на как можно более ранних стадиях. Но мы-то имеем дело с существующим приложением, поэтому так вопрос не стоит.

BlockingObservable: выход из реактивного мира

Если мы комбинируем RxJava с существующим кодом – блокирующим и императивным, то, возможно, придется перейти от Observable к обычной коллекции.

Это довольно неприятное преобразование, для него необходимо блокировать программу в ожидании завершения Observable. Пока Observable не завершится, мы не можем создать коллекцию. Тип BlockingObservable специально создан для работы с Observable в нереактивном окружении. Вообще, это последнее, к чему следует прибегать при работе с RxJava, но если требуется объединить блокирующий и неблокирующий код, то никуда не денешься.

В главе 3 мы переработали метод listPeople(), так что он стал возвращать Observable<Person> вместо List. Observable ни в каком смысле не является частным случаем Iterable, поэтому программа перестала компилироваться. Мы хотим продвигаться вперед мелкими шажками, а не переделывать сразу все, поэтому постараемся ограничить область видимости изменений. Клиентский код мог бы выглядеть так:

```
List<Person> people = pesonDao.listPeople();  
String json = marshal(people);
```

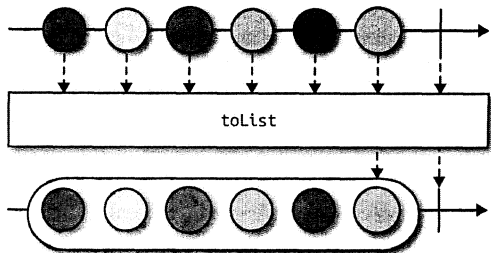
Можно представить себе метод marshal(), который *вытягивает* данные из коллекции people и сериализует их в формате JSON. Но так больше не получится, потому что мы не можем вытягивать элементы из Observable, когда нам заблагорассудится. Смысл Observable в том, что он порождает (*проталкивает*) элементы, уведомляя подписчиков, если таковые имеются. Но это радикальное изменение легко обойти с помощью BlockingObservable. Этот полезный класс совершенно не зависит от Observable, его экземпляр можно получить методом Observable.toBlocking(). У блокирующего варианта Observable имеются одноименные методы single() и subscribe(), но сходство чисто поверхностное. Однако BlockingObservable гораздо удобнее в блокирующем окружении, которое абсолютно не готово к работе с асинхронным по своей природе Observable. Операторы BlockingObservable, как правило, блокируют выполнение (ждут), пока ассоциированный с ним Observable не завершится. Это вступает в разительное противоречие с основной идеей Observable: все на свете асинхронное, ленивое и обрабатывается на лету. Например, метод Observable.forEach() асинхронно принимает события от Observable по мере поступления, тогда как метод BlockingObservable.forEach() блокирует выполнение до тех пор, пока все события не будут обработаны и поток не завершится. Исключения также больше не передаются в виде значений (событий), а повторно возбуждаются в вызывающем потоке.

В нашем случае мы хотим преобразовать Observable<Person> обратно в List<Person>, чтобы уменьшить область видимости рефакторинга:

```
Observable<Person> peopleStream = personDao.listPeople();  
Observable<List<Person>> peopleList = peopleStream.toList();  
BlockingObservable<List<Person>> peopleBlocking = peopleList.toBlocking();  
List<Person> people = peopleBlocking.single();
```

Я сознательно оставил все промежуточные типы явными, чтобы объяснить, что здесь происходит. После перехода на Rx наш API возвращает поток Observable<Person> peopleStream. Этот поток потенциально может оказаться реак-

тивным, асинхронным и событийно-ориентированным, что никак не согласуется с нашими потребностями: статический список `List`. На первом шаге мы преобразуем `Observable<Person>` в `Observable<List<Person>>`. Этот ленивый оператор буферизует все события `Person`, сохраняя их в памяти до получения события `onCompleted()`. В этот момент порождается единственное событие типа `List<Person>`, которое содержит сразу все сохраненные события (см. диаграмму ниже):



Результирующий поток завершается сразу после порождения единственного элемента типа `List`. Этот оператор асинхронный, он не ждет поступления всех событий, а лениво буферизует значения. Такой неприглядный объект `Observable<List<Person>>` `peopleList` затем преобразуется в `BlockingObservable<List<Person>>` `peopleBlocking`. Использование `BlockingObservable` можно считать хорошей идеей, только если необходимо блокирующее статическое представление в остальном асинхронного `Observable`. Если `Observable.from(List<T>)` преобразует обычную коллекцию на основе вытягивания в `Observable`, то метод `toBlocking()` делает прямо противоположное. Может возникнуть вопрос: а зачем нам две абстракции для блокирующих и неблокирующих операторов? Авторы RxJava решили, что различие между синхронной и асинхронной природой оператора настолько важно, что отмечать его лишь в документации недостаточно – оно должно быть явным. Наличие двух не связанных между собой типов гарантирует, что мы всегда будем работать с правильной структурой данных. Кроме того, `BlockingObservable` – это последнее прибежище; в обычной ситуации следует пользоваться композицией и сцеплением `Observable`. Но для целей этого упражнения нам нужно сразу отойти от `Observable`. Последний оператор – `single()` – расстается с наблюдаемыми объектами навсегда и извлекает из `BlockingObservable<T>` тот единственный элемент, который мы ожидаем получить. Похожий оператор `first()` вернул бы первое значение и отбросил все остальное. Но `single()` дополнительно проверяет, что в объекте `Observable` не осталось никаких событий. Это означает, что `single()` блокирует выполнение в ожидании вызова `onCompleted()`. А вот тот же самый код, только все операторы сцеплены:

```
List<Person> people = personDao
    .listPeople()
    .toList()
```

```
.toBlocking()  
.single();
```

Возможно, вам показалось, что вся эта суеда с обертыванием и развертыванием `Observable` – пустая трата времени. Но ведь это был только первый шаг. Следующее преобразование добавит ленивость. Пока что наш код всегда выполняет `query("...")` и обертывает результат объектом `Observable`. Как мы уже знаем, объекты `Observable` (особенно холодные) ленивые по определению. Пока на такой объект никто не подписан, он лишь представляет поток, которому не было предоставлено шанса начать порождение значений. Вы можете сколько угодно вызывать методы, возвращающие `Observable`, но пока никто не подписал, никакой полезной работы не производится. `Observable` похож на `Future` в том смысле, что лишь обещает значение, которое появится в будущем. Но если вы это значение не попросите, то холодный `Observable` и не начнет ничего порождать. С этой точки зрения, `Observable` больше напоминает тип `java.util.function.Supplier<T>`, который порождает значения типа `T` по требованию. Горячие объекты `Observable` отличаются тем, что порождают значения независимо от того, слушает кто-то или нет, но не о них сейчас речь. Само существование `Observable` еще не означает наличия какой-то фоновой задачи или побочного эффекта, в отличие от `Future`, который почти всегда подразумевает выполнение некоторой конкурентной операции.

О пользе лени

Так как же сделать наш `Observable` ленивым? Простейший способ – применить к энергичному `Observable` метод `defer()`:

```
public Observable<Person> listPeople() {  
    return Observable.defer(() ->  
        Observable.from(query("SELECT * FROM PEOPLE")));  
}
```

Метод `Observable.defer()` принимает лямбда-выражение (фабрику), умеющее порождать объект `Observable`. Этот `Observable` энергичный, поэтому мы хотим отложить его создание, и `defer()` будет ждать до самого последнего момента, т. е. до тех пор, пока кто-то не подпишется. У этого факта есть интересные следствия. Поскольку `Observable` ленивый, вызов `listPeople()` не имеет побочных эффектов и почти не сказывается на производительности. Обращения к базе данных еще не было. Мы можем расценивать `observable<Person>` как обещание, но без какой-то ни было фоновой работы. Отметим, что пока отсутствует асинхронное поведение, имеет место только ленивость. Это можно сравнить с ленивым вычислением значений в языке `Haskell` (<https://www.haskell.org/>), которое производится не раньше, чем абсолютно необходимо.

Тем, кто никогда не программировал на функциональных языках, может быть непонятно, почему ленивость – столь важная и революционная концепция. Как выясняется, это весьма полезное поведение, которое может существенно повысить качество и свободу реализации. Например, больше не нужно обращать внимания

на то, какие ресурсы выбираются, когда и в каком порядке. RxJava загрузит их только тогда, когда абсолютно необходимо.

В качестве примера рассмотрим тривиальный механизм выбора значения по умолчанию, с которым все мы неоднократно сталкивались:

```
void bestBookFor(Person person) {
    Book book;
    try {
        book = recommend(person);
    } catch (Exception e) {
        book = bestSeller();
    }
    display(book.getTitle());
}

void display(String title) {
    //...
}
```

Возможно, вы не видите в такой конструкции ничего плохого. Здесь мы пытаемся рекомендовать пользователю книгу, которая была бы ему наиболее интересна, а если не получается, то показываем просто бестселлер. Идея в том, что бестселлер найти проще, и его можно кэшировать. А если бы удалось обработать ошибки декларативно, без этих блоков try-catch, которые только затемняют логику?

```
void bestBookFor(Person person) {
    Observable<Book> recommended = recommend(person);
    Observable<Book> bestSeller = bestSeller();
    Observable<Book> book = recommended.onErrorResumeNext(bestSeller);
    Observable<String> title = book.map(Book::getTitle);
    title.subscribe(this::display);
}
```

Мы пока только исследуем возможности RxJava, поэтому я оставил промежуточные значения и типы. В реальной программе метод `bestBookFor()` выглядел бы как-то так:

```
void bestBookFor(Person person) {
    recommend(person)
        .onErrorResumeNext(bestSeller())
        .map(Book::getTitle)
        .subscribe(this::display);
}
```

Код замечательно краткий и абсолютно понятный. Сначала искать рекомендацию для `person`. В случае ошибки (`onErrorResumeNext`) предложить бестселлер. В любом случае `map` извлекает название найденной книги и возвращает его, после чего оно отображается. Оператор `onErrorResumeNext()` перехватывает исключения, возникшие во входном потоке, проглатывает их и подписывается на предоставленный *резервный* `Observable`. Именно так в Rx реализована семантика блока try-catch. Мы еще будем гораздо подробнее говорить об обработке ошибок (см. раздел

«Декларативная замена try-catch» главы 7). А пока отметим, как можно лениво вызвать `bestSeller()`, не беспокоясь о том, как бы не предложить бестселлер в случае, если удалось найти рекомендуемую книгу.

Композиция объектов Observable

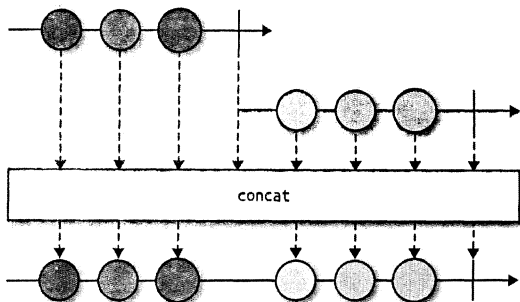
Команду `SELECT * FROM PEOPLE` не назовешь образцом современного SQL-запроса. Прежде всего, не следует выбирать все столбцы без разбору, но выборка всех строк – это еще хуже. Наш старый API не умеет разбивать результат на страницы и показывать только часть таблицы. В традиционном корпоративном приложении код мог бы выглядеть так:

```
List<Person> listPeople(int page) {  
    return query(  
        "SELECT * FROM PEOPLE ORDER BY id LIMIT ? OFFSET ?",  
        PAGE_SIZE,  
        page * PAGE_SIZE  
    );  
}
```

Это не книга по SQL, поэтому детали реализации оставим в стороне. Автор этого API не знал жалости: нам не дано выбрать диапазон записей по собственному усмотрению, можно только задавать номер страницы, начиная с 0. А вот в RxJava благодаря ленивости мы можем смоделировать чтение всей таблицы, начиная с заданной страницы:

```
import static rx.Observable.defer;  
import static rx.Observable.from;  
  
Observable<Person> allPeople(int initialPage) {  
    return defer(() -> from(listPeople(initialPage)))  
        .concatWith(defer(() ->  
            allPeople(initialPage + 1)));  
}
```

Здесь мы лениво загружаем первую страницу, например 10 элементов. Если подписчиков нет, то даже этот первый запрос не вызывается. Если имеется подписчик, потребляющий лишь несколько первых элементов (например, `allPeople(0).take(3)`), то RxJava автоматически отпишется от нашего потока, и больше никаких запросов не будет. А что случится, если мы запросили, скажем, 11 элементов, а первое обращение к `listPeople()` вернуло только 10? Тогда RxJava поймет, что исходный Observable исчерпан, а потребитель еще не насытился. Но тут она видит оператор `concatWith()`, который говорит: если Observable слева завершился, то не нужно отправлять подписчикам уведомление о завершении, а следует подписаться на Observable справа и продолжать как ни в чем не бывало. Это показано на следующей камешковой диаграмме:



Иными словами, оператор `concatWith()` может соединить два объекта `Observable` так, что когда один завершается, начинает работу другой. Конструкция `a.concatWith(b).subscribe(...)` означает, что подписчик сначала получает все события из `a`, а затем все события из `b`. В данном случае подписчик сначала получает первые 10 элементов, а потом следующие 10. Но приглядевшись внимательно, мы заподозрим в этом коде бесконечную рекурсию! `allPeople(initialPage)` вызывает `allPeople(initialPage + 1)`, а никакого условия остановки нет. В большинстве языков это прямая дорога к ошибке переполнения стека `StackOverflowError`, но только не здесь. Напомним, что вызов `allPeople()` всегда ленивый, поэтому в тот момент, как мы перестаем прослушивать события (отписываемся), рекурсия прекращается. Технически `concatWith()` все-таки может привести к ошибке `StackOverflowError`. Но потерпите до раздела «Учет запрошенного объема данных» главы 6, в котором вы узнаете, как обрабатывать переменные требования к объему входных данных.

Техника ленивой загрузки данных порциями очень полезна, потому что позволяет сконцентрироваться на бизнес-логике, а не на низкоуровневой инфраструктуре. Мы уже видели некоторые преимущества, которые `RxJava` дает даже при небольшом масштабе. Проектирование API с учетом `Rx` не влияет на архитектуру в целом, поскольку мы всегда можем вернуться к `BlockingObservable` и коллекциям `Java`. Но лучше все же иметь широкий спектр возможностей, который при необходимости можно и сузить.

Ленивое разбиение на страницы и конкатенация

Есть и другие способы реализовать ленивое разбиение на страницы средствами `RxJava`. Вообще-то, самый простой способ получить страницу – загрузить все данные и отобразить то, что нужно. Звучит глупо, но благодаря ленивости это возможно. Сначала сгенерируем все возможные номера страниц, а затем запросим загрузку каждой страницы по отдельности:

```
Observable<List<Person>> allPages = Observable
    .range(0, Integer.MAX_VALUE)
```

```
.map(this::listPeople)  
.takeWhile(list -> !list.isEmpty());
```

Если бы не RxJava, то этот код потребовал бы немеряно времени и памяти, ведь в память пришлось бы загрузить всю таблицу целиком. Но поскольку Observable ленивый, в этот момент никакого запроса к базе еще не было. Кроме того, если мы наткнемся на пустую страницу, значит, все последующие страницы также пустые (мы дошли до конца таблицы). Поэтому мы воспользовались оператором `takeWhile()`, а не `filter()`. Чтобы сериализовать `allPages` и получить `Observable<Person>`, мы можем воспользоваться оператором `concatMap()` (см. раздел «Сохранение порядка с помощью `concatMap()`» главы 3):

```
Observable<Person> people = allPages.concatMap(Observable::from);
```

`concatMap()` принимает преобразование из `List<Person>` в `Observable<Person>`, выполняемое для каждой страницы. Можно было бы вместо этого применить оператор `concatMapIterable()`, который делает то же самое, но ожидает, что преобразование вернет объект `Iterable<Person>` для каждого входящего значения (которое и так уже имеет тип `Iterable<Person>`):

```
Observable<Person> people = allPages.concatMapIterable(page -> page);
```

При любом подходе все преобразования объекта `Person` ленивые. При условии, что количество подлежащих обработке записей ограничено (например, `people.take(15)`), `Observable<Person>` вызовет метод `listPeople()` так поздно, как только возможно.

Императивная конкурентность

Мне не часто доводилось встречать явную конкурентность в корпоративных приложениях. Как правило, один запрос обрабатывается в одном потоке. Один и тот же поток выполняет все перечисленные ниже операции:

- принимает запрос на соединение по протоколу TCP;
- разбирает HTTP-запрос;
- вызывает контроллер или сервлет;
- блокируется, пока не будет получен ответ на запрос к базе данных;
- обрабатывает результаты;
- кодирует ответ (например, представляет его в формате JSON);
- отправляет последовательность байтов клиенту.

Из-за такой многоуровневой модели наблюдаемая пользователем задержка увеличивается, если сервер должен выполнить несколько запросов к базе данных. Все эти запросы производятся последовательно, хотя их было бы несложно распараллелить. Ко всему прочему, страдает еще и масштабируемость. Например, в Tomcat исполнителям, отвечающим за обработку запросов, по умолчанию отведено 200 потоков. Это означает, что больше 200 одновременных соединений быть не может.

В случае внезапного кратковременного всплеска трафика запросы на установление соединения ставятся в очередь и сервер отвечает с увеличенной задержкой. Но такая ситуация не может длиться вечно, так что рано или поздно Tomcat начнет отвергать входящие запросы. Значительную часть следующей главы мы посвятим вопросу о борьбе с этим досадным недостатком (см. раздел «Неблокирующий HTTP-сервер на основе Netty и RxNetty»). А пока не будем выходить за пределы традиционной архитектуры. У выполнения всех шагов обработки запроса в одном потоке есть свои преимущества, например, повышение эффективности кэширования и минимальные затраты на синхронизацию¹. К сожалению, в классических приложениях, где полная задержка равна сумме задержек на каждом уровне, единственный сбойный компонент может негативно сказаться на общей задержке². Кроме того, иногда есть много независимых шагов, которые можно было бы выполнить параллельно. Например, если вызывается несколько внешних API или выполняется несколько независимых SQL запросов.

JDK предлагает отличную поддержку конкурентности, особенно в форме класса `ExecutorService`, появившегося в Java 5, и класса `CompletableFuture` из Java 8. Тем не менее, она не так широко распространена, как того заслуживает. Взгляните, к примеру, на следующую программу, где нет вообще никакой конкурентности:

```
Flight lookupFlight(String flightNo) {
    //...
}

Passenger findPassenger(long id) {
    //...
}

Ticket bookTicket(Flight flight, Passenger passenger) {
    //...
}

SmtpResponse sendEmail(Ticket ticket) {
    //...
}
```

И на стороне клиента:

```
Flight flight = lookupFlight("LOT 783");
Passenger passenger = findPassenger(42);
Ticket ticket = bookTicket(flight, passenger);
sendEmail(ticket);
```

Вполне типичный, классический блокирующий код, какой можно встретить во многих приложениях. Но если рассмотреть его с точки зрения задержки, то окажется, что из четырех шагов первые два независимы. Лишь третий шаг – бронирование билета (`bookTicket()`) нуждается в результатах поиска рейса (`lookupFlight()`)

¹ На самом деле, RxJava также старается оставаться в одном потоке, пользуясь механизмом привязки к потоку в цикле обработки событий, – чтобы получить те же преимущества.

² См. также раздел «Паттерн Переборка и быстрое прекращение» главы 8.

и поиска пассажира (`findPassenger()`). Налицо очевидная возможность распараллелить выполнение. И тем не менее, немногие разработчики выбирают этот путь, потому что для него нужны такие неудобные пулы потоков, объекты `Future` и обратные вызовы. А если бы API был уже совместим с Rx? Напомню, что можно очень просто обернуть блокирующий унаследованный код объектом `Observable`, что мы и проделали в начале главы:

```
Observable<Flight> rxLookupFlight(String flightNo) {
    return Observable.defer(() ->
        Observable.just(lookupFlight(flightNo)));
}

Observable<Passenger> rxFindPassenger(long id) {
    return Observable.defer(() ->
        Observable.just(findPassenger(id)));
}
```

Семантически `rx`-методы делают то же самое и точно так же, т. е. по умолчанию являются блокирующими. Мы не выиграли ничего, кроме более многословного, с точки зрения клиента, API:

```
Observable<Flight> flight = rxLookupFlight("LOT 783");
Observable<Passenger> passenger = rxFindPassenger(42);
Observable<Ticket> ticket =
    flight.zipWith(passenger, (f, p) -> bookTicket(f, p));
ticket.subscribe(this::sendEmail);
```

И традиционные блокирующие программы, и программы с использованием `Observable` работают в точности одинаково. Сначала мы создаем объект `Observable<Flight>`, который, как мы уже знаем, по умолчанию ничего не делает. Пока кто-нибудь не запросит рейс `Flight`, этот `Observable` просто лениво сидит в памяти – и это весьма полезное свойство холодных объектов `Observable`. То же самое относится к `Observable<Passenger>`; мы имеем два потенциальных контейнера объектов `Flight` и `Passenger`, но никакие побочные эффекты еще не произведены. Не было ни запроса к базе данных, ни обращения к веб-сервису. Если мы решим на этом остановиться, то не проделаем никакой лишней работы.

Чтобы можно было выполнить метод `bookTicket()`, нам нужны конкретные экземпляры `Flight` и `Passenger`. Возникает соблазн просто заблокировать выполнение до завершения обоих этих `Observable`, воспользовавшись оператором `toBlocking()`. Но наша-то задача – по мере возможности избегать блокирования, чтобы снизить потребление ресурсов (особенно памяти) и обеспечить больший уровень конкурентности. Еще одно неудачное решение – подписаться на объекты `flight` и `passenger` методом `.subscribe()` и каким-то образом дождаться завершения обоих. Это довольно просто, когда `Observable` блокирующий, но если обратные вызовы происходят асинхронно, а нам нужно синхронизировать некоторое глобальное состояние ожидания, то задача быстро превращается в кошмар. К тому же, вложенные вызовы `subscribe()` – неидиоматический подход, обычно нам нужна одна подписка на один поток сообщений. В JavaScript обратные вызовы рабо-

тают прилично только потому, что там существует всего один поток. Идиоматический способ одновременно подписаться на несколько `Observable` дают операторы `zip` и `zipWith`. Вы, наверное, думаете, что `zip` служит для попарного объединения двух независимых потоков. Но гораздо чаще `zip` применяется просто для того, чтобы объединить два одноэлементных `Observable`. Конструкция `ob1.zip(ob2).subscribe(...)` означает получение события, когда оба объекта — `ob1` и `ob2` завершились (каждый породил свое событие). Так что встретившийся в программе `zip`, скорее всего, означает, что это просто шаг *объединения* двух или более объектов `observable`, следующий за *разветвлением* путей выполнения. Оператор `zip` — это способ асинхронного ожидания двух и более значений, в каком бы порядке они ни появились.

Но вернемся к строке `flight.zipWith(passenger, this::bookTicket)` (здесь использован более короткий синтаксис по сравнению с примером выше — ссылка на метод вместо лямбда-выражения). Я тащу за собой всю информацию о типах вместо текучего сцепления операторов, потому что хочу привлечь ваше внимание к типам возвращаемых значений. Метод `flight.zipWith(passenger, ...)` не просто вызывает функцию обратного вызова, когда `flight` и `passenger` завершатся, он еще и возвращает новый объект `Observable`, и вы наверняка узнали в нем ленивый контейнер для данных. Поразительно, но в этот момент вычисления все еще не начались. Мы лишь обернули несколько структур данных, но не запустили никаких действий. Пока на `observable<Ticket>` никто не подписался, RxJava не начинает выполнение кода. А подписываемся мы в последней строке: вызов `ticket.subscribe()` явно запрашивает билет `ticket`.



Где подписываться?

Обращайте внимание на то, в каком месте программы производится подписка. Часто бизнес-логика заключается в составлении композиции `Observable`, которая возвращается какому-то каркасу или уровню подмостей (*scaffolding*). А собственно подписка производится за кулисами в веб-каркасе или каком-то связующем коде. Ничего плохого в том, чтобы вызывать `subscribe()` самостоятельно, нет, но старайтесь откладывать это настолько, насколько возможно.

Чтобы понять, как выполняется этот код, полезно взглянуть на него снизу вверх. Мы подписались на `ticket`, поэтому RxJava должна самостоятельно подписаться на `flight` и `passenger`. В этот момент и начинается выполнение. Поскольку оба объекта `Observable` холодные и никакой конкурентности нет, то первая же подписка на `flight` приводит к вызову блокирующего метода `lookupFlight()` прямо в вызывающем потоке. По завершении этого метода RxJava может подписаться на `passenger`. Однако она уже получила экземпляр `flight` от синхронного объекта `flight`. Метод `rxFindPassenger()` вызывает `findPassenger()`, блокируя выполнение, и получает экземпляр `Passenger`. В точке соединения данные передаются дальше.

Экземпляры `Flight` и `Passenger` комбинируются с помощью переданного лямбда-выражения (`bookTicket`) и передаются методу `ticket.subscribe()`.

Создается впечатление, что возни много, а поведение-то получилось такое же, как в блокирующем коде, показанном в самом начале. Но теперь мы можем декларативно распараллелить работу, не меняя логику. Если бы наши бизнес-методы возвращали `Future<Flight>` (или `CompletableFuture<Flight>`, это неважно), то следующие два решения были бы приняты без нашего участия.

- Выполнение `lookupFlight()` уже началось и места для лени не осталось. Мы не блокируем программу в ожидании завершения такого метода, но работа уже начата.
- У нас нет никакого контроля над конкурентностью, лишь реализация метода решает, будет ли задача `Future` исполняться в пуле потоков, путем создания нового потока в ответ на запрос и т. д.

`RxJava` предоставляет пользователям более высокий уровень контроля. Из того, что при реализации типа `Observable<Flight>` конкурентность не закладывалась, вовсе не следует, что мы не сможем применить ее позже. На практике объекты `Observable`, как правило, уже асинхронны, но в редких случаях приходится надеяться на асинхронностью существующий `Observable`. Потребители нашего API – заметьте, не разработчики – свободны в выборе механизма многопоточности в случае синхронного `Observable`. Все это достигается с помощью оператора `subscribeOn()`:

```
Observable<Flight> flight =  
    rxLookupFlight("LOT 783").subscribeOn(Schedulers.io());  
Observable<Passenger> passenger =  
    rxFindPassenger(42).subscribeOn(Schedulers.io());
```

В любой момент до подписки мы можем вставить оператор `subscribeOn()`, передав ему экземпляр класса диспетчера `Scheduler`. В данном случае я воспользовался фабричным методом `Schedulers.io()`, с тем же успехом можно было взять пользовательский класс `ExecutorService` и обернуть его объектом `Scheduler`. После подписки лямбда-выражение, переданное методу `Observable.create()`, выполняется в контексте предоставленного `Scheduler`, а не в потоке клиента. Сейчас в этом нет необходимости, но в разделе «Что такое диспетчер?» ниже мы рассмотрим этот вопрос подробно. Ну а пока будем рассматривать `Scheduler` как пул потоков.

Как `Scheduler` изменяет поведение нашей программы во время выполнения? Напомню, что оператор `zip()` подписывается на два или более объектов `Observable` и ждет появления пар (или кортежей). Если подписка асинхронная, то все предшествующие `Observable` могут одновременно вызывать свой блокирующий код. Если теперь запустить программу, то методы `lookupFlight()` и `findPassenger()` начнут исполнение одновременно сразу после вызова `ticket.subscribe()`. А метод `bookTicket()` будет вызван, как только более медленный из этих двух `Observable` породит значение.

И кстати о медлительности: мы можем декларативно объявить таймаут на случай, если `Observable` не породит никакого значения в отведенное время:


```
rxLookupFlight("LOT 783")  
    .subscribeOn(Schedulers.io())  
    .timeout(100, TimeUnit.MILLISECONDS)
```

Исключения, как всегда, распространяются вниз, а не возбуждаются в произвольном месте. Так, если метод `lookupFlight()` будет выполняться дольше 100 миллисекунд, то каждый подписчик получит исключение `TimeoutException`, а не порожденное значение. Оператор `timeout()` детально рассматривается в разделе «Таймаут в случае отсутствия событий» главы 7.

Итак, не приложив особых усилий, мы получили два метода, работающих конкурентно, — в предположении, что API уже основан на Rx. Но немного смухлевали — метод `bookTicket()` все еще возвращает `Ticket`, а это значит, что он блокирующий. Даже если бронирование билетов работает молниеносно, все равно имеет смысл объявить, что метод возвращает `Observable`, облегчив себе дальнейшее развитие API. Развитие может означать как добавление конкурентности, так и использование в полностью неблокирующем окружении (см. главу 5). Напомню, что для преобразования неблокирующего API в блокирующий достаточно вызвать метод `toBlocking()`. Обратное зачастую куда труднее и требует немало дополнительных ресурсов. К тому же, очень сложно спрогнозировать эволюцию методов типа `rxBookTicket()`, но если они как-то обращаются к сети или файловой системе, не говоря уже о базе данных, то имеет смысл сразу обернуть их в `Observable`, показав на уровне типа, что возможна задержка:

```
Observable<Ticket> rxBookTicket(Flight flight, Passenger passenger) {  
    //...  
}
```

Но теперь `zipWith()` возвращает громоздкий `Observable<Observable<Ticket>>`, и код перестал компилироваться. Есть хорошее эвристическое правило: если видишь дважды обернутый тип (например, `Optional<Optional<...>>`), значит, где-то не хватает вызова `flatMap()`. Так обстоит дело и здесь. Оператор `zipWith()` принимает пару (в общем случае кортеж) событий, применяет функцию, которой эти события передаются в качестве аргументов, и передает результат дальше без каких-либо изменений. Потому-то мы сначала и видели тип `Observable<Ticket>`, а теперь видим `Observable<Observable<Ticket>>`, где `Observable<Ticket>` — тип значения, возвращенного переданной нами функцией. Решить проблему можно двумя способами. Первый — использовать промежуточную пару, возвращенную оператором `zipWith()`:

```
import org.apache.commons.lang3.tuple.Pair;  
  
Observable<Ticket> ticket = flight  
    .zipWith(passenger, (Flight f, Passenger p) -> Pair.of(f, p))  
    .flatMap(pair -> rxBookTicket(pair.getLeft(), pair.getRight()));
```

Если явное использование класса `Pair` из сторонней библиотеки, на ваш взгляд, еще не сделало код достаточно замысловатым, то можно воспользоваться ссылкой

на метод `Pair::of`, но мы, напомним, решили, что показать информацию о типе ценнее, чем сэкономить несколько ударов по клавишам. В конце концов, читаем мы код намного чаще, чем пишем. Альтернативой промежуточной паре является применение оператора `flatMap` с тождественной функцией:

```
Observable<Ticket> ticket = flight
    .zipWith(passenger, this::rxBookTicket)
    .flatMap(obs -> obs);
```

На первый взгляд, лямбда-выражение `obs -> obs` ничего не делает. По крайней мере, так было бы, если бы оно использовалось в операторе `map()`. Но напомним, что `flatMap()` применяет функцию, к каждому значению, порожденному `Observable`, т. е. эта функция принимает в качестве аргумента `Observable<Ticket>`. Но результат не помещается сразу в выходной поток, как в случае `map()`. Вместо этого возвращенное значение (тип `Observable<T>`) сериализуется, что дает `Observable<T>`, а не `Observable<Observable<T>>`. При работе с диспетчерами оператор `flatMap()` оказывается еще более полезным. Можете считать, что `flatMap()` – просто синтаксический трюк, который позволяет избежать вложенных типов `Observable<Observable<...>>`, но на самом деле его смысл гораздо глубже.



Применение `Observable.subscribeOn()` на практике

Соблазнительно думать, что оператор `subscribeOn()` – подходящий инструмент для организации конкурентности в `RxJava`. Он работает, но реальные возможности для применения `subscribeOn()` (и оператора `observeOn()`, который будет описан ниже) встречаются не так часто. На практике объекты `Observable` происходят из асинхронных источников, поэтому пользовательская диспетчеризация и не нужна вовсе. В этой главе мы используем `subscribeOn()`, чтобы показать, как можно избирательно перевести существующие приложения на принципы реактивного программирования. Но в реальной жизни диспетчеры и `subscribeOn()` – прибежище последней надежды, а не что-то, используемое постоянно.

`flatMap()` как оператор асинхронного сцепления

В нашем примере следующий шаг – отправка списка билетов `ticket` по электронной почте. Но нужно иметь в виду следующее:

1. Список может оказаться очень длинным.
2. Отправка сообщения может занять несколько миллисекунд или даже секунд.
3. Приложение должно не завершаться немедленно в случае ошибок, а продолжать работу и в самом конце сообщать, какие билеты не удалось доставить.

Последнее требование заставляет отказаться от простого решения `tickets.forEach(this::sendEmail)`, поскольку в этом случае энергично возбуждается исключение, и мы не можем продолжить обработку еще не доставленных билетов. На самом деле, исключения – это мерзкий черный ход в системе типов; как и обратные вызовы, они не слишком удобны, когда требуется надежное поведение. Именно поэтому RxJava моделирует их как уведомления специального вида, но не торопитесь, всему свое время. С учетом требования к обработке ошибок наш код должен иметь примерно такой вид:

```
List<Ticket> failures = new ArrayList<>();
for(Ticket ticket: tickets) {
    try {
        sendEmail(ticket);
    } catch (Exception e) {
        log.warn("Не удалось отправить {}", ticket, e);
        failures.add(ticket);
    }
}
```

Но мы не подумали о первых двух требованиях. Не видно причин, по которым отправлять письма нужно последовательно, в одном потоке. Традиционно мы воспользовались бы для этого пулом `ExecutorService` и передавали бы ему каждое письмо в виде отдельной задачи:

```
List<Pair<Ticket, Future<SmtpResponse>>> tasks = tickets
    .stream()
    .map(ticket -> Pair.of(ticket, sendEmailAsync(ticket)))
    .collect(toList());

List<Ticket> failures = tasks.stream()
    .flatMap(pair -> {
        try {
            Future<SmtpResponse> future = pair.getRight();
            future.get(1, TimeUnit.SECONDS);
            return Stream.empty();
        } catch (Exception e) {
            Ticket ticket = pair.getLeft();
            log.warn("Failed to send {}", ticket, e);
            return Stream.of(ticket);
        }
    })
    .collect(toList());

//-----
private Future<SmtpResponse> sendEmailAsync(Ticket ticket) {
    return pool.submit(() -> sendEmail(ticket));
}
```

Такой код должен быть знаком любому программисту на Java. Но он выглядит слишком многословным и без надобности сложным. Сначала, мы перебираем билеты в коллекции `tickets` и передаем их пулу потоков. Точнее, мы вызываем вспо-

могательный метод `sendEmailAsync()`, который передает пулу потоков `pool` вызов метода `sendEmail()`, обернутый объектом `Callable<SmtResponse>`. А если еще точнее, то экземпляры `Callable` сначала помещаются в неограниченную (по умолчанию) очередь к пулу. Отсутствие механизмов, позволяющих замедлить слишком быстрое порождение задач, которые невозможно обработать вовремя, как раз и стало стимулом для разработки реактивных потоков и противодействия (см. раздел «Противодавление» главы 6).

Поскольку экземпляр `Ticket` понадобится нам в случае ошибки, мы должны запомнить, какой объект `Future` за какой `Ticket` отвечает, для этого мы снова воспользуемся классом `Pair`. В реальной программе надо было бы завести более осмысленный контейнер, например, для объектов типа `TicketAsyncTask`. Мы собираем все такие пары и переходим к следующему шагу. В этот момент пул потоков уже конкурентно исполняет несколько вызовов `sendEmail()`, чего мы и добивались. Во втором цикле мы перебираем все объекты `Future` и пытаемся разыменовать их, вызывая блокирующий метод `get()` и дожидаясь завершения. Если `get()` завершается успешно, то соответствующий `Ticket` пропускается. Но если возникло исключение, то возвращается экземпляр `Ticket`, ассоциированный с данной задачей, — мы знаем, что доставить его не удалось, и хотим сообщить об этом позже. Метод `Stream.flatMap()` позволяет вернуть нуль или один элемент (вообще-то, сколько угодно), в отличие от метода `Stream.map()`, который всегда возвращает ровно один элемент.

Возможно, вам не дает покоя вопрос, зачем нужно два цикла вместо одного такого:

```
// ПРЕДУПРЕЖДЕНИЕ: этот код последовательный, несмотря на использование
// пула потоков
List<Ticket> failures = tickets
    .stream()
    .map(ticket -> Pair.of(ticket, sendEmailAsync(ticket)))
    .flatMap(pair -> {
        //...
    })
    .collect(toList());
```

Это интересная ошибка, которую трудно найти, если не знать, как работают потоки `Stream` в Java 8. Поскольку потоки, как и объекты `Observable`, ленивые, они обрабатывают исходную коллекцию по одному элементу за раз и только в тот момент, когда запрошена терминальная операция (например, `collect(toList())`). Это означает, что операция `map()`, запускающая фоновые задачи, не применяется сразу ко всем билетам. Билеты обрабатываются по одному, и за каждым вызовом `map()` следует `flatMap()`. Кроме того, мы в действительности запускаем первый объект `Future`, дожидаясь его завершения, затем запускаем второй и т. д. Промежуточная коллекция нужна для того, чтобы инициировать конкурентные вычисления, а вовсе не для ясности или удобочитаемости. Если уж говорить об удобочитаемости, то тип `List<Pair<Ticket, Future<SmtResponse>>>` куда хуже.

Работы много, и вероятность ошибки высока, поэтому не удивительно, что разработчики неохотно пишут конкурентный код в повседневной практике. В тех случаях,

когда имеется пул асинхронных задач, и мы хотим обрабатывать их по мере завершения, иногда применяется малоизвестный класс `ExecutorCompletionService` из JDK (<https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorCompletionService.html>). В Java8 имеется класс `CompletableFuture` (см. раздел «CompletableFuture и потоки» главы 5), стопроцентно реактивный и неблокирующий. Но чем здесь может помочь RxJava? Для начала предположим, что API отправки почтового сообщения уже доработан под RxJava:

```
import static rx.Observable.fromCallable;

Observable<SmtпResponse> rxSendEmail(Ticket ticket) {
    // необычный синхронный Observable
    return fromCallable() -> sendEmail()
}
```

Здесь нет никакой конкурентности, просто вызов `sendEmail()` обернут объектом `Observable`. Такой `Observable` встречается редко; в типичной ситуации мы воспользовались бы методом `subscribeOn()`, так чтобы `Observable` по умолчанию был асинхронным. Теперь мы можем, как и раньше, перебрать все билеты:

```
List<Ticket> failures = Observable.from(tickets)
    .flatMap(ticket ->
        rxSendEmail(ticket)
        .flatMap(response -> Observable.<Ticket>empty())
        .doOnError(e -> log.warn("Не удалось отправить {})", ticket, e))
        .onErrorReturn(err -> ticket))
    .toList()
    .toBlocking()
    .single();
```



Оператор `Observable.ignoreElements()`

Легко видеть, что внутренний `flatMap()` в нашем примере игнорирует ответ и возвращает пустой поток. В таких случаях использование `flatMap()` – перебор, гораздо эффективнее оператор `ignoreElements()`. Этот оператор просто игнорирует все порожденные значения, а уведомления `onCompleted()` и `onError()` переправляет дальше. Поскольку ответы нас не интересуют, а важны только ошибки, `ignoreElements()` здесь прекрасно подошел бы.

Все, что нас интересует, находится во внешнем `flatMap()`. Если бы это был просто `flatMap(this::rxSendEmail)`, то программа работала бы, но любая ошибка, порожденная `rxSendEmail`, завершила бы весь поток. Однако мы-то хотим перехватывать все ошибки и запоминать их для использования в будущем. Мы применили прием, аналогичный методу `Stream.flatMap()`: если ответ `response` порожден успешно, то мы преобразуем его в пустой `Observable`. По сути дела, это означает, что успешно отправленные билеты отбрасываются. А в случае ошибки мы возвращаем объект `ticket`, приведший к исключению. Дополнительный обратный вызов `doOnError()`

позволяет за протоколировать исключение. Разумеется, мы с тем же успехом могли бы добавить протоколирование в оператор `onErrorReturn()`, но я решил, что такое разделение обязанностей правильное.

Чтобы сохранить совместимость с предыдущими реализациями, мы преобразовываем `Observable` в `Observable<List<Ticket>>`, `BlockingObservable<List<Ticket>>`, `toBlocking()` и, наконец, в `List<Ticket>` (оператор `single()`). Интересно, что даже `BlockingObservable` остается ленивым. Сам по себе оператор `toBlocking()` не инициирует вычисления, т. к. не подписывается на входной поток. Он даже не блокирует выполнение. Подписка, а, стало быть, обход билетов и отправка почтовых сообщений откладываются до момента вызова `single()`.

Отметим, что если заменить внешний `flatMap()` на `concatMap()` (см. разделы «Способы комбинирования потоков: `concat()`, `merge()` и `switchOnNext()`» и «Сохранение порядка с помощью `concatMap()`» главы 3), то мы столкнемся с ошибкой, похожей на описанную выше при рассмотрении класса `Stream` из JDK. В отличие от оператора `flatMap()` (или `merge()`), который подписывается сразу на все внутренние потоки, `concatMap()` (и `concat()`) подписывается на один внутренний `Observable` за другим поочередно. И пока никто не подписался на `Observable`, работа не начинается.

Пока что мы заменили простой цикл с блоком `try-catch` не столь удобочитаемым и более сложным объектом `observable`. Зато для превращения нашего последовательного кода в многопоточное вычисление нужно добавить всего один оператор:

```
Observable
    .from(tickets)
    .flatMap(ticket ->
        rxSendEmail(ticket)
        .ignoreElements()
        .doOnError(e -> log.warn("Не удалось отправить {}", ticket, e))
        .onErrorReturn(err -> ticket)
        .subscribeOn(Schedulers.io()))
```

Его даже не сразу заметишь. Один дополнительный оператор `subscribeOn()` приводит к тому, что все методы `rxSendMail()` исполняются в контексте указанного диспетчера `Scheduler` (в данном случае `io()`). Это одна из сильных сторон библиотеки `RxJava`; она не отдает никакого особого предпочтения потокам, по умолчанию выполняется синхронно, но позволяет естественно и почти прозрачно переключиться в многопоточный режим. Разумеется, это не означает, что можно, не задумываясь, втыкать диспетчеры в любое место. Но, по крайней мере, API лаконичнее и более высокого уровня. Мы еще будем гораздо подробнее рассматривать диспетчеры в разделе «Многопоточность в `RxJava`» ниже. А пока запомните, что объекты `Observable` по умолчанию синхронны, но это можно легко изменить и добавить конкурентность в совершенно неожиданные места. Особенно это ценно при работе с унаследованными приложениями, которые можно оптимизировать без особых хлопот.

Если же вы реализуете объекты `Observable` с чистого листа, то более идиоматичным является обертывание с целью сделать их асинхронными по умолчанию.

Это означает, что вызов `subscribeOn()` нужно поместить внутрь, а не снаружи `rxSendEmail()`. Иначе вы рискуете обернуть уже асинхронный поток еще одним слоем диспетчеров. Конечно, если производитель, стоящий за `Observable`, уже асинхронный, то это даже лучше, потому что ваш поток событий (`stream`) не привязан ни к какому потоку выполнения (`thread`). Кроме того, следует откладывать подписку на `Observable` до самого последнего момента, как правило, близко к веб-каркасу, глядящему во внешний мир. Это существенное изменение подхода к программированию. Вся ваша бизнес-логика лениво ждет, пока кто-то не захочет увидеть результат³.

Замена обратных вызовов потоками

Традиционные API являются по большей части блокирующими, т. е. вынуждают программу синхронно ждать результатов. Такой подход более-менее приемлем, по крайней мере, для тех, кто не слышал о RxJava. Но блокирующий API весьма проблематичен в случае, когда данные нужно проталкивать из API производителя потребителям, — и это как раз та область, где возможности RxJava раскрываются в полной мере. Есть много примеров таких ситуаций, и проектировщики API применяют разные подходы. Как правило, от нас требуется предоставить тот или иной вид обратного вызова, который будет вызываться из API; в этом контексте часто употребляют термин «прослушиватель». Один из самых типичных сценариев такого рода — служба сообщений Java (Java Message Service — JMS)⁴. Потребление JMS обычно сводится к реализации класса, который сервер приложений или контейнер уведомляет о каждом поступающем сообщении. Без особых сложностей такие прослушиватели можно заменить композицией объектов `Observable`, что обеспечивает большую гибкость и надежность. Традиционный класс прослушивателя выглядит, как показано ниже. Мы воспользовались поддержкой JMS в каркасе Spring (<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/jms.html>), но вообще решение не зависит от технологии:

```
@Component
class JmsConsumer {

    @JmsListener(destination = "orders")
    public void newOrder(Message message) {
        //...
    }
}
```

Получив от JMS сообщение `message`, класс `JmsConsumer` должен решить, что с ним делать. Обычно из него вызывается какая-то бизнес-логика. Если новый компонент захочет получать уведомления о таких сообщениях, придется соответственно модифицировать `JmsConsumer`. А теперь представим себе объект `Observable<Message>`, на который может подписаться кто угодно. При этом нам

³ Сравните с ленивым вычислением выражений в языке Haskell.

⁴ <http://docs.oracle.com/javaee/6/tutorial/doc/bncdq.html>

доступна вся совокупность операторов RxJava, открывающая возможности отбора, фильтрации и комбинирования. Простейший способ перейти от API на основе обратных вызовов к Observable – воспользоваться темами, Subject. Каждое вновь полученное сообщение JMS передается объекту PublishSubject, который внешне выглядит как обыкновенный горячий Observable:

```
private final PublishSubject<Message> subject = PublishSubject.create();

@JmsListener(destination = "orders", concurrency="1")
public void newOrder(Message msg) {
    subject.onNext(msg);
}

Observable<Message> observe() {
    return subject;
}
```

Помните, что Observable<Message> – горячий объект; он порождает сообщения JMS сразу по мере их появления. Если в этот момент никто не подписан, сообщения просто теряются. Альтернатива – класс ReplaySubject, но он кэширует все события с момента запуска приложения и потому не годится для использования в долго работающих процессах. Если какому-то подписчику абсолютно необходимо получать все сообщения, то он должен подписаться до того, как будет инициализирован прослушиватель сообщений JMS. Кроме того, у нашего прослушивателя имеется параметр concurrency="1", который гарантирует, что Subject не будет вызываться из нескольких потоков. Вместо этого можно было бы воспользоваться методом Subject.toSerialized().

Попутно отметим, что поначалу объекты Subject кажутся очень простыми, но хорошо известно, что спустя некоторое время начинают возникать проблемы. В данном случае легко заменить Subject более идиоматичным объектом Observable, который вызывает create() непосредственно:

```
public Observable<Message> observe(
    ConnectionFactory connectionFactory, Topic topic) {
    return Observable.create(subscriber -> {
        try {
            subscribeThrowing(subscriber, connectionFactory, topic);
        } catch (JMSEException e) {
            subscriber.onError(e);
        }
    });
}

private void subscribeThrowing(
    Subscriber<? super Message> subscriber,
    ConnectionFactory connectionFactory,
    Topic orders) throws JMSEException {
    Connection connection = connectionFactory.createConnection();
    Session session = connection.createSession(true, AUTO_ACKNOWLEDGE);
    MessageConsumer consumer = session.createConsumer(orders);
```



```

consumer.setMessageListener(subscriber::onNext);
subscriber.add(onUnsubscribe(connection));
connection.start();
}

private Subscription onUnsubscribe(Connection connection) {
    return Subscriptions.create(() -> {
        try {
            connection.close();
        } catch (Exception e) {
            log.error("Ошибка при закрытии", e);
        }
    });
}
}

```

JMS API предоставляет два способа получения сообщений от брокера: синхронный – с помощью блокирующего метода `receive()` – и неблокирующий класс `MessageListener`. Неблокирующий API предпочтительнее по многим причинам, например, потому что он захватывает меньше ресурсов: потоков и памяти в стеке. Кроме того, он отлично согласуется со стилем программирования, принятым в Rx. Чем создавать экземпляр `MessageListener` и вызывать из него наш подписчик, мы можем воспользоваться лаконичным синтаксисом ссылки на метод:

```
consumer.setMessageListener(subscriber::onNext)
```

Кроме того, нужно позаботиться об освобождении ресурсов и обработке ошибок. Показанный ниже крохотный слой преобразования позволяет без труда потреблять сообщения JMS, не задумываясь о внутреннем устройстве API. В этом примере используется популярный брокер сообщений `ActiveMQ` (<http://activemq.apache.org/>), работающий локально:

```

import org.apache.activemq.ActiveMQConnectionFactory;
import org.apache.activemq.command.ActiveMQTopic;

ConnectionFactory connectionFactory =
    new ActiveMQConnectionFactory("tcp://localhost:61616");
Observable<String> txtMessages =
    observe(connectionFactory, new ActiveMQTopic("orders"))
        .cast(TextMessage.class)
        .flatMap(m -> {
            try {
                return Observable.just(m.getText());
            } catch (JMSEException e) {
                return Observable.error(e);
            }
        });
}

```

JMS, как и JDBC, повсеместно использует контролируемые исключения, в данном случае `JMSEException`, даже при вызове метода `getText()` объекта `TextMessage`. Для их надлежащей обработки (см. раздел «Обработка ошибок» главы 7) мы пользуемся оператором `flatMap()` и обортываем исключения. Теперь можно рассмат-

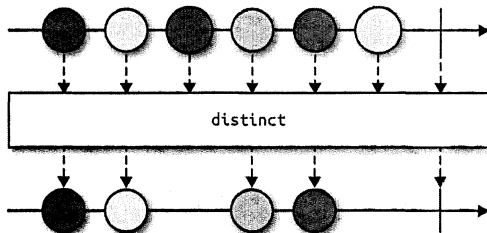
вать входящие сообщения JMS наравне с любым другим асинхронным неблокирующим потоком. И, кстати, мы воспользовались оператором `cast()`, который пытается привести входящее событие к указанному типу, но вызывает `onError()`, если это невозможно. По существу, `cast()` — это специализированный оператор `map()`, который ведет себя как `map(x -> (TextMessage)x)`.

Периодический опрос изменений

Самый худший случай блокирующего API — тот, в котором требуется опрашивать систему, чтобы узнать об изменениях. Не предлагается никакого механизма проталкивания изменений, даже в виде обратных вызовов или неопределенно долгого блокирования. API позволяет только узнать текущее состояние, а дальше уж вам решать, отличается оно от предыдущего или нет. В RxJava есть несколько весьма мощных операторов, которые дают возможность модернизировать такой API, приспособив его к стилю Rx. Первым я хочу рассмотреть простой метод, который доставляет единственное значение, представляющее состояние, например `long getOrderBookLength()`. Для отслеживания изменений мы должны вызывать этот метод достаточно часто и распознавать отличия. В RxJava этого можно добиться простейшей композицией операторов:

```
Observable
    .interval(10, TimeUnit.MILLISECONDS)
    .map(x -> getOrderBookLength())
    .distinctUntilChanged()
```

Каждые 10 миллисекунд мы порождаем синтетическое значение типа `long`, выступающее в роли простого счетчика тиков. Для каждого такого значения (т. е. каждые 10 миллисекунд) мы вызываем метод `getOrderBookLength()`. Однако в действительности отслеживаемое значение изменяется не так часто, и мы не хотим затоплять подписчиков ненужной информацией о несостоявшейся смене состояния. По счастью, можно просто добавить оператор `distinctUntilChanged()`, и RxJava будет пропускать значения, возвращенные методом `getOrderBookLength()`, если они не изменились по сравнению с предыдущим вызовом. Работу этого оператора иллюстрирует следующая диаграмма:



Этот принцип можно распространить и на другие ситуации. Допустим, требуется наблюдать за изменениями в файловой системе или в таблице базы данных. Единственный имеющийся в нашем распоряжении механизм – сделать полный снимок состояния файлов или записей базы. Мы разрабатываем API, который будет уведомлять клиентов о появлении новых элементов. Понятно, что можно было бы воспользоваться классом `java.nio.file.WatchService` или триггерами базы данных, но это лишь учебный пример. Снова будем периодически опрашивать текущее состояние:

```
Observable<Item> observeNewItems() {
    return Observable
        .interval(1, TimeUnit.SECONDS)
        .flatMapIterable(x -> query())
        .distinct();
}

List<Item> query() {
    // сделать снимок каталога файловой системы
    // или таблицы базы данных
}
```

Оператор `distinct()` хранит все элементы, которые через него проходили (см. также раздел «Устранение дубликатов с помощью `distinct()` и `distinctUntilChanged()`» главы 3). Если поступает элемент, который уже встречался, то он игнорируется. Поэтому мы можем передавать один и тот же список элементов `Item` каждую секунду. В первый раз все элементы отправляются всем подписчикам. Но если точно такой же список появляется спустя секунду, то все элементы отбрасываются, потому что встречались раньше. Если в какой-то момент окажется, что список, возвращенный методом `query()`, содержит новый элемент, то `distinct()` пропустит его, но отбросит в следующий раз. Этот простой подход позволяет заменить многочисленные вызовы `Thread.sleep()` и периодический опрос с кэшированием вручную. Он применим во многих ситуациях, например: опрос по протоколу FTP, скрапинг веб-сайтов и т. д.

Многопоточность в RxJava

Существуют сторонние блокирующие API, и с этим мы ничего не можем поделать. Возможно, у нас нет исходного кода или его переписывание слишком рискованно. В таком случае мы должны научиться жить с блокирующим кодом, а не бороться с ним.

Одна из отличительных особенностей RxJava – декларативная конкурентность, противопоставляемая императивной. Создание и управление потоками вручную осталось в прошлом (см. раздел «Пул соединений» в приложении А), и мы уже давно пользуемся управляемыми пулами потоков (например, с помощью класса `ExecutorService`). Но RxJava идет на шаг дальше: объект `Observable` может быть неблокирующим, как `CompletableFuture` в Java 8 (см. раздел «`CompletableFuture` и

потоки» главы 5), но в отличие от других, он еще и ленивый. Пока нет подписчиков, правильно написанный `Observable` не выполняет никаких действий. Однако на этом возможности `Observable` не кончаются.

Асинхронный `Observable` – это такой объект, который вызывает методы обратного вызова подписчиков (например, `onNext()`) из другого потока. Вспомните раздел «Подробнее о методе `Observable.create()`» главы 2, где мы изучали, когда метод `subscribe()` является блокирующим, т. е. ожидает поступления всех уведомлений. На практике большинство объектов `Observable` связаны с источниками, которые по природе своей асинхронны. Глава 5 целиком посвящена таким `Observable`. К их числу относится даже наш простой пример `JMS` из раздела «Замена обратных вызовов потоками» выше, в котором используется встроенный неблокирующий `API`, описанный в спецификации `JMS` (интерфейс `MessageListener`). Многие `Observable` изначально асинхронны, хотя система типов этого не навязывает и даже не предлагает. Об этом следует помнить. Блокирующие методы `subscribe()` встречаются очень редко, когда за лямбда-выражением, переданным методу `Observable.create()`, не стоит асинхронный процесс или поток. Но по умолчанию (при использовании `create()`) вся работа производится в потоке клиента (том, в котором имела место подписка). Даже если вызвать `onNext()` прямо в `create()`, никакой многопоточности не последует.

Встретив такой необычный `Observable`, мы можем декларативно выбрать так называемый диспетчер – объект типа `Scheduler`, – который будет применяться для порождения значений. В случае `CompletableFuture` мы не можем контролировать потоки, все решения принимает `API` и в худшем случае их невозможно переопределить. `RxJava` редко принимает такие решения самостоятельно, а по умолчанию выбирает безопасный вариант: поток клиента и никакой многопоточности. В этой главе мы воспользуемся простенькой «библиотекой» протоколирования⁵, которая печатает сообщение, а вместе с ним имя текущего потока и количество секунд с момента запуска программы (пользуясь методом `System.currentTimeMillis()`):

```
void log(Object label) {
    System.out.println(
        System.currentTimeMillis() - start + "\t| " +
        Thread.currentThread().getName() + "\t| " +
        label);
}
```

Что такое диспетчер?

`RxJava` безразлична к конкурентности и не вносит никакую конкурентность по своей инициативе. Однако некоторые абстракции, относящиеся к потокам, видны конечному пользователю. Кроме того, некоторые операторы не могут работать нормально в отсутствие конкурентности (часть из них упоминается в разделе «Другие применение диспетчеров» ниже). По счастью, класс `Scheduler` – единст-

⁵ Понятно, что в реальном проекте вы взяли бы какую-нибудь зарекомендовавшую себе систему протоколирования типа `Logback` (<http://logback.qos.ch/>) или `Log4J 2` (<http://logging.apache.org/log4j/2.x/>).

венный, о котором вам нужно знать, — относительно прост. В принципе, он работает похоже на класс `ScheduledExecutorService` из пакета `java.util.concurrent` — исполняет произвольный блок кода, возможно, в будущем. Но чтобы удовлетворить контракт Rx, он предлагает ряд более мелких абстракций, о которых мы поговорим в разделе «Обзор деталей реализации диспетчера».

Диспетчеры используются совместно с операторами `subscribeOn()` и `observeOn()`, а также при создании некоторых типов `Observable`. Диспетчер всего лишь создает экземпляры класса `Worker`, отвечающие за планирование и выполнение кода. Когда RxJava требуется запланировать выполнение какого-то кода, она сначала просит у `Scheduler` экземпляр `Worker`, а затем пользуется им для планирования последующих задач. Примеры работы с этим API мы приведем позже, а пока познакомимся со встроенными диспетчерами.

```
Schedulers.newThread()
```

Этот диспетчер просто запускает новый поток всякий раз, как его запрашивает `subscribeOn()` или `observeOn()`. Метод `newThread()` не назовешь удачным выбором — и не только из-за задержки, неотъемлемой от создания потока, но и потому что этот поток не используется повторно. Память для стека необходимо выделить заранее (обычно около одного мегабайта, этим управляет параметр JVM `-Xss`), и операционная система должна создать новый поток. Когда `Worker` завершает работу, поток тоже завершается. Этот диспетчер полезен только для крупномодульных задач: они работают долго, но их мало, так что маловероятно, что необходимость в повторном использовании вообще возникнет. См. также раздел «Один поток — одно подключение» в приложении А. На практике почти всегда лучше планировщик, возвращаемый методом `Schedulers.io()`.

```
Schedulers.io()
```

Этот диспетчер похож на `newThread()`, но ранее запущенные потоки могут использоваться повторно для обработки последующих запросов. Реализация ведет себя во многом так же, как класс `ThreadPoolExecutor` из пакета `java.util.concurrent` с неограниченным пулом потоков. Когда запрашивается новый `Worker`, диспетчер либо создает новый поток (который по завершении работы некоторое время остается в пуле), либо повторно использует свободный поток из пула. Имя `io()` выбрано не случайно. Этот диспетчер имеет смысл использовать для задач, ограниченных скоростью ввода-вывода, которые потребляют очень мало ресурсов процессора. Однако они проводят некоторое время в ожидании сети или диска. Поэтому рекомендуется отводить для них сравнительно большой пул потоков. В то же время будьте осторожны при работе с неограниченными ресурсами любого вида — в случае медленных или не отвечающих внешних источников, например веб-сервисов, диспетчер `io()` может насоздавать огромное количество потоков, и тогда уже перестанет отвечать ваше приложение. См. раздел «Управление отказами с помощью *Nystrix*» главы 8, где более подробно описано, как бороться с этой проблемой.

```
Schedulers.computation()
```

Этот диспетчер следует использовать для счетных задач, которые нуждаются в вычислительных ресурсах и не содержат блокирующего кода (чтения с диска или из сети, засыпания, ожидания блокировки и т. д.). Поскольку предполагается, что задачи, управляемые этим диспетчером, полностью задействуют одно процессорное ядро, не имеет смысла параллельно выполнять больше задач, чем имеется ядер. Поэтому по умолчанию число потоков, создаваемых диспетчером `computation()`, не превышает величины, которую возвращает метод `availableProcessors()` из служебного класса `Runtime.getRuntime()`.

Если по какой-то причине нужно меньше потоков, то всегда можно задать системное свойство `rx.scheduler.max-computation-threads`. Уменьшив число потоков по сравнению с умолчанием, мы гарантируем, что одно или несколько процессорных ядер будут свободны, так что при высокой нагрузке пул потоков `computation()` не исчерпает всю вычислительную мощность сервера. Ни при каких условиях невозможно создать больше вычислительных потоков, чем ядер.

В диспетчере `computation()` используется неограниченная очередь к пулу, поэтому если все ядра заняты, то задача ставится в очередь. В периоды пиковой нагрузки диспетчер ограничивает число потоков, но очередь все равно продолжает расти.

К счастью, встроенные операторы, особенно `observeOn()`, который мы будем рассматривать в разделе «Декларативная конкурентность с помощью `observeOn()`» ниже, гарантируют, что этот диспетчер не окажется перегруженным.

```
Schedulers.from(Executor executor)
```

Диспетчеры `Scheduler` устроены сложнее, чем исполнители `Executor` из пакета `java.util.concurrent`, поэтому понадобилась отдельная абстракция. Но поскольку концептуально они очень похожи, не удивительно, что имеется обертка, преобразующая `Executor` и `Scheduler` с помощью фабричного метода `from()`:

```
import com.google.common.util.concurrent.ThreadFactoryBuilder;
import rx.Scheduler;
import rx.schedulers.Schedulers;
```

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ThreadFactory;
import java.util.concurrent.ThreadPoolExecutor;
```

```
//...
```

```
ThreadFactory threadFactory = new ThreadFactoryBuilder()
    .setNameFormat("MyPool-%d")
    .build();
Executor executor = new ThreadPoolExecutor(
```

```
10, //corePoolSize
10, //maximumPoolSize
0L, TimeUnit.MILLISECONDS, //keepAliveTime, unit
new LinkedBlockingQueue<>(1000), //workQueue
threadFactory
);
Scheduler scheduler = Schedulers.from(executor);
```

Я сознательно применил для создания `ExecutorService` развернутый синтаксис, а не более простой вариант:

```
import java.util.concurrent.Executors;
//...
ExecutorService executor = Executors.newFixedThreadPool(10);
```

В фабричных классах из пакета `Executors` зашито несколько умолчаний, которые бесполезны или даже опасны в корпоративных приложениях. Например, используется очередь `LinkedBlockingQueue`, которая может расти неограниченно, что ведет к исключению `OutOfMemoryError`, если ожидающих задач становится слишком много. Кроме того, класс `ThreadFactory` по умолчанию создает потоки с бессмысленными именами вида `pool-5-thread-3`. Выбор хороших имен для потоков крайне важен при профилировании или анализе дампов памяти. Реализовывать `ThreadFactory` с нуля довольно муторно, поэтому мы воспользовались классом `ThreadFactoryBuilder` из библиотеки `Guava` (<https://github.com/google/guava>). Если вам интересно, как настраивать и правильно использовать пулы потоков, почитайте раздел «Пул соединений» в приложении А и «Управление отказами с помощью `Hystrix`» главы 8. Создавать диспетчеры из тщательно настроенных объектов `Executor` рекомендуется в высоконагруженных системах. Но поскольку `RxJava` никак не контролирует независимо созданные классом `Executor` потоки, то и не может закрепить их (т. е. заставить задачу всегда выполняться в одном и том же потоке для улучшения локальности кэша). Такой диспетчер лишь гарантирует, что один экземпляр `Scheduler.Worker` (см. раздел «Обзор деталей реализации диспетчера» ниже) обрабатывает события последовательно.

```
Schedulers.immediate()
```

`Schedulers.immediate()` — специальный диспетчер, который исполняет задачу в потоке клиента, причем не асинхронно, а с блокированием. Применять его имеет смысл, только если какая-то часть API требует наличия диспетчера, а вы вполне удовлетворены поведением `Observable` по умолчанию, не предполагающим никакой многопоточности. На самом деле, подписка на `Observable` с применением диспетчера `immediate()` обычно дает тот же эффект, что и подписка вообще без диспетчера. В общем случае держитесь от него подальше, поскольку он блокирует вызывающий поток и полезен только в редких ситуациях.

```
Schedulers.trampoline()
```

Диспетчер `trampoline()` очень похож на `immediate()`, т. к. тоже исполняет задачи в том же потоке с блокированием. Но в отличие от `immediate()`

очередная задача выполняется, когда все предыдущие завершились. Диспетчер `immediate()` вызывает задачу сразу, а `trampoline()` ждет завершения текущей задачи. Трамплин (Trampoline) – это паттерн функционального программирования, позволяющий реализовать рекурсию без бесконечного роста стека вызовов. Лучше объяснить это на примере, и начнем мы с `immediate()`. Кстати, обратите внимание, что мы не взаимодействуем с экземпляром `Scheduler` напрямую, а сначала создаем `Worker`. Почему так делается, мы скоро узнаем в разделе «Обзор деталей реализации диспетчера».

```
Scheduler scheduler = Schedulers.immediate();
Scheduler.Worker worker = scheduler.createWorker();
```

```
log("Начало Main");
worker.schedule(() -> {
    log(" Начало Outer");
    sleepOneSecond();
    worker.schedule(() -> {
        log(" Начало Inner");
        sleepOneSecond();
        log(" Конец Inner");
    });
    log(" Конец Outer");
});
log("Конец Main");
worker.unsubscribe();
```

Результат вполне ожидаемый; можно было бы заменить `schedule()` простым вызовом метода:

```
1044 | main | Начало Main
1094 | main | Начало Outer
2097 | main | Начало Inner
3097 | main | Конец Inner
3100 | main | Конец Outer
3100 | main | Конец Main
```

Внутри блока `Outer` мы планируем методом `schedule()` блок `Inner`, который вызывается немедленно, прерывая задачу `Outer`. Когда `Inner` заканчивается, управление возвращается блоку `Outer`. Это не более чем витиеватый способ блокирующего вызова задачи – только косвенно, с помощью диспетчера `immediate()`. Но что произойдет, если заменить `Schedulers.immediate()` на `Schedulers.trampoline()`? Вывод изменится:

```
1030 | main | Начало Main
1096 | main | Начало Outer
2101 | main | Конец Outer
2101 | main | Начало Inner
3101 | main | Конец Inner
3101 | main | Конец Main
```


Видите – блок Outer закончился еще до того, как Inner начался? Это объясняется тем, что задача Inner была поставлена в очередь внутри диспетчера `trampoline()`, который занят выполнением задачи Outer. Когда Outer закончилась, началась задача, стоящая первой в очереди (Inner). Можно сделать еще шаг, чтобы лучше была видна разница:

```
log("Начало Main");
worker.schedule(() -> {
    log(" Начало Outer");
    sleepOneSecond();
    worker.schedule(() -> {
        log(" Начало Middle");
        sleepOneSecond();
        worker.schedule(() -> {
            log(" Начало Inner");
            sleepOneSecond();
            log(" Конец Inner");
        });
        log(" Конец Middle");
    });
    log(" Конец Outer");
});
log("Конец Main");
```

Объект Worker от диспетчера `immediate()` выводит следующее:

```
1029 | main | Начало Main
1091 | main | Начало Outer
2093 | main | Начало Middle
3095 | main | Начало Inner
4096 | main | Конец Inner
4099 | main | Конец Middle
4099 | main | Конец Outer
4099 | main | Конец Main
```

А вот что выводит Worker от `trampoline()`:

```
1041 | main | Начало Main
1095 | main | Начало Outer
2099 | main | Конец Outer
2099 | main | Начало Middle
3101 | main | Конец Middle
3101 | main | Начало Inner
4102 | main | Конец Inner
4102 | main | Конец Main
```

`Schedulers.test()`

Этот диспетчер применяется только для тестирования, в производственном коде вы его никогда не встретите. Его основное достоинство – способность произвольно переводить вперед часы, моделируя ход времени. `TestScheduler` очень подробно описан в разделе «Диспетчеры в автономных тестах» главы 7. Сами по себе диспетчеры не очень интересны. Если вы

хотите узнать, как они устроены внутри и как написать собственный, прочитайте следующий раздел.

Обзор деталей реализации диспетчера



Это факультативный раздел. Если вас не интересуют детали реализации, можете смело переходить к разделу «Декларативная подписка с помощью `subscribeOn()`».

Класс `Scheduler` не только разрывает связь между задачами и их выполнением (поскольку обычно исполняет их в другом потоке), но и абстрагирует часы, как мы узнаем в разделе «Виртуальное время» главы 7. API класса `Scheduler` несколько проще, чем, например, класса `ScheduledExecutorService`:

```
abstract class Scheduler {
    abstract Worker createWorker();
    long now();

    abstract static class Worker implements Subscription {

        abstract Subscription schedule(Action0 action);

        abstract Subscription schedule(Action0 action,
            long delayTime, TimeUnit unit);

        long now();
    }
}
```

Когда RxJava хочет запланировать задачу (обычно, но не обязательно фоновую), она должна сначала запросить экземпляр `Worker`. Именно `Worker` разрешает планирование задачи немедленно или на какой-то момент в будущем. Как `Scheduler`, так и `Worker` имеют переопределяемый источник времени (метод `now()`), с помощью которого указывается, когда должна быть запущена задача. Упрощенно можно представлять себе, что `Scheduler` — что-то вроде пула потоков, а `Worker` — поток из этого пула.

Разделение между `Scheduler` и `Worker` необходимо для упрощения реализации некоторых требований контракта Rx, а именно того, что метод подписчика должен вызываться последовательно, а не конкурентно. Контракт `Worker` именно это и гарантирует: две задачи, запланированные одним и тем же объектом `Worker`, никогда не будут выполняться одновременно. Но независимые `Worker`'ы, полученные от одного и того же `Scheduler` вполне могут исполнять задачи одновременно.

Вместо того чтобы методично изучать API, проанализируем исходный код одного реального диспетчера, а именно `HandlerScheduler` из проекта RxAndroid (<https://github.com/ReactiveX/RxAndroid>). Этот диспетчер просто исполняет все запланированные задачи в потоке пользовательского интерфейса Android. Об-

новление пользовательского интерфейса разрешено только из этого потока (подробности см. в разделе «Разработка для Android с применением RxJava» главы 8). Это аналог потока диспетчеризации событий (Event Dispatch Thread – EDT) в Swing (<https://docs.oracle.com/javase/tutorial/uiswing/concurrency/dispatch.html>), в котором должно выполняться большинство обновлений окон и компонентов. Не удивительно, что существует проект RxSwing (<https://github.com/ReactiveX/RxSwing>) на эту тему.

Фрагмент ниже – сокращенный и неполный класс из проекта RxAndroid, приведенный исключительно в учебных целях:

```
package rx.android.schedulers;

import android.os.Handler;
import android.os.Looper;
import rx.Scheduler;
import rx.Subscription;
import rx.functions.Action0;
import rx.internal.schedulers.ScheduledAction;
import rx.subscriptions.Subscriptions;

import java.util.concurrent.TimeUnit;

public final class SimplifiedHandlerScheduler extends Scheduler {

    @Override
    public Worker createWorker() {
        return new HandlerWorker();
    }

    static class HandlerWorker extends Worker {

        private final Handler handler = new Handler(Looper.getMainLooper());
        @Override

        public void unsubscribe() {
            // Реализация будет показана ниже...
        }

        @Override
        public boolean isUnsubscribed() {
            // Реализация будет показана ниже...
            return false;
        }

        @Override
        public Subscription schedule(final Action0 action) {
            return schedule(action, 0, TimeUnit.MILLISECONDS);
        }

        @Override
        public Subscription schedule(
            Action0 action, long delayTime, TimeUnit unit) {
```

```

ScheduledAction scheduledAction = new ScheduledAction(action);
handler.postDelayed(scheduledAction, unit.toMillis(delayTime));

scheduledAction.add(Subscriptions.create(() ->
    handler.removeCallbacks(scheduledAction)));

return scheduledAction;
    }
}
}

```

Детали Android API нам сейчас не важны. Существенно, что всякий раз как мы планируем что-то в контексте `HandlerWorker`, блок кода передается специальному методу `postDelayed()`, который выполняется в выделенном потоке Android. Существует всего один такой поток, поэтому сериализуются события не только от одного `Worker`, но и от всех таких объектов.

Прежде чем передавать подлежащее выполнению действие `action`, мы оборачиваем его объектом класса `ScheduledAction`, который реализует интерфейсы `Runnable` и `Subscription`. RxJava демонстрирует ленивое поведение всюду, где возможно, в том числе и при планировании задач. Если мы по какой-то причине решим, что действие `action` вообще не стоит выполнять (это может случиться, если действие запланировано на будущее, а не выполняется сразу), то нужно будет просто вызвать метод `unsubscribe()` объекта `Subscription`, полученного от `schedule()`. Надлежащим образом обработать отписку (по крайней мере, попытаться) – долг `Worker`.

Клиент может также решить вообще отписаться от `Worker`, вызвав его метод `unsubscribe()`. При этом должны быть отписаны все стоящие в очереди задачи, а сам объект `Worker` освобожден, чтобы занятый им поток можно было использовать повторно. Во фрагменте ниже класс `SimplifiedHandlerScheduler` продолжен – добавлена логика отписки в `Worker` (показаны только модифицированные методы).

```

private CompositeSubscription compositeSubscription =
    new CompositeSubscription();

@Override
public void unsubscribe() {
    compositeSubscription.unsubscribe();
}

@Override
public boolean isUnsubscribed() {
    return compositeSubscription.isUnsubscribed();
}

@Override
public Subscription schedule(Action0 action, long delayTime, TimeUnit unit) {
    if (compositeSubscription.isUnsubscribed()) {
        return Subscriptions.unsubscribed();
    }

    final ScheduledAction scheduledAction = new ScheduledAction(action);

```

```
scheduledAction.addParent(compositeSubscription);
compositeSubscription.add(scheduledAction);

handler.postDelayed(scheduledAction, unit.toMillis(delayTime));

scheduledAction.add(Subscriptions.create(() ->
    handler.removeCallbacks(scheduledAction)));

return scheduledAction;
}
```

В разделе «Управление прослушивателями с помощью типов Subscription и Subscriber<T>» главы 2 мы говорили об интерфейсе Subscription, но в детали реализации не вдавались. Класс CompositeSubscription – одна из многих имеющихся реализаций, он представляет собой контейнер дочерних объектов Subscription (паттерн *Компоновщик*). Отписка от CompositeSubscription подразумевает отписку от всех потомков. Потомков CompositeSubscription можно также добавлять и удалять.

В нашем пользовательском диспетчере класс CompositeSubscription используется для отслеживания всех экземпляров Subscription, появившихся в результате предыдущих вызовов schedule() (см. compositeSubscription.add(scheduledAction)). С другой стороны, дочернее действие ScheduledAction должно знать о своем родителе (см. addParent()), чтобы оно могло удалить себя по завершении или в случае отмены. Иначе worker сохранял бы уже неактуальные дочерние подписки вечно. Когда клиент решает, что ему больше не нужен экземпляр HandlerWorker, он отписывается от него. Отписка распространяется на все еще существующие дочерние подписки (если таковые остались).

Это было очень краткое введение в диспетчеры в RxJava. Детали их внутреннего устройства не так уж важны для повседневной работы; на самом-то деле, они и проектировались, так чтобы сделать поведение RxJava более интуитивно понятным и предсказуемым. А теперь посмотрим, как диспетчеры решают различные проблемы конкурентности в Rx.

Декларативная подписка с помощью subscribeOn()

В разделе «Подробнее о методе Observable.create()» главы 2 мы видели, что метод subscribe() по умолчанию пользуется потоком клиента. Чтобы освежить память, ниже приводится простейшая реализация, в которой нет никакой многопоточности:

```
Observable<String> simple() {
    return Observable.create(subscriber -> {
        log("Подписан");
        subscriber.onNext("A");
        subscriber.onNext("B");
        subscriber.onCompleted();
    });
}
```

```
}

//...

log("Начало");
final Observable<String> obs = simple();
log("Создан");
final Observable<String> obs2 = obs
    .map(x -> x)
    .filter(x -> true);
log("Преобразован");
obs2.subscribe(
    x -> log("Получено " + x),
    Throwable::printStackTrace,
    () -> log("Завершен")
);
log("Выход");
```

Заметьте, в каком порядке выводятся сообщения, и обратите особое внимание на то, в каком потоке они печатаются:

```
33 | main | Начало
120 | main | Создан
128 | main | Преобразован
133 | main | Подписан
133 | main | Получено А
133 | main | Получено В
133 | main | Завершен
134 | main | Выход
```

Порядок сообщений абсолютно предсказуем. Во-первых, все строки кода выполнены в потоке `main`, не существует ни пулов потоков, ни асинхронного порождения событий. Во-вторых, порядок выполнения с первого взгляда не вполне ясен.

В самом начале работы программа печатает `Начало`, это понятно. После создания экземпляра `Observable<String>` мы видим сообщение `Создан`. Отметим, что сообщение `Подписан` появляется позже, когда мы действительно подписались. Не будь вызова `subscribe()`, блок кода внутри `Observable.create()` так и не был бы выполнен. Далее, хотя операторы `map()` и `filter()` не имеют видимых побочных эффектов, сообщение `преобразован` печатается раньше, чем `Подписан`.

Затем мы получаем все порожденные события и уведомление о завершении. В самом конце печатается сообщение `выход`. Интересное наблюдение – `subscribe()` задумывался для регистрации обратного вызова при асинхронном появлении событий. Этого предположения вы и должны придерживаться по умолчанию. Однако в данном случае многопоточности не наблюдается, и `subscribe()` блокирует выполнение. Как же так?

Существует неразрывная, но скрытая связь между `subscribe()` и `create()`. Когда мы подписываемся на объект `Observable` методом `subscribe()`, вызывается его метод обратного вызова `onSubscribe()` (обертывающий лямбда-выражение, переданное методу `create()`). Он получает ваш объект `Subscriber` в качестве аргумента. По умолчанию это происходит в том же потоке и является блокирующим действием,

поэтому все происходящее внутри `create()` блокирует `subscribe()`. Если `create()` на несколько секунд заснет, то и `subscribe()` будет приостановлен. Кроме того, если между `Observable.create()` и подписчиком (лямбда-выражением, выступающим в роли обратного вызова) есть операторы, то все они вызываются в том же потоке, из которого вызван `subscribe()`. По умолчанию RxJava не вставляет никаких средств организации конкурентности между `Observable` и `Subscriber`. Причина в том, что объекты `Observable`, как правило, поддерживаются другими механизмами конкурентности, например циклами обработки событий или пользовательскими потоками, поэтому Rx оставляет вам полную свободу действий и не навязывает никаких соглашений.

Это наблюдение готовит почву для оператора `subscribeOn()`. Вставив `subscribeOn()` в любом месте между исходным `Observable` и `subscribe()`, мы декларативно объявляем диспетчер, в контексте которого будет вызываться метод `onSubscribe`. И какие бы действия ни производились внутри `create()`, эта работа препоручается независимому `Scheduler`, а наш вызов `subscribe()` не блокируется:

```
log("Начало");
final Observable<String> obs = simple();
log("Создан");
obs
    .subscribeOn(schedulerA)
    .subscribe(
        x -> log("Получено " + x),
        Throwable::printStackTrace,
        () -> log("Завершен")
    );
log("Выход");
```

```
35 | main | Начало
112 | main | Создан
123 | main | Выход
123 | Sched-A-0 | Подписан
124 | Sched-A-0 | Получено A
124 | Sched-A-0 | Полученот B
124 | Sched-A-0 | Завершен
```

Видите: поток `main` закончил работу еще до того, как `Observable` начал порождать значения? Теперь порядок вывода сообщений не предсказуем, потому что два потока работают одновременно: `main`, который подписался и хочет выйти, и `Sched-A-0`, который начинает порождать события, как только кто-то на него подпишется. Диспетчер `schedulerA`, равно как и поток `Sched-A-0`, создается в следующем коде, написанном для иллюстрации:

```
import static java.util.concurrent.Executors.newFixedThreadPool;

ExecutorService poolA = newFixedThreadPool(10, threadFactory("Sched-A-%d"));
Scheduler schedulerA = Schedulers.from(poolA);

ExecutorService poolB = newFixedThreadPool(10, threadFactory("Sched-B-%d"));
```

```
Scheduler schedulerB = Schedulers.from(poolB);

ExecutorService poolC = newFixedThreadPool(10, threadFactory("Sched-C-%d"));
Scheduler schedulerC = Schedulers.from(poolC);

private ThreadFactory threadFactory(String pattern) {
    return new ThreadFactoryBuilder()
        .setNameFormat(pattern)
        .build();
}
```

Эти диспетчеры будут использоваться во всех примерах, но запомнить их довольно просто: три независимых диспетчера, каждый из которых управляет 10 потоками, получаемыми от экземпляра `ExecutorService`. Чтобы было проще разобратся в выводе, у каждого пула потоков своя схема именования.

Прежде чем мы приступим к делу, вы должны осознать, что в зрелых приложениях, написанных с использованием Rx, метод `subscribeOn()` применяется очень редко. Обычно объекты `Observable` поступают из естественно асинхронных источников (как, например, в RxNetty, см. раздел «Неблокирующий HTTP-сервер на основе Netty и RxNetty» главы 5) или применяют собственную диспетчеризацию (как в Hystrix, см. раздел «Управление отказами с помощью Hystrix» главы 8). Прибегать к `subscribeOn()` следует только в особых случаях, когда известно, что `Observable` синхронный (метод `create()` блокирующий). Но все же `subscribeOn()` гораздо лучше, чем самописная многопоточность внутри `create()`:

```
// Не делайте так
Observable<String> obs = Observable.create(subscriber -> {
    log("Подписан");
    Runnable code = () -> {
        subscriber.onNext("A");
        subscriber.onNext("B");
        subscriber.onCompleted();
    };
    new Thread(code, "Async").start();
});
```

В этом коде смешаны две концепции: порождение событий и выбор стратегии конкурентности. `Observable` должен отвечать только за логику порождения, а обоснованные решения насчет конкурентности может принять только клиентский код. Напомню, что тип `Observable` ленивый, но также неизменяемый в том смысле, что `subscribeOn()` влияет только на расположенных после него подписчиков. Если какой-нибудь другой клиент подпишется на тот же объект `Observable`, не поместив предварительно `subscribeOn()`, то по умолчанию никакой конкурентности не будет.

Помните, что тема этой главы – постепенное внедрение RxJava в существующие приложения. В таких обстоятельствах оператор `subscribeOn()` весьма полезен, но после того как вы освоили реактивные расширения и начали применять их повсеместно, `subscribeOn()` теряет значение. В системах, реактивных на всех уровнях,

к каковым относится Netflix, оператор `subscribeOn()` почти никогда не используется, а все объекты `Observable` тем не менее асинхронны. Большинство `Observable` поступает из асинхронных источников и по умолчанию считаются асинхронными. Поэтому у `subscribeOn()` крайне ограниченное применение, сводящееся, в основном, к адаптации существующих API или библиотек. В главе 5 мы будем писать по-настоящему асинхронные приложения, вообще обходясь без явных вызовов `subscribeOn()` и диспетчеров.

Конкурентность и поведение `subscribeOn()`

В работе `subscribeOn()` есть несколько нюансов. Во-первых, у любознательного читателя может возникнуть вопрос, что произойдет, если между `Observable` и `subscribe()` вставить два вызова `subscribeOn()`. Очень просто: *выигрывает* тот `subscribeOn()`, который ближе к исходному `Observable`. У этого факта есть важные практические следствия. Если внутри спроектированного вами API употребляется `subscribeOn()`, то клиентский код не сможет переопределить заданный вами `Scheduler`. Такое решение следует принимать осознанно: иногда проектировщику API виднее, какой диспетчер подходит. С другой стороны, всегда неплохо предоставлять перегруженный вариант API, позволяющий переопределить выбранный диспетчер.

Изучим, как ведет себя оператор `subscribeOn()`:

```
log("Начало");
Observable<String> obs = simple();
log("Создан");
obs
    .subscribeOn(schedulerA)
    // много других операторов
    .subscribeOn(schedulerB)
    .subscribe(
        x -> log("Получено " + x),
        Throwable::printStackTrace,
        () -> log("Завершен")
    );
log("Выход");
```

В распечатке видны только потоки диспетчера `schedulerA`:

```
17 | main | Начало
73 | main | Создан
83 | main | Выход
84 | Sched-A-0 | Подписан
84 | Sched-A-0 | Получено A
84 | Sched-A-0 | Получено B
84 | Sched-A-0 | Завершен
```

Интересно, что подписка на `schedulerB` не совсем игнорируется в пользу `schedulerA`. Диспетчер `schedulerB` все-таки используется короткое время, но только планирует новое действие в контексте `schedulerA`, который и выполняет всю ра-

боту. Таким образом, наличие нескольких `subscribeOn()` не просто игнорируется, а сопряжено еще и с небольшими накладными расходами.

Кстати об операторах. Мы сказали, что метод `create()`, вызываемый при появлении нового подписчика, выполняется в контексте предоставленного диспетчера (при наличии такового). Но в каком потоке исполняются все преобразования, расположенные между `create()` и `subscribe()`? Мы уже знаем, что когда операторы исполняются по умолчанию в одном и том же потоке (диспетчере), конкурентность не возникает:

```
log("Начало");
final Observable<String> obs = simple();
log("Создан");
obs
    .doOnNext(this::log)
    .map(x -> x + '1')
    .doOnNext(this::log)
    .map(x -> x + '2')
    .subscribeOn(schedulerA)
    .doOnNext(this::log)
    .subscribe(
        x -> log("Получено " + x),
        Throwable::printStackTrace,
        () -> log("Завершен")
    );
log("Выход");
```

Мы включили в конвейер несколько операторов вперемежку с `doOnNext()`, чтобы видеть, какой поток выполняется в данной точке. Напомним, что положение `subscribeOn()` несущественно — он может находиться сразу после `Observable` или непосредственно перед `subscribe()`. Результат не содержит никаких сюрпризов:

```
20 | main | Начало
104 | main | Создан
123 | main | Выход
124 | Sched-A-0 | Подписан
124 | Sched-A-0 | A
124 | Sched-A-0 | A1
124 | Sched-A-0 | A12
124 | Sched-A-0 | Получено A12
124 | Sched-A-0 | B
124 | Sched-A-0 | B1
124 | Sched-A-0 | B12
125 | Sched-A-0 | Получено B12
```

Проследите за тем, как метод `create()` вызывается и порождает события `A` и `B`. Эти события последовательно перемещаются в потоке выполнения диспетчера и в итоге доходят до подписчика. Многие программисты, начинающие изучать RxJava, думают, что если использовать диспетчер с большим количеством потоков, то он автоматически распараллелит обработку событий и каким-то образом соберет в конце все результаты. Это не так. RxJava создает один экземпляр `Worker`

(см. раздел «Обзор деталей реализации диспетчера» выше) на весь конвейер, прежде всего, чтобы гарантировать последовательную обработку событий.

Это означает, что если какой-нибудь оператор работает особенно медленно – например, `map()` читает данные с диска для преобразования проходящих событий – то операция все равно будет производиться в том же потоке. Один-единственный плохо написанный оператор может замедлить весь конвейер. Это антипаттерн в RxJava – операторы должны быть неблокирующими, быстрыми и по возможности чистыми.

И снова на помощь приходит оператор `flatMap()`. Вместо того чтобы блокироваться внутри `map()`, мы можем вызвать `flatMap()` и асинхронно собрать все результаты. Таким образом, `flatMap()` и `merge()` – это именно те операторы, которые нужны для достижения истинного параллелизма. Но даже с применением `flatMap()` это не очевидно. Представьте себе продуктовый магазин (назовем его «RxGroceries»), который предоставляет API для покупки товаров:

```
class RxGroceries {

    Observable<BigDecimal> purchase(String productName, int quantity) {
        return Observable.fromCallable(() ->
            doPurchase(productName, quantity));
    }

    BigDecimal doPurchase(String productName, int quantity) {
        log("Покупается " + quantity + " " + productName);
        // здесь должна быть бизнес-логика
        log("Куплено " + quantity + " " + productName);
        return priceForProduct;
    }
}
```

Реализация `doPurchase()` здесь не имеет значения, просто будем считать, что она занимает некоторое время и ресурсы. В качестве имитации бизнес-логики мы добавили искусственную задержку на одну секунду или чуть больше, если `quantity` велико. Блокирующие объекты `Observable` типа того, что возвращает метод `purchase()`, – в реальном приложении редкость, но в учебных целях оставим так. Мы хотели бы максимально распараллелить покупку нескольких товаров, а в конце вычислить их полную стоимость.

Первая попытка оказывается тщетной:

```
Observable<BigDecimal> totalPrice = Observable
    .just("хлеб", "масло", "молоко", "помидоры", "сыр")
    .subscribeOn(schedulerA) // НЕПРАВИЛЬНО!!!
    .map(prod -> rxGroceries.doPurchase(prod, 1))
    .reduce(BigDecimal::add)
    .single();
```

Результат правильный, это `Observable`, содержащий всего одно значение: сумму цен, вычисленную с помощью `reduce()`. Для каждого товара мы вызываем метод `doPurchase()`, задав параметр `quantity` равным 1. Но несмотря на то, что диспетчер

`schedulerA` управляет пулом из 10 потоков, программа выполняется строго последовательно:

```
144 | Sched-A-0 | Покупается 1 хлеб
1144 | Sched-A-0 | Куплено 1 хлеб
1146 | Sched-A-0 | Покупается 1 масло
2146 | Sched-A-0 | Куплено 1 масло
2146 | Sched-A-0 | Покупается 1 молоко
3147 | Sched-A-0 | Куплено 1 молоко
3147 | Sched-A-0 | Покупается 1 помидоры
4147 | Sched-A-0 | Куплено 1 помидоры
4147 | Sched-A-0 | Покупается 1 сыр
5148 | Sched-A-0 | Куплено 1 сыр
```

Обратите внимание, что обработка одного продукта блокирует обработку следующих за ним. Обработка масла начинается сразу, как только закончилась обработка хлеба, но не раньше. Странно, но даже замена `map()` на `flatMap()` не помогает, результат остается точно таким же:

```
Observable
    .just("хлеб", "масло", "молоко", "помидоры", "сыр")
    .subscribeOn(schedulerA)
    .flatMap(prod -> rxGroceries.purchase(prod, 1))
    .reduce(BigDecimal::add)
    .single();
```

Код не распараллеливается, потому что существует ровно один поток событий, который должен обрабатываться последовательно – так и задумано. В противном случае подписчика пришлось бы защищать от одновременного получения уведомлений (`onNext()`, `onComplete()` и т. д.), так что этот компромисс представляется оправданным. К счастью, мы очень близко подошли к идиоматичному решению. Главный `Observable`, порождающий товары, распараллелить нельзя. Но для каждого товара мы создаем новый независимый `Observable`, возвращенный из метода `purchase()`. Поскольку все они независимы, никто не мешает планировать их конкурентное выполнение:

```
Observable<BigDecimal> totalPrice = Observable
    .just("хлеб", "масло", "молоко", "помидоры", "сыр")
    .flatMap(prod ->
        rxGroceries
            .purchase(prod, 1)
            .subscribeOn(schedulerA))
    .reduce(BigDecimal::add)
    .single();
```

Найдите, где притаился `subscribeOn()`? Главный `Observable` на самом деле ничего не делает, поэтому специальный пул потоков не нужен. Однако каждый подпоток, созданный внутри `flatMap()`, наделяется диспетчером `schedulerA`. При каждом вызове `subscribeOn()` диспетчер получает возможность вернуть новый объект `Worker`, а, значит, запустить отдельный поток (я немного упрощаю):

```
113 | Sched-A-1 | Покупается 1 масло
114 | Sched-A-0 | Покупается 1 хлеб
125 | Sched-A-2 | Покупается 1 молоко
125 | Sched-A-3 | Покупается 1 помидоры
126 | Sched-A-4 | Покупается 1 сыр
1126 | Sched-A-2 | Куплено 1 молоко
1126 | Sched-A-0 | Куплено 1 хлеб
1126 | Sched-A-1 | Куплено 1 масло
1128 | Sched-A-3 | Куплено 1 помидоры
1128 | Sched-A-4 | Куплено 1 сыр
```

Наконец-то мы добились истинной конкурентности. Теперь все операции покупки начинаются в одно и то же время, и все в конечном итоге заканчиваются. Оператор `flatMap()` тщательно спроектирован и реализован, так что собирает события ото всех независимых потоков и последовательно передает их дальше. Правда, как мы помним из раздела «Порядок событий после `flatMap()`» главы 3, больше нельзя полагаться на порядок событий – они ни начинаются, ни завершаются в том порядке, в каком порождались источником (исходная последовательность начиналась с хлеба). Тем не менее, до оператора `reduce()` события доходят последовательно и ведут себя корректно.

К этому моменту вы должны бы уже постепенно отходить от классической модели на основе класса `Thread` и понимать, как работают диспетчеры `Scheduler`. Но если все еще испытываете затруднения, то вот вам простая аналогия.

- `Observable` без `Scheduler` работает как однопоточная программа с блокирующими вызовами методов, которые передают друг другу данные;
- `Observable` с одним оператором `subscribeOn()` можно уподобить запуску большой задачи в фоновом потоке. В этом потоке программа работает последовательно, но, по крайней мере, в фоновом режиме.
- `Observable` с оператором `flatMap()`, в котором у каждого внутреннего `Observable` есть свой оператор `subscribeOn()`, работает, как класс `ForkJoinPool` из пакета `java.util.concurrent`, где каждый подпоток является *разветвлением* (`fork`) выполнения, а `flatMap()` обеспечивает безопасное *объединение* (`join`).

Разумеется, все приведенные выше советы относятся только к блокирующим объектам `Observable`, которые редко встречаются в реальных приложениях. Если ваши `Observable` уже асинхронные, то для достижения конкурентности нужно просто понимать, как их скомбинировать и когда произвести подписку. Например, применение `merge()` к двум потокам означает одновременную подписку на оба, тогда как оператор `concat()` ждет, когда закончится первый поток и только потом подписывается на второй.

Создание пакета запросов с помощью `groupBy()`

Вы обратили внимание, что метод `RxGroceries.purchase()` принимает название товара `productName` и количество `quantity`, хотя количество всегда было равно 1? А если бы некоторые продукты повторялись в списке несколько раз – как при-

знак повышенного спроса? Первое, что приходит на ум, – просто послать один и тот же запрос, например на яйца, несколько раз, всегда указывая количество 1. По счастью, можно декларативно собрать такие запросы в пакет с помощью оператора `groupBy()` – и это решение совместимо с декларативной конкурентностью:

```
import org.apache.commons.lang3.tuple.Pair;

Observable<BigDecimal> totalPrice = Observable
    .just("хлеб", "масло", "яйца", "молоко", "помидоры",
        "сыр", "помидоры", "яйца", "яйца")
    .groupBy(prod -> prod)
    .flatMap(grouped -> grouped
        .count()
        .map(quantity -> {
            String productName = grouped.getKey();
            return Pair.of(productName, quantity);
        }))
    .flatMap(order -> store
        .purchase(order.getKey(), order.getValue())
        .subscribeOn(schedulerA))
    .reduce(BigDecimal::add)
    .single();
```

Код довольно сложный, поэтому прежде чем показывать результат, давайте разберемся в нем. Первым делом мы группируем товары просто по названию, отсюда и тождественная функция `prod -> prod`. Результат имеет такой непрезентабельный тип: `Observable<GroupedObservable<String, String>>`. Но ничего плохого в нем нет. Далее `flatMap()` поочередно получает на входе объекты типа `GroupedObservable<String, String>`, представляющие все товары с одинаковым названием. Например, среди них будет объект `Observable["яйца", "яйца", "яйца"]` с ключом "яйца". Если бы в `groupBy()` была передана другая функция формирования ключа, например `prod.length()`, то та же последовательность имела бы ключ 4.

Теперь нам нужно построить внутри `flatMap()` объект `Observable` типа `Pair<String, Integer>`, представляющий уникальный товар и его количество. Как `count()`, так и `map()` возвращают `Observable`, поэтому все отлично стыкуется. Второй оператор `flatMap()` получает параметр `order` типа `Pair<String, Integer>` и оформляет покупку, но теперь количество может быть больше 1. Результат выглядит прекрасно; обратите внимание, что чем больше заказано единиц товара, тем дольше оформляется заказ, но все равно гораздо быстрее, чем в случае нескольких запросов одного и того же:

```
164 | Sched-A-0 | Покупается 1 хлеб
165 | Sched-A-1 | Покупается 1 масло
166 | Sched-A-2 | Покупается 3 яйца
166 | Sched-A-3 | Покупается 1 молоко
166 | Sched-A-4 | Покупается 2 помидоры
166 | Sched-A-5 | Покупается 1 сыр
1151 | Sched-A-0 | Куплено 1 хлеб
1178 | Sched-A-1 | Куплено 1 масло
1180 | Sched-A-5 | Куплено 1 сыр
```

```
1183 | Sched-A-3 | Куплено 1 молоко
1253 | Sched-A-4 | Куплено 2 помидоры
1354 | Sched-A-2 | Куплено 3 яйца
```

Если вы полагаете, что в вашей системе такая сборка пакетов может принести пользу, то почитайте еще раздел «Пакетирование и объединение команд» главы 8.

Декларативная конкурентность с помощью `observeOn()`

Хотите верить, хотите нет, но конкурентность в RxJava можно описать двумя операторами: вышеупомянутым `subscribeOn()`, а также `observeOn()`. Они, на первый взгляд, очень похожи, и это вызывает недоумение у начинающих, но на самом деле у них есть четкая и вполне разумная семантика.

Оператор `subscribeOn()` позволяет указать, какой диспетчер будет использован для вызова `OnSubscribe` (лямбда-выражения внутри `create()`). Поэтому любой код, переданный `create()`, исполняется в другом потоке – например, чтобы не блокировать главный поток. Напротив, `observeOn()` управляет тем, какой диспетчер будет использован для вызова подписчиков, расположенных после `observeOn()`. Например, вызов `create()` происходит в контексте диспетчера `io()` (благодаря `subscribeOn(io())`), чтобы не блокировать пользовательский интерфейс (UI). Но обновление виджетов обязано происходить в потоке пользовательского интерфейса (такое ограничение накладывают и Swing, и Android), поэтому мы вставляем оператор `observeOn()`, указав, например, диспетчер `AndroidSchedulers.mainThread()` перед операторами или подписчиками, изменяющими UI. Тем самым мы можем использовать один диспетчер для `create()` и всех операторов, предшествующих первому `observeOn()`, и другой диспетчер (или несколько разных диспетчеров) для применения преобразований. Лучше всего объяснить это на примере:

```
log("Начало");
final Observable<String> obs = simple();
log("Создан");
obs
    .doOnNext(x -> log("Найдено 1: " + x))
    .observeOn(schedulerA)
    .doOnNext(x -> log("Найдено 2: " + x))
    .subscribe(
        x -> log("Получено 1: " + x),
        Throwable::printStackTrace,
        () -> log("Завершен")
    );
log("Выход");
```

`observeOn()` встречается в конвейере и, в отличие от `subscribeOn()`, его положение играет важную роль. Какой бы диспетчер ни применялся к операторам, предшествующим `observeOn()` (если вообще какой-то задан), для всех операторов после него, используется указанный в нем диспетчер. В данном случае оператор `subscribeOn()` отсутствует, поэтому действует режим по умолчанию (без конкурентности).

```

23 | main | Начало
136 | main | Создан
163 | main | Подписан
163 | main | Найдено 1: A
163 | main | Найдено 1: B
163 | main | Выход
163 | Sched-A-0 | Найдено 2: A
164 | Sched-A-0 | Получено 1: A
164 | Sched-A-0 | Найдено 2: B
164 | Sched-A-0 | Получено 1: B
164 | Sched-A-0 | Завершен

```

Все операторы до `observeOn` выполняются в потоке клиента, это режим по умолчанию в RxJava. А операторы после `observeOn()` выполняются в контексте указанного диспетчера. Картина складывается еще более наглядная, когда в конвейере есть `subscribeOn()` и несколько `observeOn()`:

```

log("Начало");
final Observable<String> obs = simple();
log("Создан");
obs
    .doOnNext(x -> log("Найдено 1: " + x))
    .observeOn(schedulerB)
    .doOnNext(x -> log("Найдено 2: " + x))
    .observeOn(schedulerC)
    .doOnNext(x -> log("Найдено 3: " + x))
    .subscribeOn(schedulerA)
    .subscribe(
        x -> log("Получено 1: " + x),
        Throwable::printStackTrace,
        () -> log("Завершен")
    );
log("Выход");

```

Сможете предсказать результат? Напомню, все операторы после `observeOn()` выполняются в контексте указанного диспетчера, но только до тех пор, пока не встретится следующий `observeOn()`. Кроме того, `subscribeOn()` может располагаться в любом месте между `Observable` и `subscribe()`, но теперь он влияет только на операторы до первого `observeOn()`:

```

21 | main | Начало
98 | main | Создан
108 | main | Выход
129 | Sched-A-0 | Подписан
129 | Sched-A-0 | Найдено 1: A
129 | Sched-A-0 | Найдено 1: B
130 | Sched-B-0 | Найдено 2: A
130 | Sched-B-0 | Найдено 2: B
130 | Sched-C-0 | Найдено 3: A
130 | Sched-C-0 | Получено: A
130 | Sched-C-0 | Найдено 3: B
130 | Sched-C-0 | Получено: B
130 | Sched-C-0 | Завершен

```


Подписка происходит в контексте диспетчера `schedulerA`, потому что именно его мы передали оператору `subscribeOn()`. Оператор "Найдено 1" также выполнялся в контексте этого диспетчера, потому что он расположен до первого `observeOn()`. Затем ситуация становится более интересной. Оператор `observeOn()` переключается с текущего диспетчера на `schedulerB`, и "Найдено 2" печатается уже в новом контексте. Последний оператор `observeOn(schedulerC)` влияет как на печать "Найдено 3", так и на подписчика. Напомню, что подписчик работает в контексте последнего встретившегося диспетчера.

Операторы `subscribeOn()` и `observeOn()` прекрасно работают вместе, когда требуется разорвать физическую связь между производителем (`Observable.create()`) и потребителем (`Subscriber`). По умолчанию эта связь не разрывается, и RxJava просто пользуется одним и тем же потоком. Одного `subscribeOn()` недостаточно, потому что мы только сменим поток. Один `observeOn()` лучше, но тогда мы блокируем поток клиента в случае синхронного `Observable`. Поскольку операторы по большей части неблокирующие, а передаваемые им лямбда-выражения обычно короткие и не накладные, то в типичном случае в цепочке операторов присутствует всего по одному `subscribeOn()` и `observeOn()`. Оператор `subscribeOn()` можно поместить близко к исходному `Observable`, чтобы сделать код понятнее, а `observeOn()` — близко к `subscribe()`, чтобы лишь сам подписчик пользовался этим специальным диспетчером, а остальные операторы работали в контексте диспетчера, указанного в `subscribeOn()`.

Вот более сложная программа, в которой используются оба эти оператора:

```
log("Начало");
Observable<String> obs = Observable.create(subscriber -> {
    log("Подписан");
    subscriber.onNext("A");
    subscriber.onNext("B");
    subscriber.onNext("C");
    subscriber.onNext("D");
    subscriber.onCompleted();
});
log("Создан");
obs
    .subscribeOn(schedulerA)
    .flatMap(record -> store(record).subscribeOn(schedulerB))
    .observeOn(schedulerC)
    .subscribe(
        x -> log("Получено: " + x),
        Throwable::printStackTrace,
        () -> log("Завершен")
    );
log("Выход");
```

Здесь `store()` — простая вложенная операция:

```
Observable<UUID> store(String s) {
    return Observable.create(subscriber -> {
        log("Сохраняется " + s);
```

```
// какая-то долгая работа
subscriber.onNext(UUID.randomUUID());
subscriber.onCompleted();
});
}
```

Порождение событий происходит в контексте `schedulerA`, но каждое событие обрабатывается независимо с применением `schedulerB` для повышения уровня конкурентности. С этой техникой мы познакомились в разделе «Конкурентность и поведение `subscribeOn()`» выше. Подписка в самом конце происходит в контексте еще одного диспетчера, `schedulerC`. Уверен, вы теперь понимаете, какой диспетчер (поток) будет исполнять каждое действие, но на всякий случай приведу результат (пустые строки добавлены для большей понятности).

```
6 | main | Начало
93 | main | Создан
121 | main | Выход

122 | Sched-A-0 | Подписан
124 | Sched-B-0 | Сохраняется A
124 | Sched-B-1 | Сохраняется B
124 | Sched-B-2 | Сохраняется C
124 | Sched-B-3 | Сохраняется D

1136 | Sched-C-1 | Получено: 44b8b999-e687-485f-b17a-a11f6a4bb9ce
1136 | Sched-C-1 | Получено: 532ed720-eb35-4764-844e-690327ac4fe8
1136 | Sched-C-1 | Получено: 13ddf253-c720-48fa-b248-4737579a2c2a
1136 | Sched-C-1 | Получено: 0eced01d-3fa7-45ec-96fb-572ff1e33587
1137 | Sched-C-1 | Завершено
```

Оператор `observeOn()` особенно важен в приложениях с графическим пользовательским интерфейсом, где мы не хотим блокировать поток диспетчеризации событий UI. В Android и в Swing некоторые действия, в частности обновление UI, должны выполняться в определенном потоке. Но если делать в этом потоке слишком много, то интерфейс перестанет реагировать. В таких случаях мы располагаем `observeOn()` близко к `subscribe()`, чтобы код подписчика выполнялся в контексте конкретного диспетчера (например, в потоке UI). Но другие преобразования, даже не требующие много времени, следует выполнять вне потока UI. В серверных приложениях оператор `observeOn()` используется редко, потому что истинный источник конкурентности встроен в большинство объектов `Observable`. Это приводит к интересному наблюдению: RxJava контролирует конкурентность с помощью всего двух операторов (`subscribeOn()` и `observeOn()`), но чем больше вы используете реактивные расширения, тем реже встречаете их в производственном коде.

Другие применения диспетчеров

Есть много операторов, в которых по умолчанию используется тот или иной диспетчер. Если пользователь не предоставил никакого другого, то обычно применяется диспетчер `Schedulers.computation()` – в документации это всегда ясно указывается. Например, оператор `delay()` получает входящие события и отправ-

ляет их дальше после заданной задержки. Понятно, что он не может блокировать на это время исходный поток выполнения, поэтому должен пользоваться другим диспетчером:

```
Observable
    .just('A', 'B')
    .delay(1, SECONDS, schedulerA)
    .subscribe(this::log);
```

Если пользователь не указал диспетчер `schedulerA`, то все операторы после `delay()` будут пользоваться диспетчером `computation()`. В этом нет ничего дурного, однако, если ваш подписчик окажется заблокирован в ожидании завершения ввода-вывода, то он займет один экземпляр `worker` из разделяемого всеми глобального диспетчера `computation()`, что может негативно сказаться на системе в целом. Есть и другие важные операторы, поддерживающие задание диспетчера пользователем: `interval()`, `range()`, `timer()`, `repeat()`, `skip()`, `take()`, `timeout()` и еще несколько, с которыми нам только предстоит познакомиться. Если не предоставить такому оператору диспетчер, то будет использован диспетчер `computation()`, что в большинстве случаев безопасно.

Хорошо понимать работу диспетчеров необходимо для написания масштабируемого и безопасного кода с применением RxJava. Различие между `subscribeOn()` и `observeOn()` особенно важно в условиях высокой нагрузки, когда каждая задача должна исполняться точно в ожидаемое время. В полностью реактивных приложениях, где все длительные операции асинхронны, нужно очень мало потоков и, следовательно, диспетчеров. Но всегда находится какой-нибудь сторонний API или зависимость, требующие блокирующего кода.

И последнее, но оттого не менее важное – мы должны гарантировать, что диспетчеры, находящиеся дальше от исходного `Observable`, способны справиться с нагрузкой, генерируемой предшествующими им диспетчерами. Эта опасность будет детально разобрана в главе 6.

Резюме

В этой главе описано несколько паттернов, встречающихся в традиционных приложениях, которые можно заменить на RxJava. Надеюсь, теперь вы понимаете, что высокочастотная биржевая торговля или поток сообщений из социальной сети – не единственные сферы применения RxJava. На самом деле, почти любой API можно органично заменить объектом `Observable`. Даже если прямо сейчас вы не хотите или не видите необходимости в использовании всей мощи реактивных расширений, библиотека позволит постепенно развивать реализацию, не внося несомкстимых изменений. Кроме того, все плоды, предлагаемые RxJava – ленивость, декларативную конкурентность или асинхронное сцепление, – в конечном итоге пожинает клиент. И еще лучше то, что благодаря ненавязчивому преобразованию `Observable` в `BlockingObservable` традиционные клиенты могут потреблять ваш API, как им будет удобно, а вы всегда сможете предоставить простой мост.

Стоит освоить RxJava в достаточной мере, чтобы оценить преимущества ее применения даже в унаследованных системах. Несомненно, работать с реактивными объектами Observable сложнее, и на изучение требуется время. Но открывающиеся новые возможности невозможно переоценить. Что, если бы вы могли написать целое приложение от начала до конца на основе реактивных расширений? Проект с чистого листа, где у вас есть контроль над всеми API, интерфейсом и внешней системой? В главе 5 мы обсудим, как можно написать такое приложение и каковы последствия.



Глава 5.

Реактивность сверху донизу

«Всё в мире – поток» – так звучит часто цитируемая мудрость RxJava. В главе 4 мы узнали, как внедрить RxJava в некоторые места кода. Но довольно быстро вы поймете, что в истинно реактивных приложениях потоки используются не эпизодически, а повсеместно – сверху донизу. Это упрощает рассуждения, и все приложение получается единообразным. Неблокирующие приложения обычно демонстрируют отличную производительность и пропускную способность даже на сравнительно слабом оборудовании. Ограничив число потоков, мы сможем в полной мере задействовать процессор, не потребляя гигабайты памяти.

Один из факторов, ограничивающих масштабируемость Java-программ, – механизм ввода-вывода. Пакет `java.io` прекрасно спроектирован, в нем много небольших реализаций интерфейсов `Input/OutputStream` и `Reader/Writer`, которые декорируют и обертывают друг друга, постепенно расширяя функциональность. Но как бы мне ни было симпатично такое изящное разделение обязанностей, стандартные средства ввода-вывода в Java целиком и полностью блокирующие, т. е. каждый поток, желающий читать из сокета или файла или писать туда, должен неопределенно долго ждать результата. Хуже того, потоки, ожидающие завершения ввода-вывода из-за медленной сети или даже небыстро вращающегося диска, трудно прервать. Проблемой является не блокирование само по себе, ведь когда один поток блокирован, другие могут взаимодействовать с остальными открытыми сокетами. Но создание потоков и управление ими обходится дорого, а переключение между потоками занимает время. Java-приложения вполне способны поддерживать десятки тысяч одновременных подключений, если только правильно спроектированы. А эта задача намного упрощается, если объединить RxJava с некоторыми современными событийно-ориентированными библиотеками и каркасами.

Решение проблемы C10k

Проблемой C10k называются исследования, направленные на оптимизацию системы таким образом, чтобы она могла поддерживать 10 000 одновременных соединений на одном сервере потребительского класса. Даже сегодня решить эту инженерную задачу средствами традиционного инструментария Java отнюдь не

тривиально. Но есть много реактивных подходов, которые позволяют с легкостью решить проблему C10k, и благодаря RxJava воспользоваться ими совсем несложно. В этой главе мы рассмотрим ряд способов реализации, позволяющих увеличить степень масштабируемости на несколько порядков. Все они возвращаются во круг концепции реактивного программирования. Если вам повезло работать над совсем новым проектом, то почему бы не подумать о его реализации в реактивном стиле сверху донизу? Такое приложение никогда не будет синхронно ждать завершения вычисления или действия. Чтобы избежать блокирования, его архитектура должна быть полностью *событийно-ориентированной* и *асинхронной*. Мы изучим несколько примеров простого HTTP-сервера и посмотрим, как они ведут себя с точки зрения принятых проектных решений. Следует признать, что за повышение производительности и масштабируемости приходится расплачиваться сложностью. Но RxJava позволяет заметно уменьшить сложность, традиционно присущую таким подходам.

Классическая модель – один поток на каждое подключение – если и решает проблему C10k, то с большим трудом. При 10 000 потоках происходит следующее:

- отводится несколько гигабайтов ОЗУ под стеки;
- огромной нагрузке подвергается механизм сборки мусора, хотя память в стеке и не очищается (очень много корневых и живых объектов);
- очень много процессорного времени тратится на одно лишь контекстное переключение потоков.

Классическая модель «поток на каждый socket» неплохо послужила нам и до сих пор работает во многих приложениях. Но после выхода на определенный уровень конкурентности рост числа потоков становится опасным. Обслуживание тысячи одновременных подключений одним сервером потребительского класса – вполне обычное явление, особенно если применяются долгоживущие TCP-соединения, как в HTTP с заголовком *Keep-Alive*, при отправке серверных событий или в технологии WebSocket. Но каждый поток потребляет память (под стек) независимо от того, занимается он каким-то вычислением или простаивает в ожидании данных.

Существует два принципиально разных вида масштабируемости: горизонтальная и вертикальная. Чтобы увеличить число одновременных подключений, мы можем просто добавить новые серверы, поручив каждому обслуживанию части нагрузки. Для этого требуется поставить перед серверами балансировщик нагрузки. Но такое горизонтальное масштабирование не решает проблему C10k, поскольку в ней речь идет об одном сервере. С другой стороны, под вертикальным масштабированием понимается покупка более мощных серверов. Беда в том, что из-за блокирующего ввода-вывода памяти необходимо непропорционально много по сравнению с сильно недогруженным процессором. Даже если большой сервер корпоративного класса сможет обслужить сотни тысяч одновременных подключений (за очень большие деньги), он даже близко не подойдет к решению проблемы C10M – десять миллионов одновременных подключений. Это число взято не с

потолка; два года назад правильно спроектированному Java-приложению удалось достичь этого фантастического уровня на типичном сервере¹.

В этой главе мы совершим путешествие по различным способам реализации HTTP-сервера: однопоточный сервер, пул потоков и событийно-ориентированная архитектура. Идея в том, чтобы сравнить сложность реализации с производительностью и пропускной способностью. В конце главы мы убедимся, что вариант, основанный на RxJava, сочетает относительную простоту с выдающейся производительностью.

Традиционные HTTP-серверы на основе потоков

В этом разделе мы сравним, как блокирующие серверы, даже правильно написанные, ведут себя в условиях высокой нагрузки. Это упражнение все мы выполняли, когда учились: написать сервер, работающий на уровне сокетов. Мы реализуем совсем простенький HTTP-сервер, который посылает ответ 200 OK на любой запрос. Более того, простоты ради мы вообще будем игнорировать запрос.

Однопоточный сервер

Самое простое, что можно сделать, – открыть `ServerSocket` и обрабатывать запросы на подключение по мере поступления. Пока обслуживается один клиент, остальные запросы ставятся в очередь.

```
class SingleThread {  
  
    public static final byte[] RESPONSE = (  
        "HTTP/1.1 200 OK\r\n" +  
        "Content-length: 2\r\n" +  
        "\r\n" +  
        "OK").getBytes();  
  
    public static void main(String[] args) throws IOException {  
        final ServerSocket serverSocket = new ServerSocket(8080, 100);  
        while (!Thread.currentThread().isInterrupted()) {  
            final Socket client = serverSocket.accept();  
            handle(client);  
        }  
    }  
  
    private static void handle(Socket client) {  
        try {  
            while (!Thread.currentThread().isInterrupted()) {  
                readFullRequest(client);  
                client.getOutputStream().write(RESPONSE);  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
            IOUtils.closeQuietly(client);  
        }  
    }  
}
```

¹ <https://mrotaru.wordpress.com/2013/06/20/12-million-concurrent-connections-with-migratorydata-websocket-server/>

```

    }

    private static void readFullRequest(Socket client) throws IOException {
        BufferedReader reader = new BufferedReader(
            new InputStreamReader(client.getInputStream()));
        String line = reader.readLine();
        while (line != null && !line.isEmpty()) {
            line = reader.readLine();
        }
    }
}

```

Нигде, кроме университета, вы такую низкоуровневую реализацию не увидите, но она работает. Получив любой запрос, мы игнорируем его содержимое и без-условно возвращаем ответ 200 ОК. Перейдя в браузере по адресу `localhost:8080` мы увидим на экране строку `ok`. Класс называется `SingleThread` не без причины. Метод `ServerSocket.accept()` блокирует выполнение в ожидании запроса на подключение от клиента, а затем возвращает клиентский сокет – экземпляр класса `Socket`. Пока программа взаимодействует с `Socket` (читает и пишет), ей могут поступать новые запросы на подключение, но никто их не принимает, потому что наш поток занят обслуживанием первого клиента. Это как очередь в кабинет врача: один пациент заходит, остальные ждут. Вы обратили внимание на параметр 100 после номера прослушиваемого порта 8080? Это максимальное число запросов на подключение в очереди (по умолчанию 50). Все запросы сверх того отклоняются. Хуже того, мы притворились, что реализуем протокол HTTP/1.1, в котором по умолчанию используются долговременные соединения. И пока клиент не разорвал соединение, мы на всякий случай оставляем его открытым, блокируя тем самым новых клиентов.

Теперь вернемся к клиентскому подключению. Сначала мы должны прочесть весь запрос, а затем отправить ответ. Обе операции потенциально блокирующие и зависят от скорости сети и ее загруженности. Если, установив соединение, клиент на несколько секунд задерживает отправку запроса, то все остальные клиенты должны ждать. Очевидно, что обработка всех входящих запросов в одном потоке масштабируется так себе и едва-едва решает проблему C1 (одно одновременное подключение).

В приложении А приведен исходный код других блокирующих серверов с обоснованием. Вместо того чтобы тратить время на подробный анализ немасштабируемых архитектур с блокированием, мы приведем их краткий перечень, а затем перейдем к тестам производительности и сравнению разных подходов.

В разделе «Системный вызов `fork()` в программе на С» приведен исходный код простого сервера, написанного на С с использованием системного вызова `fork()`. Несмотря на кажущуюся простоту, порождение нового процесса для каждого клиентского подключения, особенно кратковременного, – большая нагрузка на операционную систему. Каждому процессу нужно выделить память, да и его инициализация занимает время. К тому же, непрерывный запуск и остановка тысяч процессов – это непроизводительная трата системных ресурсов.



В разделе «Один поток – одно подключение» реализован блокирующий сервер, который создает новый поток для каждого клиентского подключения. Предположительно такое решение должно неплохо масштабироваться, но реализация страдает теми же проблемами, что и `fork()` на C: для создания нового потока требуется время и ресурсы, и это особенно расточительно в случае кратковременных подключений. А если не наложить ограничения на какие-то аспекты вычислительной системы, то эти ограничения сами найдут вас, причем в самый неожиданный и неподходящий момент. Так, наша программа начинала работать нестабильно и в конечном итоге аварийно завершалась с исключением `OutOfMemoryError`, когда число одновременных подключений составляло несколько тысяч.

В разделе «Пул потоков для обслуживания подключений» рассматривается сервер `ThreadPool`, в котором также используется один поток на одно подключение, но после отключения клиента поток используется повторно, так что мы не несем каждый раз издержки на создание потока. Именно так работают все популярные контейнеры сервлетов, например `Tomcat` и `Jetty`, которые по умолчанию создают пул, содержащий от 100 до 200 потоков. В `Tomcat` имеется так называемый NIO-коннектор, который позволяет выполнять некоторые операции с сокетами асинхронно, но основная часть работы, совершаемой в сервлетах и надстроенных над ними каркасах, все равно блокирующая. Это означает, что традиционные приложения принципиально ограничены парой тысяч подключений, даже если пользуются современными контейнерами сервлетов.

Неблокирующий HTTP-сервер на основе Netty и RxNetty

Теперь сосредоточимся на событийно-ориентированных подходах к написанию HTTP-сервера, куда более многообещающих с точки зрения масштабируемости. Очевидно, что блокирующая модель обработки с использованием отдельного потока на каждый запрос, масштабируется плохо. Нам нужен способ, который позволит управлять одновременными клиентскими подключениями с применением небольшого числа потоков. У такого подхода немало преимуществ:

- снижается потребление памяти;
- улучшается использование процессора и его кэша;
- заметно повышается масштабируемость на одном узле.

Один из недостатков – утрата простоты и понятности. Потокам запрещено блокироваться при выполнении операций, мы больше не можем притвориться, что получение или отправка данных через сеть – то же самое, что вызов локального метода. Задержка непредсказуема, а время ответа на несколько порядков больше. Когда вы читаете этот текст, на рынке еще, наверное, остались вращающиеся жесткие диски, которые работают даже медленнее локальной сети. В этом разделе мы разработаем крохотное событийно-ориентированное приложение на основе каркаса `Netty`, а затем переработаем его под `RxNetty` (<https://github.com/ReactiveX/RxNetty>). И в конце приведем результаты тестов производительности для всех рассмотренных подходов.

Netty (<http://netty.io/>) – полностью событийно-ориентированный каркас; мы никогда не блокируем выполнение в ожидании получения или отправки данных. Просто байты в виде экземпляров класса `ByteBuf` передаются конвейеру обработки. При работе с TCP/IP создается впечатление, что между двумя компьютерами установлено соединение, по которому течет последовательный поток байтов. Но в действительности TCP/IP построен на базе протокола IP, который умеет передавать данные только блоками – *пакетами*. На операционную систему возлагается обязанность собрать пакеты в правильном порядке и создать иллюзию непрерывного потока. Netty отбрасывает эту абстракцию и работает на уровне последовательности байтов, а не потока. Как только хотя бы несколько байтов поступает приложению, Netty уведомляет наш обработчик. А при отправке байтов мы получаем объект `ChannelFuture` без какого бы то ни было блокирования (подробнее о будущих объектах чуть ниже).

В нашем неблокирующем HTTP-сервере будет три компонента. Первый запускает сервер и настраивает окружение:

```
import io.netty.bootstrap.ServerBootstrap;
import io.netty.channel.*;
import io.netty.channel.nio.NioEventLoopGroup;
import io.netty.channel.socket.nio.NioServerSocketChannel;

class HttpTcpNettyServer {

    public static void main(String[] args) throws Exception {
        EventLoopGroup bossGroup = new NioEventLoopGroup(1);
        EventLoopGroup workerGroup = new NioEventLoopGroup();
        try {
            new ServerBootstrap()
                .option(ChannelOption.SO_BACKLOG, 50_000)
                .group(bossGroup, workerGroup)
                .channel(NioServerSocketChannel.class)
                .childHandler(new HttpInitializer())
                .bind(8080)
                .sync()
                .channel()
                .closeFuture()
                .sync();
        } finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}
```

Это самый простой HTTP-сервер, написанный в Netty. Важнейшие его части – пул `bossGroup`, который отвечает за прием входящих подключений, и пул `workerGroup` для обработки событий. Эти пулы невелики: в пуле `bossGroup` один поток, а в `workerGroup` – примерно столько, сколько имеется процессорных ядер. Но этого с лихвой хватает в хорошо написанном сервере на основе Netty. Мы еще не сказали, что должен делать сервер, помимо прослушивания порта 8080. Это настраивается с помощью класса `ChannelInitializer`:

```
import io.netty.channel.ChannelInitializer;
import io.netty.channel.socket.SocketChannel;
import io.netty.handler.codec.http.HttpServerCodec;

class HttpInitializer extends ChannelInitializer<SocketChannel> {

    private final HttpHandler httpHandler = new HttpHandler();

    @Override
    public void initChannel(SocketChannel ch) {
        ch
            .pipeline()
            .addLast(new HttpServerCodec())
            .addLast(httpHandler);
    }
}
```

Вместо того чтобы предоставлять единственную функцию для обработки запроса, мы строим конвейер, который обрабатывает входящие экземпляры `ByteBuf` по мере поступления. На первом шаге конвейера из поступивших байтов составляются высокоуровневые объекты HTTP-запросов. Это встроенный обработчик. Он используется также для представления HTTP-ответов в виде последовательности байтов. В более надежных приложениях мы часто встречаем дополнительные обработчики для решения более мелких задач: декодирование фреймов, декодирование протоколов, обеспечение безопасности и т. д. Все данные и уведомления проходят через такой конвейер.

Вы, наверное, уже увидели здесь аналогию с RxJava. На втором месте в конвейере находится компонент бизнес-логики, который обрабатывает запрос, а не просто перехватывает или обогащает его. Хотя класс `HttpServerCodec` по необходимости хранит данные о состоянии (он транслирует входящие пакеты в высокоуровневые экземпляры `HttpRequest`), наш собственный класс `HttpHandler` может быть синглтоном без состояния:

```
import io.netty.channel.*;
import io.netty.handler.codec.http.*;

@Sharable
class HttpHandler extends ChannelInboundHandlerAdapter {

    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) {
        ctx.flush();
    }

    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) {
        if (msg instanceof HttpRequest) {
            sendResponse(ctx);
        }
    }

    private void sendResponse(ChannelHandlerContext ctx) {
```

```

final DefaultFullHttpResponse response = new DefaultFullHttpResponse(
    HTTP_1_1,
    HttpResponseStatus.OK,
    Unpooled.wrappedBuffer("OK".getBytes(UTF_8)));
response.headers().add("Content-length", 2);
ctx.writeAndFlush(response);
}

@Override
public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) {
    log.error("Error", cause);
    ctx.close();
}
}

```

Построив объект ответа `DefaultFullHttpResponse`, мы отправляем его методом `writeAndFlush()`. Но в отличие от метода `write()` обычных сокетов, этот метод не является блокирующим, а возвращает объект `ChannelFuture`. Мы подписываемся на него методом `addListener()`, в котором асинхронно закрываем канал:

```

ctx
    .writeAndFlush(response)
    .addListener(ChannelFutureListener.CLOSE);

```

Канал – это абстракция коммуникационного канала, например HTTP-подключения, поэтому закрытие канала закрывает и подключение. Но мы не хотим это делать, чтобы поддерживать долговременные соединения.

В Netty используется очень немного потоков для обработки тысяч соединений. Мы не храним для каждого соединения тяжеловесные структуры данных и не заводим для них отдельные потоки. Это гораздо ближе к тому, что происходит на уровне «железа». Компьютер получает IP-пакет и будит процесс, прослушивающий указанный в нем порт получателя. TCP/IP-соединения – это лишь абстракция, которую часто реализуют с помощью потоков. Но если приложение предъявляет жесткие требования к нагрузке и количеству соединений, то работа на уровне пакетов позволит удовлетворить их с большим успехом. Однако каналы (облегченное представление потоков) и конвейеры обработчиков, возможно с запоминанием состояния, все равно остаются.

RxNetty и сервер в виде объекта Observable

Netty – становой хребет, на котором базируются многие успешные продукты и каркасы, например: Akka (<http://akka.io/>), Elasticsearch (<https://www.elastic.co/>), HornetQ (<http://hornetq.jboss.org/>), Play (<https://www.playframework.com/>), Ratpack (<https://ratpack.io/>) и Vert.x (<http://vertx.io/>). Существует тонкая обертка вокруг Netty, адаптирующая его API к RxJava. Мы перепишем наш неблокирующий сервер на RxNetty (<https://github.com/ReactiveX/RxNetty>). Но начнем с простого сервера валютных курсов, чтобы познакомиться с API:

```

import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;

```

```
import io.reactivex.netty.protocol.tcp.server.TcpServer;

class EurUsdCurrencyTcpServer {

    private static final BigDecimal RATE = new BigDecimal("1.06448");

    public static void main(final String[] args) {
        TcpServer
            .newServer(8080)
            .<String, String>pipelineConfigurator(pipeline -> {
                pipeline.addLast(new LineBasedFrameDecoder(1024));
                pipeline.addLast(new StringDecoder(UTF_8));
            })
            .start(connection -> {
                Observable<String> output = connection
                    .getInput()
                    .map(BigDecimal::new)
                    .flatMap(eur -> eurToUsd(eur));
                return connection.writeAndFlushOnEach(output);
            })
            .awaitShutdown();
    }

    static Observable<String> eurToUsd(BigDecimal eur) {
        return Observable
            .just(eur.multiply(RATE))
            .map(amount -> eur + " EUR is " + amount + " USD\n")
            .delay(1, TimeUnit.SECONDS);
    }
}
```

Это автономный TCP-сервер, написанный с применением RxNetty. Вы, вероятно, понимаете его основные части. Сначала мы создаем новый TCP-сервер, прослушивающий порт 8080. Мы должны также настроить конвейер. Netty предоставляет низкоуровневую абстракцию (`ByteBuf`) сообщений, проходящих по конвейеру. Первый обработчик реорганизует (разбивает и объединяет) последовательности байтов `ByteBuf`, преобразуя их в последовательности строк с помощью встроенного класса `LineBasedFrameDecoder`. Затем другой декодер преобразует объекты `ByteBuf`, содержащие полные строки в экземпляры класса `String`. Начиная с этого момента, мы работаем только со строками `String`.

При поступлении нового запроса на соединение выполняется метод обратного вызова. Объект `connection` позволяет асинхронно отправлять и принимать данные. Первым вызывается метод `connection.getInput()`. Он возвращает объект типа `Observable<String>`, который порождает значение всякий раз, как серверу поступает новая строка с запросом клиента. Возвращенный `Observable` асинхронно уведомляет нас о новых входных данных. Мы преобразуем строку в число типа `BigDecimal`. Затем с помощью метода `eurToUsd()` мы имитируем вызов какой-то службы курсов валют. Чтобы пример был более реалистичным, мы добавили искусственную задержку `delay()`, моделирующую ожидание ответа. Понятно, что метод `delay()` асинхронный, и ни о каком засыпании не может быть и речи – в это время мы продолжаем получать и преобразовывать запросы.

Полученный в результате всех преобразований объект `output` типа `Observable` подается на вход методу `writeAndFlushOnEach()`. Полагаю, что здесь все понятно — мы получаем последовательность входных данных, преобразуем ее и отправляем результат преобразования в качестве ответа. Попробуем обратиться к серверу с помощью `telnet`. Обратите внимание, что из-за смоделированной задержки часть ответов появляется не сразу, а спустя несколько запросов:

```
$ telnet localhost 8080
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
2.5
2.5 EUR is 2.661200 USD
0.99
0.99 EUR is 1.0538352 USD
0.94
0.94 EUR is 1.0006112 USD
20
30
40
20 EUR is 21.28960 USD
30 EUR is 31.93440 USD
40 EUR is 42.57920 USD
```

Мы рассматриваем этот сервер, как функцию, преобразующую запрос в ответ. Поскольку TCP/IP-соединение — это не просто функция, а поток блоков данных, возможно взаимозависимых, то `RxJava` прекрасно показывает себя в этой ситуации. Богатый набор операторов позволяет производить нетривиальные преобразования входных данных в выходные. Разумеется, выходной поток необязательно базируется на входном; например, если мы реализуем события, отправляемые сервером, то сервер просто публикует данные, и никакой вход ему не нужен.

Сервер `EurUsdCurrencyTcpServer` реактивный, потому что действует только, когда поступают данные. Мы выделяем отдельный поток для каждого клиента. Эта реализация способна поддерживать тысячи одновременных соединений, а вертикальная масштабируемость ограничена только объемом трафика, а не числом более-менее простаивающих соединений.

Поняв принцип работы `RxNetty`, мы можем вернуться к исходной задаче о HTTP-сервере, возвращающем ответы `ok`. В `RxNetty` встроена поддержка HTTP-клиентов и серверов, но мы начнем с простой реализации на базе сокетов TCP/IP:

```
import io.netty.handler.codec.LineBasedFrameDecoder;
import io.netty.handler.codec.string.StringDecoder;
import io.reactivex.netty.examples.AbstractServerExample;
import io.reactivex.netty.protocol.tcp.server.TcpServer;
import static java.nio.charset.StandardCharsets.UTF_8;

class HttpTcpRxNettyServer {

    public static final Observable<String> RESPONSE = Observable.just(
```

```

"HTTP/1.1 200 OK\r\n" +
"Content-length: 2\r\n" +
"\r\n" +
"OK");

public static void main(final String[] args) {
    TcpServer
        .newServer(8080)
        .<String, String>pipelineConfigurator(pipeline -> {
            pipeline.addLast(new LineBasedFrameDecoder(128));
            pipeline.addLast(new StringDecoder(UTF_8));
        })
        .start(connection -> {
            Observable<String> output = connection
                .getInput()
                .flatMap(line -> {
                    if (line.isEmpty()) {
                        return RESPONSE;
                    } else {
                        return Observable.empty();
                    }
                })
            ;
            return connection.writeAndFlushOnEach(output);
        })
        .awaitShutdown();
}
}

```

Памятуя о сервере `EurUsdCurrencyTcpServer`, понять, как работает `HttpTcpRxNettyServer`, довольно просто. Поскольку в учебном примере мы всегда возвращаем один и тот же ответ 200 OK, разбирать запрос не обязательно. Однако правильно написанный сервер не должен отправлять ответ, не прочитав запрос. Поэтому мы начинаем с поиска пустой строки в тексте, возвращенном `getInput()`, поскольку пустая строка обозначает конец HTTP-запроса. И лишь после этого порождаем строку 200 OK. Построенный таким образом объект `Observable` передается методу `connection.writeString()`. Иными словами, ответ будет отправлен клиенту, как только в запросе встретится первая пустая строка².

Реализация HTTP-сервера на уровне TCP/IP – интересное упражнение, которое поможет понять тонкости протокола HTTP. По счастью, нам необязательно реализовывать HTTP и REST-совместимые веб-сервисы, пользуясь только абстракцией TCP/IP. Как и `Netty`, `RxNetty` предлагает целый ряд встроенных компонентов для обслуживания HTTP:

```

import io.reactivex.netty.protocol.http.server.HttpServer;

class RxNettyHttpServer {

    private static final Observable<String> RESPONSE_OK =

```

² На самом деле, первая пустая строка обозначает лишь конец заголовков HTTP-запроса, а не всего запроса. Далее может следовать тело запроса. Но для учебного примера это не так важно. – *Прим. перевод.*

```
Observable.just("OK");

public static void main(String[] args) {
    HttpServer
        .newServer(8086)
        .start((req, resp) ->
            resp
                .setHeader(CONTENT_LENGTH, 2)
                .writeStringAndFlushOnEach(RESPONSE_OK)
        ).awaitShutdown();
}
```

Если вам скучно возвращать один и тот же ответ 200 ОК, то можно без особого труда построить неблокирующий REST-совместимый веб-сервис, опять-таки для курсов валют:

```
class RestCurrencyServer {

    private static final BigDecimal RATE = new BigDecimal("1.06448");

    public static void main(final String[] args) {
        HttpServer
            .newServer(8080)
            .start((req, resp) -> {
                String amountStr = req.getDecodedPath().substring(1);
                BigDecimal amount = new BigDecimal(amountStr);
                Observable<String> response = Observable
                    .just(amount)
                    .map(eur -> eur.multiply(RATE))
                    .map(usd ->
                        "{\"EUR\": " + amount + ", " +
                        "\"USD\": " + usd + "}");
                return resp.writeString(response);
            })
            .awaitShutdown();
    }
}
```

С этим сервером можно взаимодействовать с помощью браузера или программы curl. Вызов `substring(1)` в начале необходим, чтобы вырезать из запроса первый знак косой черты:

```
$ curl -v localhost:8080/10.99
> GET /10.99 HTTP/1.1
> User-Agent: curl/7.35.0
> Host: localhost:8080
> Accept: */*
>
< HTTP/1.1 200 OK
< transfer-encoding: chunked
<
{"EUR": 10.99, "USD": 11.6986352}
```


Имея несколько реализаций простого HTTP-сервера, мы можем сравнить их с точки зрения производительности, масштабируемости и пропускной способности. Ведь именно ради этого мы отказались от знакомой модели на основе потоков и перешли на использование RxJava и асинхронных API.

Сравнение производительности блокирующего и реактивного сервера

Чтобы доказать, что затраты на написание неблокирующего реактивного HTTP-сервера окупаются, мы выполним ряд тестов производительности для каждой реализации. Интересно, что выбранный нами инструмент тестирования `wrk` (<https://github.com/wg/wrk>) также неблокирующий; иначе он не смог бы смоделировать нагрузку, эквивалентную десяткам тысяч одновременных соединений. Интересная альтернатива – программа `Gatling` (<http://gatling.io/>), написанная с применением библиотеки `Akka`. Традиционные потоковые инструменты нагрузочного тестирования типа `JMeter` и `ab` (<https://httpd.apache.org/docs/2.4/programs/ab.html>) неспособны создать такую высокую нагрузку и сами становятся узким местом.

Все реализации на базе JVM³ тестировались при 10 000, 20 000 и 50 000 одновременных HTTP-клиентов, т. е. TCP/IP-соединений. Нас интересовало количество обслуженных запросов в секунду (пропускная способность), а также медиана и 99-й процентиль времени ответа. Напомню: медиана означает, что 50 % ответов обслуживались не менее быстро, а 99-й процентиль говорит, что 1 % запросов обслуживался медленнее данной величины.



Тестовая среда

Все тесты выполнялись на домашних ноутбуках под управлением Linux с ядром 3.13.0-62-generic, оснащенных процессором Intel i7 CPU 2.4 ГГц, 8 ГБ оперативной памяти и SSD-диском. На машине-клиенте работали официальные сборки `wrk`, `Gatling` и `JMeter`. Клиенты подключались к серверу через один маршрутизатор Gigabit Ethernet. Среднее время `ping` между клиентом и сервером составляло 289 мкс (стандартное отклонение 42 мкс, минимум 160 мкс).

Каждый тест продолжался не менее одной минуты плюс 30 секунд на прогрев. Использовались JDK 1.8.0_66, RxJava 1.0.14, RxNetty 0.5.1 и Netty 4.0.32.Final. Нагрузка на систему замерялась с помощью программы `htop`.

Тесты запускались следующей командой (с переменным параметром `-c`, представляющим количество одновременных клиентов):

```
wrk -t6 -c10000 -d60s --timeout 10s --latency http://server:8080
```

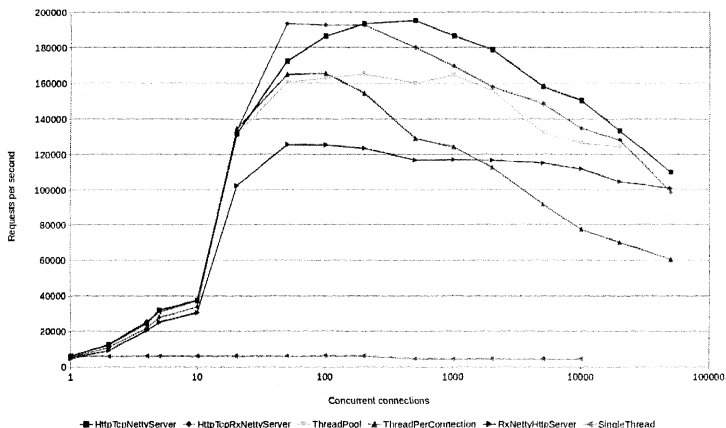
Простой сервер, возвращающий 200 OK

В первом тесте сравнивалась производительность различных реализаций, которые всегда возвращают ответ 200 OK и не выполняют никаких серверных задач.

³ Мы исключаем программы, написанные на C и для других реактивных платформ, например `Node.js`.

Это не очень реалистичный тест, но он дает представление о верхних пределах сервера и сети Ethernet. В последующих тестах мы добавим случайную задержку на сервере.

На рисунке ниже показано количество запросов в секунду, обрабатываемых каждой реализацией (в логарифмическом масштабе).



Помните, что это только прикидочный тест, предшествующий более реалистичным сценариям, подразумевающим выполнение какой-то работы на сервере. Но уже здесь проявляются некоторые интересные тенденции.

- Реализации на основе Netty и RxNetty демонстрируют наибольшую пропускную способность, достигающую 200 000 запросов в секунду.
- Не удивительно, что SingleThread заметно медленнее – способен обработать примерно 6 000 запросов в секунду вне зависимости от уровня конкурентности.
- Но SingleThread оказывается самой быстрой реализацией при наличии единственного клиента. Накладные расходы на организацию пула потоков, событийную ориентированность в случае (Rx)Netty и другие детали реализации не проходят бесследно. Правда, это преимущество быстро исчезает с ростом числа клиентов. Кроме того, пропускная способность сервера сильно зависит от производительности клиента.
- Как ни странно, сервер ThreadPool показывает очень неплохую производительность, но становится нестабильным при высокой нагрузке (много ошибок в отчете wrk) и окончательно «ломается» при 50 000 одновременных клиентов (время ожидания ответа превышает 10 секунд).

- Сервер `ThreadPoolPerConnection` тоже работает очень хорошо, но когда число потоков превышает 100–200, пропускная способность быстро падает. При 50 000 потоков нагрузка на JVM становится непереносимой, отведение нескольких гигабайтов под стеки вызывает серьезные проблемы.

Не будем тратить много времени на анализ результатов этого искусственного теста. В конце концов, реальные серверы редко возвращают ответ немедленно. Попробуем вместо этого смоделировать работу, выполняемую при обработке запроса.

Моделирование работы на сервере

Чтобы смоделировать работу на сервере, мы просто вставим вызов `sleep()` между запросом и ответом. Это честно: во время обработки запроса серверы зачастую не сильно занимают процессор. Традиционный сервер блокирует выполнение в ожидании данных от внешних ресурсов, занимая один поток. Реактивный же сервер ждет внешнего сигнала (например, события или сообщения, содержащего ответ), не занимая в это время никаких ресурсов.

Поэтому в блокирующих реализациях мы добавим `sleep()`, а в неблокирующих будем использовать оператор `Observable.delay()` или аналогичный механизм для имитации медленного ответа от внешней службы:

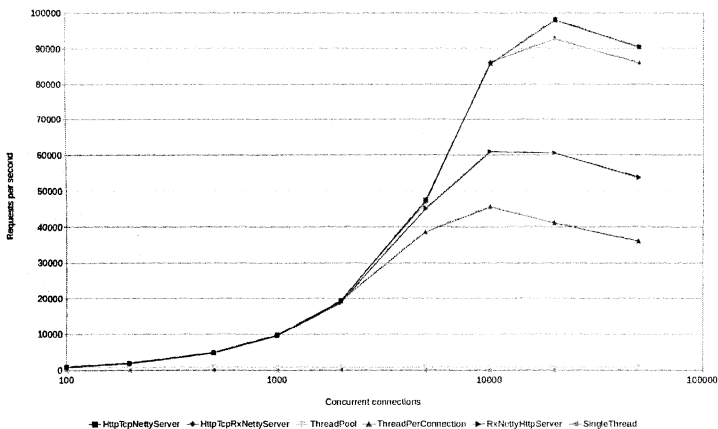
```
public static final Observable<String> RESPONSE = Observable.just(
    "HTTP/1.1 200 OK\r\n" +
    "Content-length: 2\r\n" +
    "\r\n" +
    "OK")
    .delay(100, MILLISECONDS);
```

В блокирующей реализации нет смысла использовать неблокирующую задержку, потому что она все равно должна дожидаться ответа, даже если реализация задержки неблокирующая. Итак, мы вставили задержку на 100 миллисекунд в обработку каждого ответа, поэтому взаимодействие с сервером продолжается не менее одной десятой секунды. Теперь тест производительности станет гораздо более реалистичным и интересным. На следующем графике показана зависимость количества запросов в секунду от числа клиентских соединений.

Результаты ближе к тому, что можно ожидать в условиях реальной нагрузки. Обе реализации на основе `Netty` (`HttpTcpNettyServer` и `HttpTcpRxNettyServer`), которым соответствуют верхние кривые, намного превосходят остальные и легко справляются с 90 000 запросов в секунду (RPS). На самом деле, вплоть до 10 000 одновременных клиентов сервер масштабируется линейно. И доказать это очень просто: один клиент генерирует примерно 10 RPS (каждый запрос занимает около 100 мс, так что за 1 секунду клиент успевает послать 10 запросов). Два клиента генерируют 20 RPS, 5 клиентов – 50 RPS и т. д. При 10 000 одновременных соединений следует ожидать 100 000 RPS, так что мы близко подошли к теоретическому пределу (90 000 RPS).

В нижней части графика расположены серверы `SingleThread` и `ThreadPool`. Их производительность совсем ничтожна, что и не удивительно. Если все запросы об-

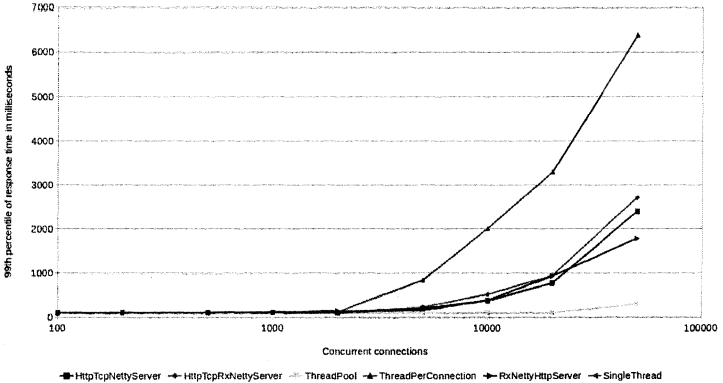
работывает один поток, а каждый запрос занимает не менее 100 мс, то понятно, что более 10 запросов в секунду не обработать. `ThreadPool` намного лучше, здесь есть 100 потоков, каждый из которых обрабатывает 10 RPS, что в сумме дает 1000 RPS. Но эти результаты на несколько порядков хуже тех, что показывают реализации на основе реактивных `Netty` и `RxJava`. Кроме того, сервер `SingleThread` при высокой нагрузке *отклоняет почти все запросы*. При 50 000 одновременных соединений он принимал какую-то мизерную долю запросов, но почти ни разу не смог уложиться в 10-секундный таймаут, выставленный `wrk`.



Вы спрашиваете, почему в сервере `ThreadPool` мы ограничились всего лишь 100 потоками? Потому что именно таково умолчание в популярных контейнерах серверов, но, конечно, можно было бы задать и больше. Поскольку все соединения долговременные и поток из пула захватывается на все время существования соединения, мы можем рассматривать реализацию `ThreadPerConnection` как пул с неограниченным числом потоков. Удивительно, но такая реализация работает вполне прилично, даже когда JVM должна управлять 50 000 конкурентных потоков, каждый из которых представляет одно соединение. Фактически `ThreadPerConnection` не намного хуже `RxNettyHttpServer`. Но одной пропускной способности, измеренной в RPS, не достаточно, нужно еще проанализировать время ответа на каждый запрос. Оно зависит от требований, но обычно мы хотим получить одновременно высокую пропускную способность, чтобы были задействованы все ресурсы сервера, и малое время ответа — т. е. высокую *воспринимаемую* производительность.

Среднее время ответа редко может служить хорошим индикатором. С одной стороны, среднее скрывает выбросы (немногие запросы, которые обрабатывались недопустимо медленно). С другой стороны, *типичное* время ответа (наблюдаемое большинством клиентов) меньше среднего, опять-таки из-за выбросов. Проценти-

ли намного более показательны, поскольку описывают распределение конкретного значения. На следующем графике показана зависимость 99-го перцентиля времени ответа от числа одновременных соединений (клиентов) для всех реализаций. Значение на оси Y говорит, что время обработки 99 % запросов было меньше данного значения. Очевидно, мы хотим, чтобы эти значения были как можно меньше (но они не могут оказаться меньше 100 мс – модельной задержки) и с увеличением нагрузки росли как можно медленнее,



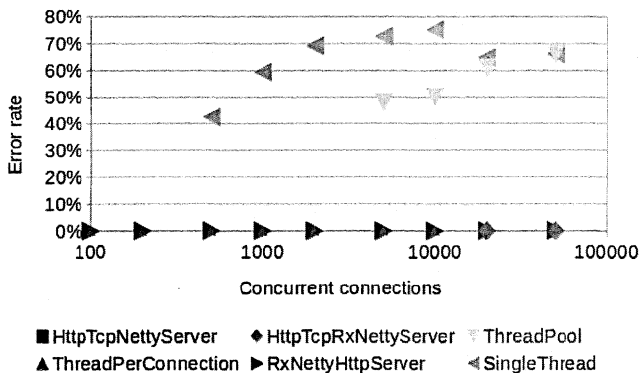
Реализация `ThreadPerConnection` выделяется в худшую сторону. Пока число одновременных соединений не превышает 1000, все реализации идут ноздря в ноздю. Но в какой-то момент `ThreadPerConnection` начинает отвечать медленно – в несколько раз медленнее соперников. Тому есть две основных причины: во-первых, слишком много контекстных переключений между тысячами потоков и, во-вторых, более частые циклы сборки мусора. То есть JVM тратит много времени на служебные операции, а для полезной работы почти ничего не остается. Тысячи одновременных соединений простаивают в ожидании своей очереди.

Удивляетесь, почему сервер `ThreadPool` имеет такой выдающийся 99-й процентиль? Он заметно превосходит в этом отношении все прочие реализации и остается стабильным даже при высокой нагрузке. Но давайте вспомним, как выглядит реализация `ThreadPool`:

```
BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(1000);
executor = new ThreadPoolExecutor(100, 100, 0L, MILLISECONDS, workQueue,
    (r, ex) -> {
        ((ClientConnection) r).serviceUnavailable();
    });
```

Вместо того чтобы использовать построитель `Executors`, мы конструируем объект `ThreadPoolExecutor` напрямую, беря на себя контроль над очередью `workQueue` и

обработчиком `RejectedExecutionHandler`. Последний выполняется, когда в очереди нет места. По существу, мы предотвращаем перегрузку сервера, отбрасывая запросы, которые невозможно выполнить немедленно. Ни у какой другой реализации нет такого механизма защиты, который часто называют *быстрым прекращением*. Мы еще вернемся к этому вопросу в разделе «Управление отказами с помощью Hystrix» главы 8, а пока сопоставим время реакции `ThreadPool` с частотой ошибок:



Согласно ответу `wrk`, количество ошибок исчезающе мало во всех реализациях, кроме `SingleThread` и `ThreadPool`. Это любопытный компромисс: сервер `ThreadPool` всегда отвечает с максимальной скоростью, гораздо быстрее соперников. Но он также без колебаний отклонит запрос, если в данный момент перегружен. Разумеется, подобный механизм можно включить и в реактивные реализации на основе `Netty/RxJava`.

Подведем итоги. Пулы потоков и независимые потоки не могут удовлетворить требования к пропускной способности и времени ответа, с которыми легко справляются реактивные реализации.

Обзор реактивных HTTP-серверов

Стек протоколов TCP/IP и построенный на его основе HTTP являются принципиально событийно-ориентированными. Несмотря на искусственно созданную иллюзию конвейерного входа и выхода, в глубине находятся пакеты данных, прибывающие асинхронно. Поэтому абстракция представления сетевого стека в виде блокирующего потока байтов «протекает», как почти любая абстракция в информатике. Особенно это заметно, если мы хотим в полной мере задействовать все возможности оборудования.

Классические подходы к сетевому программированию вполне пригодны, даже при умеренной нагрузке. Но если требуется масштабирование до пределов, неслыханных в традиционных Java-приложениях, то приходится переходить на ре-

активную парадигму. Netty – прекрасный каркас для построения реактивных, событийно-ориентированных сетевых приложений, но напрямую он используется редко. Вместо этого он входит составной частью во многие библиотеки и каркасы, в т. ч. RxNetty. Каркас RxNetty особенно интересен, потому что сочетает мощь событийно-ориентированного сетевого программирования с простотой операторов RxJava. Мы по-прежнему рассматриваем коммуникацию по сети как поток сообщений (пакетов), но абстрагируем его типом `Observable<ByteBuf>`.

Вы помните поставленную в начале главы проблему 10 000 одновременных соединений? Нам удалось решить ее с помощью реализаций на основе Netty и RxNetty. На самом деле, мы успешно реализовали HTTP-серверы, способные поддерживать 50 000 одновременных постоянных соединений, т. е. решили проблему C50k. При наличии большого числа физических клиентов (с сервером-то все в порядке) и меньшей частоте запросов на линиях связи те же реализации легко могли бы выдержать 100 000 и более одновременных соединений, хотя код занимает всего десяток строк.

Понятно, что реализация серверной части протокола HTTP (да и любого протокола, HTTP был выбран только из-за его широкой распространенности) – лишь часть истории. Важно еще, что сервер делает, а, как правило, он выступает в роли клиента какого-то другого сервера. Поэтому мы в этой главе сконцентрировали внимание на реактивных неблокирующих HTTP-серверах. Это разумно, но есть немало мест – подчас совершенно неожиданных – где может просочиться блокирующий код. Прежде всего, мы уделили много внимания серверной части, но целиком упустили из виду клиентскую. Однако современные серверы, особенно в больших распределенных системах, играют также и роль клиента, поскольку запрашивают данных у других сервисов или передают им свои данные. Вполне можно предположить, что один-единственный запрос к популярной поисковой системе распространяется на сотни или даже тысячи расположенных за ее фасадом компонентов, т. е. имеет место множество клиентских запросов. Очевидно, что если бы эти запросы были последовательными и блокирующими, то время ответа системы оказалось бы недопустимо большим.

Какого бы совершенства мы ни достигли в реализации инфраструктурного кода сервера, если он вынужден иметь дело с блокирующими API, то масштабируемость пострадает, как показали наши тесты производительности. В экосистеме Java известно несколько источников блокирования, которые мы далее кратко рассмотрим.

Код HTTP-клиента

Широко известны серверы, которые обращаются к другим службам и объединяют полученные ответы. Вы без труда найдете десятки стартапов, которым удалось интересными способами объединить доступные источники данных и предоставить на этой основе службу, имеющую самостоятельную ценность. Современные API в основном строятся на принципах REST, а роль SOAP неуклонно уменьшается – но обе технологии основаны на вездесущем протоколе HTTP.

Даже один блокирующий запрос может «положить» сервер, значительно снизив его производительность. К счастью, есть немало зрелых неблокирующих HTTP-клиентов, и с одним из них – Netty – мы уже познакомились. Неблокирующие HTTP-клиенты пытаются решить два класса задач.

- Большое количество независимых одновременных запросов, каждый из которых требует нескольких клиентских обращений к сторонним API. Это типично для сервисно-ориентированных архитектур, когда для ответа на один запрос привлекается несколько сервисов.
- Сервер выполняет много клиентских HTTP-запросов, например, в ходе выполнения пакетных операций. Вспомните о поисковых роботах или службах индексирования, которые постоянно держат открытыми несколько тысяч соединений.

Но независимо от природы сервера проблема остается одной и той же: поддержка большого числа (десятков тысяч и более) открытых HTTP-соединений сопряжена с высокими накладными расходами. И это особенно неприятно, если службы, к которым мы подключаемся (на сей раз в роли клиента), работают медленно, а, значит, мы должны удерживать ресурсы в течение длительного времени.

Само же TCP/IP-соединение обходится довольно дешево. Операционная система должна хранить дескриптор сокета для каждого открытого соединения (примерно один килобайт) – и это, в общем-то, всё. Поступивший пакет (сообщение) ядро отправляет соответствующему процессу, например JVM. Один килобайт – это совсем немного по сравнению с приблизительно одним мегабайтом, который отводится под стек каждого потока, заблокированного в ожидании данных из сокета. Поэтому классическая модель «одно соединение – один поток» плохо масштабируется на случай высокопроизводительных серверов, и мы должны научиться мирно сосуществовать с истинной моделью сетевого взаимодействия, а не пытаться абстрагировать ее с помощью блокирующего кода. А хорошая новость заключается в том, что комбинация RxJava + Netty дает гораздо лучшую абстракцию, приближенную к «железу».

Неблокирующий HTTP-клиент на основе RxNetty

RxJava в сочетании с Netty предлагает абстракцию, достаточно близкую к тому, как в действительности работает сеть. Не делая вид, будто HTTP-запрос – почти то же самое, что обычный вызов метода в пределах JVM, эта абстракция включает в себя асинхронность. Более того, мы уже не можем притвориться, что HTTP – это просто протокол типа «запрос – ответ». Наступление событий, посылаемых сервером (один запрос, несколько ответов), веб-сокеты (полнодуплексная связь) и, наконец, HTTP/2 (множество параллельных запросов и ответов по одной линии связи, чередующихся друг с другом) открывает совершенно новые сценарии использования HTTP.

На стороне клиента RxNetty предоставляет лаконичный API для простейших сценариев. Мы отправляем запрос и получаем в ответ объекты `Observable`, допускающие композицию:


```
Observable<ByteBuf> response = HttpClient
    .newClient("example.com", 80)
    .createGet("/")
    .flatMap(HttpClientResponse::getContent);
response
    .map(bb -> bb.toString(UTF_8))
    .subscribe(System.out::println);
```

Метод `createGet()` возвращает объект класса, производного от `Observable<HttpClientResponse>`. Очевидно, что клиент не блокируется в ожидании ответа, так что `Observable` кажется вполне подходящим выбором. Но это только начало. В самом классе `HttpClientResponse` имеется метод `getContent()`, который возвращает объект типа `Observable<ByteBuf>`. В разделе «Неблокирующий HTTP-сервер на основе Netty и RxNetty» выше мы говорили, что `ByteBuf` – абстракция блока данных, передаваемого по сети. С точки зрения клиента, это *часть* ответа. А раз так, то RxNetty идет дальше по сравнению с другими неблокирующими HTTP-клиентами и уведомляет нас не тогда, когда пришел весь ответ. Вместо этого мы получаем поток сообщений `ByteBuf`, возможно завершаемый уведомлением о завершении `Observable`, когда сервер решит разорвать соединение.

Такая модель гораздо ближе к тому, как работает стек TCP/IP, и лучше масштабируется на различные способы использования. Она может работать как с простой последовательностью «запросов-ответ», так и в сложных сценариях, включающих потоки сообщений. Но помните, что даже единственный ответ, например содержащий HTML-разметку, скорее всего, поступает в виде нескольких блоков. Разумеется, в RxJava есть масса способов собрать их воедино, например: `Observable.toList()` или `Observable.reduce()`. Но это ваш выбор: если хотите потреблять данные по мере поступления, мелкими порциями, то никто вам не будет препятствовать. В этом отношении RxNetty – инструмент очень низкого уровня, потому что абстракция не создает таких узких мест, как избыточная буферизация или блокирование, чтобы сохранить исключительно высокую степень масштабируемости. Если вас интересует реактивный, надежный, но более высокоуровневый HTTP-клиент, обратитесь к разделу «Библиотека Retrofit со встроенной поддержкой RxJava» главы 8.

В отличие от реактивных API на основе обратных вызовов, RxNetty прекрасно уживается с другими `Observable`, поэтому можно легко распараллеливать, комбинировать и разбивать работу на части. Представим, к примеру, что имеется поток URL-адресов, к которым нужно в режиме реального времени подключаться и получать данные. Поток может быть фиксированным (построенным из обычного списка `List<URL>`) или динамическим – когда новые адреса поступают непрерывно. Если требуется построить постоянный поток пакетов из всех этих источников, то можно просто применить к ним оператор `flatMap()`:

```
Observable<URL> sources = //...

Observable<ByteBuf> packets =
    sources
    .flatMap(url -> HttpClient
```

```
.newClient(url.getHost(), url.getPort())
.createGet(url.getPath())
.flatMap(HttpClientResponse::getContent);
```

Это несколько искусственный пример, поскольку мы мешаем в одну кучу сообщения `ByteBuf` из разных источников, но идея понятна. Для каждого `URL` из входного `Observable` мы порождаем асинхронный поток экземпляров `ByteBuf`, поступающих из этого `URL`. Если мы захотим сначала преобразовать входящие данные, например, собрав отдельные блоки в одно событие, то легко сможем это сделать, например, с помощью `reduce()`. Итог таков: мы легко можем поддерживать десятки тысяч открытых HTTP-соединений, простаивающих или получающих данные. Лимитирующим фактором является уже не память, а вычислительная мощность процессора и пропускная способность сети. JVM не нужно выделять гигабайты памяти для обработки вполне обозримого числа транзакций.

API для работы с HTTP – одно из основных узких мест в современных приложениях. Они обходятся недорого в плане потребления ресурсов процессора, но блокирующее обращение по HTTP ведет себя как обычный процедурный вызов, что значительно ограничивает масштабируемость. Но даже если тщательно выскрести все блокирующие обращения по HTTP, синхронный код может проявиться в совершенно неожиданных местах. Такой подвох есть в методе `equals()` из класса `java.net.URL`: он обращается с запросом к сети. Да, именно так: при сравнении двух экземпляров класса `URL` этот обманчиво быстрый метод посылает запрос и получает ответ. Ниже показана последовательность вызовов, читайте сверху вниз:

```
java.net.URL.equals(URL.java)
java.net.URLStreamHandler.equals(URLStreamHandler.java)
java.net.URLStreamHandler.sameFile(URLStreamHandler.java)
java.net.URLStreamHandler.hostsEqual(URLStreamHandler.java)
java.net.URLStreamHandler.getHostAddress(URLStreamHandler.java)
java.net.InetAddress.getByName(InetAddress.java)
java.net.InetAddress.getAllByName(InetAddress.java)
java.net.InetAddress.getAllByName0(InetAddress.java)
java.net.InetAddress.getAddressesFromNameService(InetAddress.java)
java.net.InetAddress$2.lookupAllHostAddr(InetAddress.java)
[машинный код]
```

Чтобы узнать, равны ли два объекта `URL`, JVM вызывает метод `lookupAllHostAddr()`, который (в секции машинного кода) вызывает функцию `gethostbyname` (или аналогичную), которая обращается с синхронным запросом к DNS-серверу. Эффект может быть катастрофическим – если имеется горстка потоков, и некоторые из них неожиданно блокируются. Вспомните хотя бы серверы на основе `RxNetty` – в них использовалось всего несколько десятков потоков. Другая чреватая печальными последствиями ситуация может возникнуть, если метод `URL.equals()` вызывается часто, например, при работе с коллекцией `Set<URL>`. Это неожиданное поведение класса `URL` довольно широко известно, как и тот факт, что метод `equals()` может давать разные результаты в зависимости от связности Интернета в момент вызова.

Мы привели этот факт, как иллюстрацию того, что написание по-настоящему реактивных приложений – дело трудное и изобилующее капканами. В следующем разделе мы рассмотрим другой, более очевидный источник блокирования: доступ к базе данных.

Доступ к реляционной базе данных

Выше мы пришли к выводу, что любой сервер рано или поздно становится клиентом какого-то другого сервиса. Еще одно наблюдение: почти любая вычислительная система, с которой нам довелось работать, была распределенной. Если два машины, разделенные сетевым кабелем, должны взаимодействовать между собой, то такую систему уже можно считать пространственно распределенной. Если довести эту мысль до логического завершения, то даже один компьютер следует считать распределенной системой, поскольку кэши отдельных процессоров не всегда согласованы и должны синхронизироваться по протоколу передачи сообщений. Но не будем заходить так далеко и ограничимся архитектурой, в которой имеется сервер приложения и сервер базы данных.

Для доступа к реляционным базам данных в Java уже давно существует стандарт *Java Database Connectivity* (JDBC). С точки зрения потребителя, JDBC представляет собой набор API для взаимодействия с любой реляционной СУБД: PostgreSQL, Oracle Database и многими другими. Основные абстракции: *Connection* (TCP/IP, проводное соединение), *Statement* (запрос к базе данных) и *ResultSet* (представление результата, полученного от базы данных). Современные разработчики редко пользуются этим API напрямую, потому что имеются более удобные абстракции: нетребовательный к ресурсам класс *JdbcTemplate* из каркаса Spring, библиотеки генерации кода типа *jOOQ* (<http://www.jooq.org/>), средства объектно-реляционного отображения типа JPA и другие. JDBC печально известен сложностью обработки ошибок в сочетании с контролируемыми исключениями (с появлением try-c-ресурсами в Java 7 все значительно упростилось):

```
import java.sql.*;

try (
    Connection conn = DriverManager.getConnection("jdbc:h2:mem:");
    Statement stat = conn.createStatement();
    ResultSet rs = stat.executeQuery("SELECT 2 + 2 AS total")
) {
    if (rs.next()) {
        System.out.println(rs.getInt("total"));
        assert rs.getInt("total") == 4;
    }
}
```

В примере выше используется встраиваемая база данных H2, которая часто применяется в интеграционном тестировании. Но в производственной системе редко встретишь базу данных на одной машине с приложением. Для любого взаи-

модействия с базой необходимо обращение к сети. Основной частью JDBC является API, который должен реализовать любой поставщик базы.

Когда клиент запрашивает у JDBC API новый объект `Connection`, реализация должна установить физическое соединение с базой данных: открыть клиентский сокет, авторизоваться и т. д. В разных базах данных применяются различные протоколы передачи данных (почти всегда двоичные), так что на реализацию JDBC (иначе говоря, драйвер) возлагается ответственность за трансляцию низкоуровневого протокола в единообразный API. Все это замечательно работает (если оставить в стороне различные диалекты SQL), но, к несчастью, в 1997 году, когда стандарт JDBC был выпущен в составе JDK 1.1, никто не предполагал, насколько важным станет реактивное асинхронное программирование через двадцать лет. Конечно, с тех пор было выпущено несколько версий API, но все они принципиально блокирующие, т. е. дожидаются завершения операции в базе данных.

Это в точности та же проблема, с которой мы столкнулись в HTTP. В приложении должно быть столько потоков, сколько имеется активных соединений с базой данных (запросов). JDBC – единственный зрелый стандарт доступа к разнообразным реляционным СУБД переносимым способом (опять-таки, о диалектах SQL забываем). Несколько лет назад спецификация сервлетов была значительно переработана в версии 3.0, когда появился метод `HttpServletRequest.startAsync()`. Очень жаль, что стандарт JDBC все еще придерживается классической модели.

Есть несколько причин, из-за которых JDBC остается блокирующим. Веб-сервер легко может поддержать сотни тысяч открытых соединений, например, если он лишь периодически отправляет небольшие блоки данных. Но СУБД для каждого запроса выполняет более-менее одинаковую последовательность шагов:

1. *Разбор запроса* (занимает процессор) – преобразование строки запроса в дерево разбора.
2. *Оптимизация запроса* (занимает процессор) – применение к дереву запроса различных правил и статистики с целью построить план выполнения.
3. *Исполнение запроса* (ограничено скоростью ввода-вывода) – поиск в хранилище данных кортежей, удовлетворяющих запросу.
4. Сериализация и отправка клиенту *результатирующего набора* (ограничено быстродействием сети).

Понятно, что базе данных необходимо много ресурсов для обработки запроса. Как правило, большая часть времени тратится на исполнение, а диски (как вращающиеся, так и SSD) уж так устроены, что если и допускают распараллеливание, то в очень малой степени. Поэтому количество одновременных запросов, которые база данных может выполнить, прежде чем достигнет насыщения, ограничено. И это ограничение зависит, главным образом, от дизайна движка СУБД и используемого оборудования. Существует также целый ряд не столь очевидных аспектов: блокировки, контекстные переключения и заполнение процессорного кэша. В общем, следует ожидать порядка нескольких сотен запросов в секунду. Это очень мало по сравнению с сотнями тысяч открытых HTTP-соединений, что легко достижимо с помощью неблокирующих API.

Зная, что пропускная способность базы данных сильно ограничена возможностями оборудования, не имеет смысла пытаться написать полностью реактивные драйверы. Технически можно реализовать протокол передачи данных поверх Netty или RxNetty, который никогда не будет блокировать поток клиента. На самом деле, есть много нестандартных, независимо разработанных решений (см. `postgresql-async` (<https://github.com/mauricio/postgresql-async>), `postgresql-async-driver` (<https://github.com/alaisi/postgres-async-driver>), `adbcj` (<https://github.com/mheath/adbcj>) и `finagle-mysql` (<http://twitter.github.io/finagle/guide/Protocols.html#mysql>)), в которых предпринимается попытка реализовать такой протокол для той или иной СУБД на основе неблокирующего сетевого стека. Но принимая во внимание, что JVM способна без особого напряжения обработать от сотен до тысяч потоков (см. раздел «Один поток – одно подключение» приложения А), мы вряд ли сильно выиграем от переписывания с нуля хорошо зарекомендовавшего себя JDBC API. Даже в библиотеке Slick (<https://www.lightbend.com/community/core-tools/slick>) из широко распространенного реактивного стека Lightbend на основе комплекта инструментов Akka на нижнем уровне используется JDBC. Существуют также реализуемые сообществом проекты, наводящие мост между RxJava и JDBC, например `rxjava-jdbc` (<https://github.com/davidmoten/rxjava-jdbc>).

Касательно взаимодействия с реляционными базами данных можно дать такую рекомендацию: использовать выделенный тщательно настроенный пул потоков, в котором изолировать блокирующий код. Прочие части приложения могут быть реактивными и работать в небольшом числе потоков, но с прагматической точки зрения стоит сохранить верность JDBC, поскольку попытка заменить его чем-то более реактивным, скорее всего, выльется в массу сложностей без очевидного выигрыша. В разделе «От коллекций к Observable» главы 4 мы уже дали несколько советов по поводу взаимодействия с JDBC в классических программных стеках. Но тем не менее, можно немного поэкспериментировать с RxJava, даже поверх блокирующего JDBC.

NOTIFY и LISTEN на примере PostgreSQL

В PostgreSQL имеется интересный встроенный механизм сообщений, доступный с помощью команд `LISTEN` и `NOTIFY`, расширяющих SQL. Любой клиент PostgreSQL может послать уведомление в виртуальный канал следующим образом:

```
NOTIFY my_channel;  
NOTIFY my_channel, '{"answer": 42}';
```

Здесь мы посылаем пустое уведомление, за которым следует произвольная строка (в формате JSON, XML или любом другом), в канал с именем `my_channel`. Канал – это, по существу, очередь внутри движка PostgreSQL. Интересно, что отправка уведомления – часть транзакции, поэтому его доставка производится только после фиксации, а в случае отката сообщение отбрасывается.

Для получения уведомлений из канала необходимо начать его прослушивание с помощью команды `LISTEN`. Получить уведомления из прослушиваемого канала

можно только одним способом: периодически вызывая метод `getNotifications()`. Это ведет к случайной задержке, неоправданной нагрузке на процессор и лишним контекстным переключениям, но, увы, так уж спроектирован API. Ниже приведен полный пример с блокированием:

```
try (Connection connection =
    DriverManager.getConnection("jdbc:postgresql:db")) {
    try (Statement statement = connection.createStatement()) {
        statement.execute("LISTEN my_channel");
    }
    Jdbc4Connection pgConn = (Jdbc4Connection) connection;
    pollForNotifications(pgConn);
}

//...

void pollForNotifications(Jdbc4Connection pgConn) throws Exception {
    while (!Thread.currentThread().isInterrupted()) {
        final PGNotification[] notifications = pgConn.getNotifications();
        if (notifications != null) {
            for (final PGNotification notification : notifications) {
                System.out.println(
                    notification.getName() + ": " +
                    notification.getParameter());
            }
        }
        TimeUnit.MILLISECONDS.sleep(100);
    }
}
```

Мало того что мы блокируем поток клиента, так еще и вынуждены держать открытым одно JDBC-соединение, потому что прослушивание привязывается к конкретному соединению. Показанный код довольно многословный, но простой. Вызвав `LISTEN`, мы входим в бесконечный цикл, в котором запрашиваем новые уведомления. Метод `getNotifications()` разрушающий, т. е. возвращенные уведомления отбрасываются, поэтому при повторном вызове мы не получим те же самые события. Метод `getName()` возвращает имя канала (например, `my_channel`), а `getParameter()` — содержимое необязательного события, например, полезную нагрузку в формате JSON.

Этот API выглядит ужасно старомодно, отсутствие уведомлений в нем индицируется возвратом `null`, а вместо коллекций используются массивы. Давайте-ка сделаем его более дружелюбным по отношению к Rx. Раз нет механизма проталкивания уведомлений, то придется реализовать опрос с помощью неблокирующего оператора `interval()`. Нужно подумать о разных мелких деталях, без которых наш пользовательский тип `Observable` не будет вести себя правильно; мы обсудим их после примера (пока еще не полного).

```
Observable<PGNotification> observe(String channel, long pollingPeriod) {
    return Observable.<PGNotification>create(subscriber -> {
```

```

try {
    Connection connection = DriverManager
        .getConnection("jdbc:postgresql:db");
    subscriber.add(Subscriptions.create(() ->
        closeQuietly(connection)));
    listenOn(connection, channel);
    Jdbc4Connection pgConn = (Jdbc4Connection) connection;
    pollForNotifications(pollingPeriod, pgConn)
        .subscribe(Subscribers.wrap(subscriber);
    } catch (Exception e) {
        subscriber.onError(e);
    }
}).share();
}

void listenOn(Connection connection, String channel) throws SQLException {
    try (Statement statement = connection.createStatement()) {
        statement.execute("LISTEN " + channel);
    }
}

void closeQuietly(Connection connection) {
    try {
        connection.close();
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

Удивительно, насколько короче мог бы быть пример, если бы не эти `SQLException`. Но ничего. Наша цель – разработать надежный тип `Observable<PGNotification>`. Прежде всего, мы откладываем открытие соединения с базой данных до тех пор, пока не появится подписчик. А чтобы избежать утечки соединений (серьезная проблема в любом приложении, напрямую работающем с JDBC), мы гарантируем закрытие соединения в момент отписки подписчика. Кроме того, если в потоке встречается ошибка, то происходит отписка, а, значит, и закрытие соединения.

Теперь мы готовы вызвать метод `listenOn()` начать получение уведомлений по открытому соединению. Если во время выполнения команды произойдет исключение, оно будет перехвачено и обработано путем вызова `subscriber.onError(e)`. В результате, во-первых, ошибка доходит до подписчика, а, во-вторых, закрывается соединение. Если же `LISTEN` завершается успешно, то при следующем обращении к `getNotifications()` мы получим все случившиеся после этого момента события.

Мы не хотим блокировать какие-либо потоки, поэтому создаем в методе `pollForNotifications()` внутренний объект `Observable` и вызываем для него оператор `interval()`. На этот объект подписывается тот же самый `Subscriber`, но обернутый методом `Subscribers.wrap()`, чтобы для одного подписчика не вызывался дважды метод `onStart()`.

```

Observable<PGNotification> pollForNotifications(
    long pollingPeriod,

```

```

    AbstractJdbc2Connection pgConn) {
return Observable
    .interval(0, pollingPeriod, TimeUnit.MILLISECONDS)
    .flatMap(x -> tryGetNotification(pgConn))
    .filter(arr -> arr != null)
    .flatMapIterable(Arrays::asList);
}

Observable<PGNotification[]> tryGetNotification(
    AbstractJdbc2Connection pgConn) {
    try {
        return Observable.just(pgConn.getNotifications());
    } catch (SQLException e) {
        return Observable.error(e);
    }
}

```

Периодически мы проверяем результат `getNotifications()`, предварительно обернув его странно выглядящим типом `Observable<PGNotification[]>`. Поскольку возвращенный массив `PGNotification[]` может быть равен `null`, мы отфильтровываем значения `null` оператором `filter()`, а затем с помощью `flatMapIterable()` убираем обертку, предварительно преобразовав массив в `List<PGNotification>` методом `Arrays::asList`. Я призываю вас тщательно проанализировать все эти шаги, следя за типами промежуточных объектов `Observable`. Единственная причина включения `closeQuietly()` и `tryGetNotification()` — обработка контролируемого исключения `SQLException`. Отметим, что в методе `closeQuietly()` мы проглатываем это исключение, потому что он вызывается в контексте, где сделать все равно ничего нельзя, например, когда клиент только что отписался и переправить исключение дальше нет возможности.

И последняя деталь реализации — вызовы `publish()` и `refCount()` в конце первого метода. Эти операторы позволяют разделить одно JDBC-соединение между несколькими подписчиками. Без них каждый новый подписчик открывал бы и прослушивал новое соединение, что весьма расточительно. Дополнительно `refCount()` отслеживает количество подписчиков и, когда последний отписывается, производит физическое закрытие соединения с базой. В разделе «Реализация единственной подписки с помощью `publish().refCount()`» главы 2 приведены детали операторов `publish()` и `refCount()`, в частности, как они изменяют поведение лямбда-выражения, переданного методу `Observable.create()`.

Памятуя, что по одному соединению можно прослушивать несколько каналов, в качестве упражнения попробуйте реализовать `observe()`, так чтобы он повторно использовал соединение, разделяя его между всеми подписчиками и всеми каналами, в которых они заинтересованы. Имеющаяся реализация разделяет соединение, если вызвать `observe()` один раз и подписаться несколько раз, тогда как можно было легко использовать одно соединение повторно, даже для подписчиков, интересующихся разными каналами.

На практике нет никаких причин использовать команды `LISTEN` и `NOTIFY` в PostgreSQL; на рынке имеются более быстрые и надежные очереди сообщений.

Но благодаря этому примеру мы поняли, как адаптировать JDBC к реактивному применению, несмотря на блокирование и опрос.

CompletableFuture и потоки

Помимо лямбда-выражений, нового API `java.time` и ряда более мелких добавлений, Java 8 принесла нам класс `CompletableFuture`. Он позволил заметно улучшить интерфейс `Future`, известный с версии Java 5. Чистые объекты `Future` представляют асинхронные операции, выполняемые в фоновом режиме, обычно они поступают от `ExecutorService`. Но API класса `Future` слишком примитивный, он заставляет разработчиков блокировать выполнение путем вызова метода `Future.get()`. Невозможно эффективно дождаться завершения первого объекта `Future`, не прибегая к активному ожиданию. Никаких других способов композиции объектов `Future` не предлагается. В следующем разделе мы кратко опишем, как работает класс `CompletableFuture`. А затем реализуем интероперабельность `CompletableFuture` и `Observable`.

Краткое введение в *CompletableFuture*

Класс `CompletableFuture` успешно закрывает этот пробел, предоставляя десятки полезных методов, почти все неблокирующие и допускающие композицию. Мы привыкли к тому, что `map()` асинхронно преобразует входные события на лету. А `Observable.flatMap()` позволяет заменить одно событие объектом `Observable`, сцепляющим асинхронные задачи. Похожая операция возможна и с объектами `CompletableFuture`. Представьте себе службу, которой нужны два несвязанных между собой элемента информации: `User` и `GeoLocation`. Зная тот и другой, мы просим несколько независимых турагентств найти рейс `flight` и заказываем билет `Ticket` у поставщика, который вернет ответ первым, — поощряя тем самым за скорость и реактивность. Последнее требование реализовать особенно трудно и до выхода Java 8 для эффективного поиска самой быстрой задачи приходилось использовать класс `ExecutorCompletionService`:

```
User findById(long id) {
    //...
}

GeoLocation locate() {
    //...
}

Ticket book(Flight flight) {
    //...
}

interface TravelAgency {
    Flight search(User user, GeoLocation location);
}
```

А вот как все это собиралось воедино:

```
ExecutorService pool = Executors.newFixedThreadPool(10);
List<TravelAgency> agencies = //...
User user = findById(id);
GeoLocation location = locate();
ExecutorCompletionService<Flight> ecs =
    new ExecutorCompletionService<>(pool);
agencies.forEach(agency ->
    ecs.submit(() ->
        agency.search(user, location)));
Future<Flight> firstFlight = ecs.poll(5, SECONDS);
Flight flight = firstFlight.get();
book(flight);
```

Класс `ExecutorCompletionService` не завоевал популярность у разработчиков на Java, а с появлением `CompletableFuture` стал и вовсе не нужен. Но сначала обратите внимание, как мы обернули `ExecutorService` объектом `ExecutorCompletionService`, чтобы потом можно было вызвать метод `poll` для получения завершившихся задач по мере поступления. Применение простого `ExecutorService` дало бы нам кучу объектов `Future` и никакой информации о том, какой завершится первым, так что класс `ExecutorCompletionService` оказался полезным. И все же нам пришлось пожертвовать одним потоком, чтобы с блокированием ждать ответа от агентств. К тому же, мы не задействуем конкурентность там, где это возможно (для одновременной загрузки `User` и `GeoLocation`).

Цель рефакторинга – преобразовать все методы в асинхронные и затем нужным образом комбинировать объекты `CompletableFuture`. Таким образом, код становится неблокирующим (главный поток завершается практически сразу), и мы можем максимально распараллелить его:

```
CompletableFuture<User> findByIdAsync(long id) {
    return CompletableFuture.supplyAsync(() -> findById(id));
}

CompletableFuture<GeoLocation> locateAsync() {
    return CompletableFuture.supplyAsync(this::locate);
}

CompletableFuture<Ticket> bookAsync(Flight flight) {
    return CompletableFuture.supplyAsync(() -> book(flight));
}

@Override
public CompletableFuture<Flight> searchAsync(User user, GeoLocation location) {
    return CompletableFuture.supplyAsync(() -> search(user, location));
}
```

Мы просто обернули блокирующие методы асинхронными объектами `CompletableFuture`. Метод `supplyAsync()` принимает в качестве необязательного аргумента объект `Executor`. Если он не задан, используется глобальный обь-

ект, определенный в `ForkJoinPool.commonPool()`. Вообще, рекомендуется задавать свой `Executor`, но в демонстрационном примере мы воспользовались исполнителем по умолчанию. Просто имейте в виду, что он разделяется между всеми объектами `CompletableFuture`, параллельными потоками (см. раздел «Потоки Java 8 и `CompletableFuture`» главы 8) и еще в нескольких не столь очевидных местах.

```
import static java.util.function.Function.identity;

List<TravelAgency> agencies = //...
CompletableFuture<User> user = findByIdAsync(id);
CompletableFuture<GeoLocation> location = locateAsync();

CompletableFuture<Ticket> ticketFuture = user
    .thenCombine(location, (User us, GeoLocation loc) -> agencies
        .stream()
        .map(agency -> agency.searchAsync(us, loc))
        .reduce((f1, f2) ->
            f1.applyToEither(f2, identity())
        )
        .get()
    )
    .thenCompose(identity())
    .thenCompose(this::bookAsync);
```

Это весьма содержательный пример. Объяснение устройства класса `CompletableFuture` во всей полноте выходит за рамки этой книги, но некоторые части API полезны в контексте RxJava. Сначала мы начинаем асинхронное получение `User` и `GeoLocation`. Эти операции независимы и могут выполняться одновременно. Но для продолжения работы нам нужны результаты обеих, естественно, без блокирования и расходования потока клиента. Именно это и делает метод `thenCombine()` — принимает два объекта `CompletableFuture` (`user` и `location`) и вызывает указанную функцию, когда оба завершатся — асинхронно. Интересно, что функция обратного вызова может вернуть значение, которое станет новым содержимым результирующего объекта `CompletableFuture`:

```
CompletableFuture<Long> timeFuture = //...
CompletableFuture<ZoneId> zoneFuture = //...

CompletableFuture<Instant> instantFuture = timeFuture
    .thenApply(time -> Instant.ofEpochMilli(time));

CompletableFuture<ZonedDateTime> zdtFuture = instantFuture
    .thenCombine(zoneFuture, (instant, zoneId) ->
        ZonedDateTime.ofInstant(instant, zoneId));
```

Класс `CompletableFuture` имеет много общего с `Observable`. Метод `thenApply()` на лету выполняет преобразование того, что находится внутри `Future`, — точно так же, как `Observable.map()`. В нашем примере `CompletableFuture<Long>` преобразуется в `CompletableFuture<Instant>` посредством функции `Instant::ofEpochMilli`, отображающей `Long` в `Instant`. Затем мы берем два объекта `Future` (`instantFuture`

и `zoneFuture`) и передаем их будущие значения, `instant` и `zoneId`, методу `thenCombine()`. Это преобразование возвращает значение типа `ZoneDateTime`, но, поскольку большинство методов класса `CompletableFuture` неблокирующие, в ответ мы получаем `CompletableFuture<ZonedDateTime>` — очень похоже на оператор `zip()` типа `Observable`.

Но вернемся к примеру заказа билетов. Следующий фрагмент кода, наверное, выглядит загадочно:

```
List<TravelAgency> agencies = //...

agencies
    .stream()
    .map(agency -> agency.searchAsync(us, loc))
    .reduce((f1, f2) ->
        f1.applyToEither(f2, identity()))
    )
    .get()
```

Нам нужно начать асинхронную операцию для каждого агентства `TravelAgency`, вызвав метод `searchAsync()`. В ответ мы немедленно получаем список `List<CompletableFuture<Flight>>`; эта структура очень неудобна, если нам нужно всего лишь получить первый завершившийся объект `Future`. Существуют методы `CompletableFuture.allOf()` и `CompletableFuture.anyOf()`. Семантически последний делает именно то, что требуется: принимает группу объектов `CompletableFuture` и возвращает тот, который завершился первым, отбрасывая все остальные. Это очень похоже на оператор `Observable.amb()` (см. раздел «Когда потоки не синхронизированы: `combineLatest()`, `withLatestFrom()` и `amb()`» главы 3). К сожалению, синтаксис метода `anyOf()` крайне неудобен. Прежде всего, он принимает массив переменной длины (`varargs`) и всегда возвращает объект типа `CompletableFuture<Object>`, а не того типа, который был указан в исходных объектах `Future`, например `Flight`. Мы можем воспользоваться этим методом, но получается некрасиво:

```
.thenCombine(location, (User us, GeoLocation loc) -> {
    List<CompletableFuture<Flight>> fs = agencies
        .stream()
        .map(agency -> agency.searchAsync(us, loc))
        .collect(toList());
    CompletableFuture[] futuresArr = new CompletableFuture[fs.size()];
    fs.toArray(futuresArr);
    return CompletableFuture
        .anyOf(futuresArr)
        .thenApply(x -> ((Flight) x));
})
```

Опишем прием, основанный на методе `Stream.reduce()`. Существует оператор `CompletableFuture.applyToEither()`, который принимает два объекта `CompletableFuture` и применяет заданное преобразование к тому, который завершится первым. Он чрезвычайно полезен, когда имеются две однородные задачи и нас интересует только первая завершившаяся. В примере ниже мы запрашиваем

User у двух разных серверов: основного и вспомогательного. Получив первый ответ – неважно, от кого, – мы применяем к нему преобразование, которое извлекает дату рождения пользователя. Второй объект `CompletableFuture` не прерывается, но его результат отбрасывается. Понятно, что в итоге мы получаем значение типа `CompletableFuture<LocalDate>`:

```
CompletableFuture<User> primaryFuture = //...
CompletableFuture<User> secondaryFuture = //...

CompletableFuture<LocalDate> ageFuture =
    primaryFuture
        .applyToEither(secondaryFuture,
            user -> user.getBirth());
```

Метод `applyToEither()` может работать только с двумя `CompletableFuture`, тогда как дурацкий `anyOf()` принимает любое число объектов. По счастью, мы можем вызвать `applyToEither()` для первых двух `Future`, взять результат (самый быстрый из первых двух) и применить тот же метод к нему и третьему `Future` (получив самый быстрый из первых трех). Продолжая действовать таким образом, мы получим объект `CompletableFuture`, завершившийся быстрее всех. Этот полезный прием можно эффективно реализовать, применив оператор `reduce()`. И последняя хитрость – метод `identity()` из класса `Function`. Это требование метода `applyToEither()`: мы должны предоставить преобразование, которое применяется к первому полученному результату. Если его не нужно никак преобразовывать, то указываем тождественную функцию, которую можно записать также в виде `f -> f` или `(Flight f) -> f`.

Наконец, мы реализовали объект `CompletableFuture<Flight>`, который завершится, когда ответит самое быстрое агентство `TravelAgency`, – асинхронно. Остался еще один мелкий вопрос, относящийся к результату `thenCombine()`. Любое значение, возвращенное преобразованием, которое было передано методу `thenCombine()`, обертывается объектом `CompletableFuture`. В нашем случае преобразование возвращает `CompletableFuture<Flight>`, поэтому тип результата `thenCombine()` такой: `CompletableFuture<CompletableFuture<Flight>>`. Двойное обертывание – обычное дело и в случае `Observable`, поэтому, чтобы избавиться от него, применяется тот же прием: `flatMap()`! (см раздел «Обертывание с помощью `flatMap()`» главы 3). Но помните о различии имен: если оператору `map()` соответствует метод `thenApply()`, то оператору `flatMap()` – метод `thenCompose()`:

```
Observable<Observable<String>> badStream = //...
Observable<String> goodStream = badStream.flatMap(x -> x);

CompletableFuture<CompletableFuture<String>> badFuture = //...
CompletableFuture<String> goodFuture = badFuture.thenCompose(x -> x);
```

Обычно оператор `flatMap()/thenCompose()` применяется для сцепления асинхронных вычислений, но здесь мы просто убираем ненужный обертывающий тип. Имейте в виду, что `thenCompose()` ожидает, что переданное ему преобразование

возвращает значение типа `CompletableFuture`. Но поскольку внутренний тип уже `Future`, то задание функции `identity()` или просто `x -> x` исправляет тип, снимая обертку с внутреннего `Future`.

Наконец, имея объект типа `CompletableFuture<Flight> (flightFuture)`, мы вызываем метод `bookAsync()`, принимающий значение типа `Flight` в качестве аргумента:

На этот раз для вызова `bookAsync()` более естественно применить `thenCompose()`. Этот метод возвращает значение типа `CompletableFuture<Ticket>`, поэтому во избежание двойного обертывания мы выбрали `thenCompose()`, а не `thenApply()`.

Интероперабельность с `CompletableFuture`

Фабричный метод `Observable.from(Future<T>)`, возвращающий `Observable<T>`, уже существует. Но из-за ограничений старого API `Future<T>` у него есть ряд недостатков, самый серьезный из которых – внутреннее блокирование из-за вызова `Future.get()`. В классической реализации типа `Future<T>` нет способа регистрировать обратные вызовы и обрабатывать их асинхронно, поэтому в реактивных приложениях он бесполезен.

А вот `CompletableFuture` – совсем другая история. Семантически можно рассматривать `CompletableFuture` как `Observable` со следующими характеристиками:

Горячий.

Вычисление, указанное в `CompletableFuture`, начинается энергично, вне зависимости от того, зарегистрированы какие-нибудь обратные вызовы типа `thenApply()` или нет.

Кэшируется.

Фоновое вычисление, указанное в `CompletableFuture`, энергично производится только один раз, и результат отправляется всем зарегистрированным обратным вызовам. Более того, если обратный вызов зарегистрирован уже после его завершения, то он вызывается немедленно, получая вычисленное значение (или исключение).

Порождает ровно один элемент или исключение.

По сути своей, `Future<T>` завершается только один раз (или никогда) и возвращает значение типа `T` или исключение. Это соответствует контракту `Observable`.

Преобразование `CompletableFuture` в `Observable` с одним элементом

Сначала напомним служебную функцию, которая принимает значение типа `CompletableFuture<T>` и возвращает объект `Observable<T>` с корректным поведением:

```
class Util {  
    static <T> Observable<T> observe(CompletableFuture<T> future) {
```

```

return Observable.create(subscriber -> {
    future.whenComplete((value, exception) -> {
        if (exception != null) {
            subscriber.onError(exception);
        } else {
            subscriber.onNext(value);
            subscriber.onCompleted();
        }
    });
});
}
}

```

Чтобы получать уведомления об успешном и неудачном завершении, мы воспользовались методом `CompletableFuture.whenComplete()`. Он получает два параметра. Если `exception` не равно `null`, значит, объект `Future` завершился с ошибкой. В противном случае завершение было успешным, и объект содержит значение `value`. В обоих случаях мы уведомляем переданного подписчика `subscriber`. Отметим, что если подписка оформлена после завершения `CompletableFuture` (все равно, успешного или нет), функции обратного вызова вызываются немедленно. `CompletableFuture` кэширует результат сразу после завершения, поэтому функции, зарегистрированные позже, можно вызывать синхронно в вызывающем потоке.

Возникает искушение зарегистрировать обработчик отписки, который пытается отменить `CompletableFuture` в случае отписки:

```

// Не делайте так!
subscriber.add(Subscriptions.create(
    () -> future.cancel(true)));

```

Это неудачная мысль. Мы можем создать много объектов `Observable` на базе одного и того же `CompletableFuture`, и у каждого из них может быть много подписчиков. Если отписаться до завершения `Future` решит всего один подписчик, то отмена окажет влияние на всех остальных.

Помните, что `CompletableFuture` является *горячим* и *кэшируемым* в терминологии Rx. Он начинает вычисление немедленно, тогда как `Observable` откладывает это до появления первого подписчика. Имея это в виду, мы можем еще улучшить наш API с помощью следующих тривиальных функций:

```

Observable<User> rxFindById(long id) {
    return Util.observe(findByIdAsync(id));
}

Observable<GeoLocation> rxLocate() {
    return Util.observe(locateAsync());
}

Observable<Ticket> rxBook(Flight flight) {
    return Util.observe(bookAsync(flight));
}

```

Понятно, что если бы исходный API с самого начала поддерживал `Observable`, то все эти многослойные адаптеры не понадобились бы. Но коль скоро в нашем распоряжении только объекты типа `CompletableFuture`, то преобразовать их в `Observable` можно эффективно и безопасно. Преимущество RxJava в том, что исходную задачу можно решить гораздо лаконичнее:

```
Observable<TravelAgency> agencies = agencies();
Observable<User> user = rxFindById(id);
Observable<GeoLocation> location = rxLocate();

Observable<Ticket> ticket = user
    .zipWith(location, (usr, loc) ->
        agencies
            .flatMap(agency -> agency.rxSearch(usr, loc))
            .first()
        )
    .flatMap(x -> x)
    .flatMap(this::rxBook);
```

Клиентский код на основе RxJava API содержит меньше «шума» и читать его проще. Rx естественным образом поддерживает «будущие объекты с несколькими значениями» в форме потоков. Если тождественные преобразования вида `x -> x` внутри `flatMap()` все еще пугают вас, то всегда можно расщепить поток `zipWith()` с помощью вспомогательного контейнера `Pair`:

```
import org.apache.commons.lang3.tuple.Pair;

//...
Observable<Ticket> ticket = user
    .zipWith(location, (usr, loc) -> Pair.of(usr, loc))
    .flatMap(pair -> agencies
        .flatMap(agency -> {
            User usr = pair.getLeft();
            GeoLocation loc = pair.getRight();
            return agency.rxSearch(usr, loc);
        })
    .first()
    .flatMap(this::rxBook);
```

Сейчас вы уже должны понимать, почему дополнительная функция `x -> x` больше не нужна. Оператор `zipWith()` принимает два независимых объекта `Observable` и асинхронно ждет событий от обоих. В Java нет встроенных классов для пар и кортежей, поэтому мы должны предоставить преобразование, которое принимает события из двух потоков и объединяет их в один объект `Observable<Pair<User, Location>>`. Этот объект подается на вход следующего `Observable`. Затем мы используем `flatMap()` для конкурентного поиска по `User` и `Location` во всех турагентствах. `flatMap()` разворачивает обертку (с точки зрения синтаксиса), поэтому в результате получается просто объект типа `Observable<Flight>`. Естественно, в обоих случаях вызывается `first()`, чтобы обрабатывать только первый рейс `flight` из входного потока (от самого быстрого агентства).

От Observable к CompletableFuture

Иногда можно встретить API, который поддерживает тип `CompletableFuture`, но не `RxJava`. И это довольно распространенная ситуация, учитывая, что первый тип – часть JDK, а второй входит в стороннюю библиотеку. В таких случаях было бы желательно преобразовать `Observable` в `CompletableFuture`. Это можно сделать двумя способами.

`Observable<T> В CompletableFuture<T>`

Пользуйтесь этим способом, когда ожидаете, что во входном потоке будет только один элемент, например, когда `Rx` обортывает вызов метода или в случае паттерна «запрос-ответ». Объект `CompletableFuture<T>` завершается успешно, когда поток завершается и содержит ровно одно значение. Исключением – если так завершился поток или если поток содержит больше одного значения.

`Observable<T> В CompletableFuture<List<T>>`

В этом случае `CompletableFuture` завершается, когда порождены все события в `Observable` и этот поток завершился. Как мы увидим ниже, это просто частный случай первого преобразования.

Первый вариант можно реализовать так:

```
static <T> CompletableFuture<T> toFuture(Observable<T> observable) {
    CompletableFuture<T> promise = new CompletableFuture<>();
    observable
        .single()
        .subscribe(
            promise::complete,
            promise::completeExceptionally
        );
    return promise;
}
```

Прежде чем объяснять код, подчеркну, что у этого преобразования есть важный побочный эффект: оно подписывается на `Observable` и тем самым инициирует вычисление холодных объектов `Observable`. Кроме того, каждый вызов этого метода производит подписку заново; об этом проектом решении следует помнить.

Но сама реализация весьма интересна. Сначала мы заставляем `Observable` породить ровно одно событие, вызывая оператор `single()`; в противном случае возбуждается исключение. Если событие порождено и поток завершился, то вызывается метод `CompletableFuture.complete()`. Оказывается, что создать объект `CompletableFuture` с чистого листа можно без поддерживающего пула потоков и асинхронной задачи. Это самый настоящий `CompletableFuture`, но единственный способ завершить его и уведомить все зарегистрированные функции обратного вызова – вызвать метод `complete()` явно. Это эффективный способ асинхронного обмена данными, по крайней мере, в случае, когда `RxJava` недоступна.

В случае ошибки мы можем уведомить о ней все зарегистрированные функции обратного вызова, вызвав метод `CompletableFuture.completeExceptionally()`.

Как ни удивительно, это и есть полная реализация. Объект `Future`, возвращенный из `toFuture()`, ведет себя так, будто с ним связана какая-то фоновая задача, хотя в действительности мы завершаем его явно. Преобразование из `Observable<T>` в `CompletableFuture<List<T>>` до смешного простое:

```
static <T> CompletableFuture<List<T>> toFutureList(Observable<T> observable) {  
    return toFuture(observable.toList());  
}
```

Интероперабельность типов `CompletableFuture` и `Observable` весьма полезна. Первый правильно спроектирован, но ему не хватает выразительности и богатства второго. Поэтому, если вы вынуждены работать с типом `CompletableFuture` в приложении, основанном на `RxJava`, то применяйте эти простые преобразования на как можно более ранней стадии, чтобы API был единообразным и предсказуемым. И не забывайте о различии между энергичным (горячим) `Future` и `Observable`, по умолчанию ленивым.

Сравнение типов `Observable` и `Single`

Я часто вижу, что люди боятся библиотеки `RxJava` из-за ее потоковой природы. Объект `Observable` — это поток, потенциально бесконечный, и все операторы также описываются в терминах потоков. Но, как и список `List<T>`, который может состоять из одного элемента, некоторые объекты `Observable<T>`, по определению, порождают ровно одно событие. Странно было бы иметь список, который всегда содержит только один элемент, поэтому в таких случаях мы используем просто тип `T` или `Optional<T>`. А в `RxJava` существует специальная абстракция для `Observable`, порождающих ровно один элемент, она называется `rx.Single<T>`.

Тип `single<T>`, по существу, представляет собой контейнер для будущего значения типа `T` или `Exception`. В этом смысле класс `CompletableFuture` из `Java 8` — ближайший родственник `single` (см. раздел «`CompletableFuture` и потоки» выше). Но в отличие от `CompletableFuture`, тип `single` ленивый и не начинает порождать значения, пока кто-то не подпишется. Обычно `single` используется в API, которые заведомо возвращают только одно значение асинхронно и с высокой вероятностью ошибки. Очевидно, что `single` — прекрасный кандидат, когда речь идет о взаимодействии типа «запрос-ответ», как в случае обращений по сети. Задержка высока по сравнению с обычным вызовом метода, а ошибки неизбежны. А поскольку тип `single` ленивый и асинхронный, мы можем применять всевозможные приемы для уменьшения задержки и повышения надежности, например, выполнять независимые операции одновременно и объединять ответы (см. раздел «Объединение ответов с помощью `zip`, `merge` и `concat`» ниже). Тип `single` устраняет неоднозначность API, позволяя уточнить тип возвращаемого `Observable`. Взгляните на следующее объявление:

```
Observable<Float> temperature() {  
    //...  
}
```

Трудно сказать что-то определенное о контракте этого метода. Возвращает ли он только один результат измерения температуры и потом завершается? Или бесконечный поток значений температуры? Хуже того, при некоторых условиях он даже может завершиться, не вернув ни одного события. А если бы метод `temperature()` возвращал значение типа `Single<Float>`, то мы сразу бы поняли, чего ожидать.

Создание и потребление объектов типа Single

Тип `Single` похож на `Observable` в том, что касается поддерживаемых операторов, поэтому не будем тратить на них много времени. А просто сравним поведение одноименных операторов обоих типов и сосредоточим внимание на ситуациях, когда `Single` полезен. Существует несколько способов создать объект `Single`, начиная с операторов `just()` и `error()`:

```
import rx.Single;

Single<String> single = Single.just("Hello, world!");
single.subscribe(System.out::println);

Single<Instant> error =
    Single.error(new RuntimeException("Oops!"));
error
    .observeOn(Schedulers.io())
    .subscribe(
        System.out::println,
        Throwable::printStackTrace
    );
```

У оператора `just()` нет перегруженных вариантов, принимающих несколько значений, — в конце концов, объект типа `Single`, по определению, может хранить только одно значение. Кроме того, метод `subscribe()` принимает два аргумента, а не три. В обратном вызове `onComplete()` просто нет смысла, поскольку `Single` завершается, либо породив одно значение (первый обратный вызов), либо в ошибкой (второй обратный вызов). Реакция на сам факт завершения эквивалентна подписке на единственное значение. Заодно мы включили оператор `observeOn()`, который работает в точности также, как его аналог в `Observable`. То же самое относится к методу `subscribeOn()` (см. раздел «Императивная конкурентность» главы 4). Наконец, оператор `error()` позволяет создать объект `Single`, который всегда завершается с заданным исключением `Exception`.

Давайте реализуем более жизненный сценарий — отправку HTTP-запроса. В разделе «Неблокирующий клиент на основе RxNetty» выше мы научились строить асинхронные HTTP-клиенты с применением RxNetty. На этот раз воспользуемся библиотекой `async-http-client` (<https://github.com/AsyncHttpClient/async-http-client>), которая тоже основана на Netty. Вместе с отправкой HTTP-запроса мы должны указать функцию обратного вызова, которую следует асинхронно вызвать, когда придет ответ или произойдет ошибка. Это отлично согласуется со способом создания объекта `Single`:

```

import com.ning.http.client.AsyncCompletionHandler;
import com.ning.http.client.AsyncHttpClient;
import com.ning.http.client.Response;

AsyncHttpClient asyncHttpClient = new AsyncHttpClient();

Single<Response> fetch(String address) {
    return Single.create(subscriber ->
        asyncHttpClient
            .prepareGet(address)
            .execute(handler(subscriber)))
}

AsyncCompletionHandler handler(SingleSubscriber<? super Response> subscriber) {
    return new AsyncCompletionHandler() {
        public Response onCompleted(Response response) {
            subscriber.onSuccess(response);
            return response;
        }

        public void onThrowable(Throwable t) {
            subscriber.onError(t);
        }
    };
}

```

Метод `Single.create()`, на первый взгляд, похож на `Observable.create()`, но имеет важные ограничения; разрешается вызвать либо `onSuccess()` единожды, либо `onError()` — тоже единожды. Технически можно создать объект `Single`, который вообще не завершается, но многократный вызов `onSuccess()` запрещен. Помимо метода `Single.create()`, создать объект можно методом `Single.fromCallable()`, который принимает `Callable<T>` и возвращает `Single<T>`. Вот так все просто.

Но вернемся к нашему HTTP-клиенту. Получив ответ, мы уведомляем подписчиков, вызывая метод `onSuccess()`, или распространяем асинхронное исключение методом `onError()`. Используется `Single` примерно так же, как `Observable`:

```

Single<String> example =
    fetch("http://www.example.com")
        .flatMap(this::body);

String b = example.toBlocking().value();

//...

Single<String> body(Response response) {
    return Single.create(subscriber -> {
        try {
            subscriber.onSuccess(response.getResponseBody());
        } catch (IOException e) {
            subscriber.onError(e);
        }
    });
}

```

```

}

// Функциональность та же, что в body():
Single<String> body2(Response response) {
    return Single.fromCallable(() ->
        response.getResponseBody());
}

```

К сожалению, метод `Response.getResponseBody()` может возбуждать исключение `IOException`, поэтому мы не вправе написать просто `map(Response::getResponseBody)`. Но зато мы видим, как работает `Single.flatMap()`. Обернув потенциально опасный метод `getResponseBody()` объектом `Single<String>`, мы гарантируем, что возможная ошибка инкапсулирована и ясно обозначена средствами системы типов. Зная о методе `Observable.flatMap()`, мы легко сообразим, как работает `Single.flatMap()`: если вторая стадия вычисления (в данном случае `this::body`) завершается ошибкой, то ошибкой завершается и весь объект `Single`. Интересно, что в классе `Single` есть методы `map()` и `flatMap()`, но отсутствует `filter()`. Понимаете, почему? Метод `filter()` теоретически мог бы отфильтровать содержимое `Single<T>`, если оно не удовлетворяет заданному предикату `Predicate<T>`. Но `Single<T>` *должен* содержать ровно один элемент, а применение `filter()` могло бы породить `Single` вообще без элементов.

По аналогии с `BlockingObservable` (см. раздел «`BlockingObservable`: выход из реактивного мира» главы 4) у класса `Single` есть собственный класс `BlockingSingle`, экземпляры которого создаются методом `Single.toBlocking()`. Как и раньше, сам факт создания объекта `BlockingSingle<T>` еще не приводит к блокированию. Но вызов его метода `value()` блокирует выполнение, пока не будет получено значение типа `T` (в нашем примере – тело ответа типа `String`). Исключения повторно возбуждаются методом `value()`.

Объединение ответов с помощью `zip`, `merge` и `concat`

Класс `rx.Single` был бы бесполезен без допускающих композицию операторов. Самый важный из них – оператор `Single.zip()`, работающий так же, как `Observable.zip()` (см. раздел «Попарная композиция с помощью `zip()` и `zipWith()`» главы 3), но с более простой семантикой. `Single` всегда порождает единственное значение (или исключение), поэтому результатом `Single.zip()` (или метода экземпляра `Single.zipWith()`) всегда является одна пара или кортеж. Оператор `zip()` по сути дела представляет собой способ создания третьего объекта `Single`, когда оба исходных завершатся⁴.

Предположим, что мы хотим отобразить на странице сайта статью. Для выполнения запроса нужно выполнить три независимые операции: прочитать содержимое статьи из базы данных, запросить у сайта социальной сети, сколько она набрала лайков, и обновить счетчик прочтений. Наивная реализация не только выполняет все три действия последовательно, но и рискует получить неприемле-

⁴ В классе `CompletableFuture` этот оператор называется `thenCombine()`.

мо большую задержку, если какой-то шаг окажется медленным. С применением `Single` каждый шаг моделируется отдельно:

```
import org.springframework.jdbc.core.JdbcTemplate;

//...

Single<String> content(int id) {
    return Single.fromCallable(() -> jdbcTemplate
        .queryForObject(
            "SELECT content FROM articles WHERE id = ?",
            String.class, id))
        .subscribeOn(Schedulers.io());
}

Single<Integer> likes(int id) {
    // асинхронный HTTP-запрос сайте социальной сети
}

Single<Void> updateReadCount() {
    // интересен только побочный эффект, никакое значение не возвращается
}
```

В качестве примера мы показали, как создать объект `Single` методом `fromCallable`, передав лямбда-выражение. Это полезное средство, оно обрабатывает за нас ошибки (см. раздел «А где же мои исключения?» главы 7). В методе `content()` используется класс `JdbcTemplate` из каркаса `Spring`, который ненавязчиво загружает статью из базы данных. `JDBC API` блокирующий, поэтому мы явно вызываем оператор `subscribeOn()`, чтобы сделать объект `Single` асинхронным. Реализации методов `likes()` и `updateReadCount()` мы опустили. Можно представить себе, что `likes()` отправляет асинхронный `HTTP`-запрос какому-то `API` средствами `RxNetty` (см. раздел «Неблокирующий `HTTP`-клиент на основе `RxNetty`» выше). Метод `updateReadCount()` интересен своим типом: `Single<Void>`. Видя его, можно сделать вывод, что он производит какой-то побочный эффект, не возвращающий никакого значения, но с заметной задержкой. Тем не менее, нам хотелось бы получать асинхронные уведомления о возможных ошибках. В `RxJava` для таких случаев есть специальный тип `Completable`. Он означает, что операция либо завершится без результирующего значения, либо асинхронно возбуждает исключение.

Объединить все три операции в одном `zip` не составляет труда:

```
Single<Document> doc = Single.zip(
    content(123),
    likes(123),
    updateReadCount(),
    (con, lks, vod) -> buildHtml(con, lks)
);

//...

Document buildHtml(String content, int likes) {
```

```
//...  
}
```

Оператор `single.zip()` принимает три объекта `single` (имеются перегруженные варианты с числом аргументов от двух до девяти) и вызывает переданную нами функцию, когда все три завершатся. Значение, возвращенное функцией, помещается в объект `Single<Document>`, который можно подвергнуть дальнейшим преобразованиям. Имейте в виду, что результат типа `void` никогда не используется в преобразованиях. Это означает, что мы ждем завершения `updateReadCount()`, но не нуждаемся в его результате (пустом). Это может быть требованием, но в принципе возможна оптимизация: построение HTML-документа будет работать и тогда, когда `updateReadCount()` выполняется асинхронно – ждать его завершения или ошибки совершенно необязательно.

Теперь подумаем, что произойдет, если вызов `likes()` завершится неудачно или займет недопустимо много времени (что гораздо хуже). Без реактивных расширений мы либо вообще не увидели бы HTML-разметки, либо ждали бы очень долго. Но и наша реализация в этом отношении не многим лучше. Класс `Single` поддерживает ряд операторов, например `timeout()`, `onErrorReturn()` и `onErrorResumeNext()`, которые повышают устойчивость к ошибкам. Все они ведут себя так же, как их аналоги в `Observable`.

Интероперабельность с Observable и CompletableFuture

С точки зрения системы типов, классы `Observable` и `Single` не связаны между собой. По существу, это означает, что вместо `Observable` нельзя использовать `Single` и наоборот. Но есть две ситуации, когда преобразование между ними имеет смысл:

- если мы используем `Single` как `Observable`, порождающий одно значение и уведомление о завершении (или уведомление об ошибке);
- если в классе `Single`, отсутствуют некоторые операторы, имеющиеся в `Observable`, например, `cache()`⁵.

Продemonстрируем на примере второй ситуации.

```
Single<String> single = Single.create(subscriber -> {  
    System.out.println("Подписка");  
    subscriber.onSuccess("42");  
});
```

```
Single<String> cachedSingle = single  
    .toObservable()  
    .cache()  
    .toSingle();
```

```
cachedSingle.subscribe(System.out::println);  
cachedSingle.subscribe(System.out::println);
```

⁵ На момент написания этой книги (в версии RxJava 1.1.6). Набор доступных операторов быстро расширяется, поэтому установите последнюю версию и проверьте еще раз.

Мы воспользовались здесь оператором `cache()`, так что `Single` генерирует значение "42" только один раз для первого подписчика. Оператор `Single.toObservable()` безопасен и прост для понимания. Он принимает экземпляр `Single<T>` и преобразует его в `Observable<T>`, порождая один элемент сразу после уведомления о завершении (или об ошибке, если этот объект `Single` завершился именно так). Противоположный оператор `Observable.toSingle()` (не путайте с оператором `single()`; см. раздел «Проверка того, что `Observable` содержит ровно один элемент, с помощью `single()`» главы 3) не столь очевиден. Как и `single()`, оператор `toSingle()` возбуждает исключение с сообщением «*Observable emitted too many elements*», если `Observable` порождает больше одного элемента. А если `Observable` пуст, то будет возбуждено исключение с сообщением «*Observable emitted no items*»:

```
Single<Integer> emptySingle =
    Observable.<Integer>empty().toSingle();
Single<Integer> doubleSingle =
    Observable.just(1, 2).toSingle();
```

Вы, наверное, думаете, что если операторы `toObservable()` и `toSingle()` стоят близко друг к другу, то второй безопасен, однако это необязательно. Например, промежуточный `Observable` может продублировать или отбросить событие, порожденное объектом `Single`:

```
Single<Integer> ignored = Single
    .just(1)
    .toObservable()
    .ignoreElements() // ПРОБЛЕМА
    .toSingle();
```

Здесь оператор `ignoreElements()` из класса `Observable` просто отбрасывает то единственное значение, которое породил `single`. Следовательно, когда дойдет дело до оператора `toSingle()`, он увидит лишь завершившийся объект `Observable` без единого элемента. Не забывайте, что оператор `toSingle()`, как и все остальные операторы, которые нам до сих пор встречались, ленивый. Исключение, сообщаемое, что `Single` породил не в точности одно событие, возникнет не раньше, чем кто-то подпишется.

Когда использовать тип `Single`?

Имея две абстракции, `Observable` и `Single`, важно четко различать их и понимать, когда какую использовать. Как всегда бывает со структурами данных, не существует решения на любой случай. Тип `single` следует применять в следующих ситуациях.

- Операция должна завершаться, возвращая единственное значение или исключение. Например, обращение к веб-сервису, который либо дает ответ от внешнего сервера, либо заканчивается каким-то исключением.
- К предметной области понятие *потока* неприменимо, в этом случае использование `Observable` только вводило бы в заблуждение и привело бы к ненужным накладным расходам.

- Тип `Observable` слишком тяжеловесный, и результаты измерений *доказали*, что в вашей конкретной задаче `Single` быстрее.
- С другой стороны, `Observable` следует предпочесть при следующих условиях:
 - моделируются события (сообщения, события пользовательского интерфейса), которые, по определению, могут возникать много раз, возможно, бесконечно много;
 - или наоборот – до завершения объекта событие может возникнуть, а может и не возникнуть.

Последний случай особенно интересен. Как вы думаете, имеет ли смысл определить метод поиска в некотором хранилище `findById(int)` как возвращающий `Single<Record>`, а не `Record` или `Observable<Record>`? На первый взгляд, логично: мы ищем элемент по его идентификатору (предполагая тем самым, что такая запись `Record` всего одна). Однако нет гарантии, что для каждого идентификатора существует запись. Следовательно, метод технически может не вернуть *ничего*, а это моделируется как `null`, `Optional<Record>` или `Observable<Record>`, причем последний тип вполне способен представить пустой поток, за которым следует уведомление о завершении. А как насчет `Single`? Такой объект должен либо вернуть единственное значение (`Record`), либо возбудить исключение. Вы, конечно, можете моделировать отсутствие записи исключением, но обычно это считается предосудительной практикой. Решение о том, является ли отсутствие записи с указанным идентификатором, действительно *исключительной* ситуацией, определенно принимается не на уровне хранения.

Резюме

В главах 2 и 3 мы привели общий обзор библиотеки `RxJava` и получили представление о том, как с ней работать. В этой главе мы рассмотрели более сложные вопросы, относящиеся к проектированию полностью реактивных приложений. Мы познакомились с практическими приемами реализации событийно-ориентированных систем, обходящихся без акцидентальной сложности. Были представлены результаты несколько тестов производительности, доказывающие превосходство `RxJava` в сочетании с неблокирующим сетевым стеком типа `Netty`. Вас, конечно, никто не заставляет использовать такие продвинутые библиотеки, но стремление выжать максимальную пропускную способность из серверов потребительского класса, безусловно, окупается.



Глава 6.

Управление потоком и противодействие

В предыдущих главах мы близко познакомились с основанной на проталкивании природой RxJava. События порождаются в начале потока и потребляются всеми подписчиками в конце. Мы не обращали особого внимания на то, что происходит, если подписчик работает настолько медленно, что не поспевает за событиями, порождаемыми внутри метода `Observable.create()`. Данная глава целиком посвящена этой проблеме.

В RxJava есть два способа управиться с производителями, работающими быстрее подписчиков:

- различные механизмы управления потоком (flow control), в т. ч. выборка и пакетирование, реализованные встроенными операторами;
- подписчики могут запрашивать столько элементов, сколько в состоянии обработать, используя канал обратной связи, который называется противодействием (backpressure).

В этой главе будут описаны оба механизма.

Управление потоком

До того как в RxJava началась реализация противодействия, иметь дело с производителями (объектами `Observable`), работающими быстрее потребителей (объектов `Observer`), было трудно. Есть довольно много операторов, придуманных специально для борьбы с производителями, отправляющими слишком много событий, и большинство из них интересны сами по себе. Одни собирают события в пакеты, другие отбрасывают некоторые события. В этом разделе мы дадим краткий обзор таких операторов с примерами.

Периодическая выборка и отбрасывание событий

Бывают ситуации, когда абсолютно необходимо получить и обработать каждое событие, порожденное объектом `Observable`. Но есть и такие, когда вполне доста-

точно периодической выборки. Самый очевидный пример – получение результатов измерений от некоторого устройства, например, от датчика температуры (см. раздел. «Устранение дубликатов с помощью `distinct()` и `distinctUntilChanged()`» главы 3). Частота измерений устройством зачастую не важна, особенно если измерения производятся часто, но мало отличаются. Оператор `sample()` периодически проверяет входной поток (например, раз в секунду) и отправляет дальше последнее полученное событие. Если в течение одной секунды не было событий, то ничего и не отправляется, а следующая выборка происходит секунду спустя. Это показано в примере ниже:

```
long startTime = System.currentTimeMillis();
Observable
    .interval(7, TimeUnit.MILLISECONDS)
    .timestamp()
    .sample(1, TimeUnit.SECONDS)
    .map(ts -> ts.getTimestampMillis() - startTime + "ms: " + ts.getValue())
    .take(5)
    .subscribe(System.out::println);
```

Этот код напечатает что-то вроде:

```
1088ms: 141
2089ms: 284
3090ms: 427
4084ms: 569
5085ms: 712
```

В первом столбце печатается время от начала подписки до момента выборки. Видно, что первое событие появляется спустя секунду с небольшим (как и просил оператор `sample()`), а последующие отстоят друг от друга примерно на секунду. Но гораздо интереснее сами значения. Оператор `interval()` порождает очередное натуральное число каждые 7 миллисекунд. Следовательно, к моменту первой выборки можно ожидать появления 142-го числа ($1000/7$), т. е. числа 141 (начиная с нуля).

Рассмотрим несколько более сложный пример. Пусть имеется список имен, которые отправляются в поток с заданными абсолютными задержками, например:

```
Observable<String> names = Observable
    .just("Mary", "Patricia", "Linda",
        "Barbara",
        "Elizabeth", "Jennifer", "Maria", "Susan",
        "Margaret", "Dorothy");

Observable<Long> absoluteDelayMillis = Observable
    .just(0.1, 0.6, 0.9,
        1.1,
        3.3, 3.4, 3.5, 3.6,
        4.4, 4.8)
    .map(d -> (long) (d * 1_000));

Observable<String> delayedNames = names
```

```

.zipWith(absoluteDelayMillis,
        (n, d) -> Observable
            .just(n)
            .delay(d, MILLISECONDS))
.flatMap(o -> o);

delayedNames
    .sample(1, SECONDS)
    .subscribe(System.out::println);

```

Сначала мы строим последовательность имен, а вслед за ней – последовательность абсолютных задержек (в секундах, которые затем преобразуются в миллисекунды). С помощью оператора `zipWith()` мы задерживаем (`delay()`) порождение имен: например, имя *Mary* появится спустя 100 мс с момента подписки, а имя *Dorothy* – спустя 4.8 секунд. Оператор `sample()` периодически (раз в секунду) выбирает из потока последнее имя, встретившееся за указанный в нем период. Так, после первой секунды мы напечатаем *Linda*, а спустя еще секунду – *Barbara*. В промежутке между 2000-й и 3000-й миллисекундой в потоке нет ни одного имени, поэтому `sample()` ничего не порождает. Через две секунды после порождения *Barbara* мы увидим имя *Susan*. Оператор `sample()` переправляет дальше уведомление о завершении (и об ошибке) и отбрасывает последний период. Если мы хотим видеть также имя *Dorothy*, то можем искусственно задержать отправку уведомления о завершении:

```

static <T> Observable<T> delayedCompletion() {
    return Observable.<T>empty().delay(1, SECONDS);
}

```

```
//...
```

```

delayedNames
    .concatWith(delayedCompletion())
    .sample(1, SECONDS)
    .subscribe(System.out::println);

```

У оператора `sample()` есть и другой вариант, принимающий в качестве аргумента не фиксированный период, а значение типа `Observable`. Этот второй `Observable` (его называют отборником) говорит, когда делать выборку из входного потока: всякий раз как отборник порождает какое-нибудь значение, из потока выбирается элемент (если, конечно, что-то появилось с момента последней выборки). Этот перегруженный вариант `sample()` можно применять для динамического изменения темпа выборки или для выборки элементов в строго определенные моменты времени. Например, можно запоминать некоторое значение в момент перерисовки фрейма или нажатия клавиши. Тривиальное применение – моделирование фиксированного периода с помощью оператора `interval()`:

```

// эти два оператора эквивалентны:
obs.sample(1, SECONDS);
obs.sample(Observable.interval(1, SECONDS));

```

Как видим, в поведении `sample()` имеются некоторые нюансы. Но лучше не полагаться на правильное понимание документации или на ручную проверку, а написать автоматизированные тесты. Тестирование зависящих от времени операторов типа `sample()` рассматривается в разделе «Виртуальное время» главы 7.

У оператора `sample()` имеется псевдоним `throttleLast()`. Есть и симметричный оператор `throttleFirst()`, который отдает только первое событие в каждом периоде. Применив к нашему потоку имен `throttleFirst()` вместо `sample()`, мы получим вполне ожидаемые результаты:

```
Observable<String> names = Observable
    .just("Mary", "Patricia", "Linda",
        "Barbara",
        "Elizabeth", "Jennifer", "Maria", "Susan",
        "Margaret", "Dorothy");

Observable<Long> absoluteDelayMillis = Observable
    .just(0.1, 0.6, 0.9,
        1.1,
        3.3, 3.4, 3.5, 3.6,
        4.4, 4.8)
    .map(d -> (long) (d * 1_000));

//...

delayedNames
    .throttleFirst(1, SECONDS)
    .subscribe(System.out::println);
```

Вот что будет напечатано:

```
Mary
Barbara
Elizabeth
Margaret
```

Как и `sample()` (он же `throttleLast()`), `throttleFirst()` ничего не передает дальше, если в течение периода не было ни одного события, как между именами *Barbara* и *Elizabeth*.

Буферизация событий в списке

Буферизация и скользящие окна – одни из самых удивительных операторов, встроенных в RxJava. Оба они пропускают входной поток через окно, которое запоминает несколько последовательных элементов и сдвигается вперед. С другой стороны, они позволяют собирать входные события в пакет для более эффективной обработки. На практике это гибкие и многогранные инструменты, позволяющие разными способами агрегировать данные на лету.

Оператор `buffer()` собирает пакеты событий в список `List` в режиме реального времени. Но в отличие от оператора `toList()`, `buffer()` порождает несколько списков, группируя в каждом заданное число последовательных событий, а не единст-

венный список, содержащий все события. В простейшей форме `buffer()` группирует события от предшествующего `Observable` в списки одинакового размера:

```
Observable
    .range(1, 7) //1, 2, 3, ... 7
    .buffer(3)
    .subscribe((List<Integer> list) -> {
        System.out.println(list);
    })
);
```

Конечно, оператор `subscribe(System.out::println)` работал бы ничуть не хуже; мы оставили информацию о типе в учебных целях. На выходе печатаются три события, порожденные оператором `buffer(3)`:

```
[1, 2, 3]
[4, 5, 6]
[7]
```

Оператор `buffer()` продолжает получать входящие события и сохраняет их во внутреннем буфере, пока тот не достигнет размера 3. Как только это произойдет, весь буфер (типа `List<Integer>`) передается дальше. Если при появлении уведомления о завершении внутренний буфер не пуст (но и не достиг размера 3), то он все равно передается дальше. Именно поэтому мы видим в конце список из одного элемента.

Оператор `buffer(int)` позволяет заменить несколько мелких событий меньшим числом более крупных пакетов. Например, чтобы уменьшить нагрузку на базу данных, мы могли бы сохранять события не по отдельности, а пакетом:

```
interface Repository {
    void store(Record record);
    void storeAll(List<Record> records);
}
```

```
//...
```

```
Observable<Record> events = //...
```

```
events
    .subscribe(repository::store);
// или
events
    .buffer(10)
    .subscribe(repository::storeAll);
```

Во втором случае подписчик вызывает метод `storeAll` объекта `Repository`, который сохраняет сразу целый пакет из 10 элементов. Потенциально это может увеличить пропускную способность приложения.

У оператора `buffer()` много перегруженных вариантов. Чуть более сложный позволяет задать, сколько самых старых значений из внутреннего буфера отбра-

сывать при отправке списка дальше. Звучит не очень понятно, но, по существу, речь о том, что вы можете смотреть на поток событий через скользящее окно определенного размера:

```
Observable
    .range(1, 7)
    .buffer(3, 1)
    .subscribe(System.out::println);
```

В результате получаем несколько перекрывающихся списков:

```
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
[4, 5, 6]
[5, 6, 7]
[6, 7]
[7]
```

Для вычисления скользящего среднего временного ряда¹ можно воспользоваться оператором вида `buffer(N, 1)`. В следующем примере генерируется 1000 случайных чисел с нормальным распределением. Затем мы берем скользящее окно из 100 элементов (продвигая его на один элемент за один шаг) и вычисляем среднее по такому окну². Выполните эту программу и убедитесь, что скользящее среднее изменяется гораздо более плавно, чем случайные неупорядоченные значения.

```
import java.util.Random;
import java.util.stream.Collectors;

//...

Random random = new Random();
Observable
    .defer(() -> just(random.nextGaussian()))
    .repeat(1000)
    .buffer(100, 1)
    .map(this::averageOfList)
    .subscribe(System.out::println);

//...

private double averageOfList(List<Double> list) {
    return list
        .stream()
        .collect(Collectors.averagingDouble(x -> x));
}
```

Нетрудно сообразить, что вызов `buffer(N)` эквивалентен `buffer(N,N)`. В простейшей форме `buffer()` уничтожает весь внутренний буфер, когда тот заполняется

¹ https://en.wikipedia.org/wiki/Moving_average.

² Имейте в виду, что это не самый эффективный алгоритм, потому что он многократно складывает одни и те же числа.

ся. Интересно, что второй параметр в варианте `buffer(int, int)` (определяющий, сколько элементов пропустить при отправке буфера дальше) может быть больше первого, при этом некоторые элементы вообще отбрасываются!

```
Observable<List<Integer>> odd = Observable
    .range(1, 7)
    .buffer(1, 2);
odd.subscribe(System.out::println);
```

В данном случае мы отправляем первый элемент, затем два пропускаем: первый и второй. Далее цикл повторяется: `buffer()` отправляет третий элемент, но пропускает третий и четвертый. В результате на выходе получается такая последовательность: [1] [3] [5] [7]. Обратите внимание, что каждый элемент потока `odd` представляет собой список из одного элемента. Применив к нему оператор `flatMap()` или `flatMapIterable()`, мы можем снова получить простой `Observable<Integer>`:

```
Observable<Integer> odd = Observable
    .range(1, 7)
    .buffer(1, 2)
    .flatMapIterable(list -> list);
```

Оператор `flatMapIterable()` принимает функцию, которая преобразует каждое значение в потоке (одноэлементный список `List<Integer>`) в `List`. Здесь достаточно указать тождественное преобразование (`list -> list`).

Буферизация по времени

На самом деле, `buffer()` – обширное семейство операторов. Один из них собирает события в пакет не по размеру (так что размеры всех пакетов одинаковы), а по времени. Если операторы `throttleFirst()` и `throttleLast()` отбирали только первое и последнее событие в указанном периоде, то вышеупомянутый вариант `buffer()` запоминает в буфере все события за период. Вернемся к примеру с нами:

```
Observable<String> names = just(
    "Mary", "Patricia", "Linda", "Barbara", "Elizabeth",
    "Jennifer", "Maria", "Susan", "Margaret", "Dorothy");
Observable<Long> absoluteDelays = just(
    0.1, 0.6, 0.9, 1.1, 3.3,
    3.4, 3.5, 3.6, 4.4, 4.8
).map(d -> (long) (d * 1_000));

Observable<String> delayedNames = Observable.zip(names,
    absoluteDelays,
    (n, d) -> just(n).delay(d, MILLISECONDS)
).flatMap(o -> o);

delayedNames
    .buffer(1, SECONDS)
    .subscribe(System.out::println);
```


Здесь перегруженный вариант `buffer()` принимает период времени (одна секунда) и собирает все события, поступившие за этот период. В результате на выходе мы получаем:

```
[Mary, Patricia, Linda]
[Barbara]
[]
[Elizabeth, Jennifer, Maria, Susan]
[Margaret, Dorothy]
```

Третий список строк пуст, потому что за этот период не было ни одного события. Такой вариант `buffer()` полезен, например, для подсчета количества событий в последовательных промежутках времени, скажем, количества событий клавиатуры в секунду:

```
Observable<KeyEvent> keyEvents = //...

Observable<Integer> eventPerSecond = keyEvents
    .buffer(1, SECONDS)
    .map(List::size);
```

Если, на наше счастье, не будет односекундных промежутков без событий, то мы не получим разрыва в результатах измерений. Впрочем, это не самый эффективный способ, как мы скоро убедимся при обсуждении оператора `window()`.

Самый полный перегруженный вариант `buffer()` позволяет точно контролировать, когда начинать буферизацию событий и когда отправлять буфер дальше. Иными словами, вы определяете периоды группировки входящих событий. Допустим, что программа следит за каким-то устройством, которое очень часто отправляет данные телеметрии. Объем данных гигантский, поэтому, чтобы сэкономить на обработке, мы хотим рассматривать только выборку данных. Алгоритм такой:

- в течение рабочего дня (9:00–17:00) мы выберем интервал длительностью 100 мс из каждой секунды (приблизительно 10 % данных);
- в нерабочее время выбираются интервалы длительностью 200 мс из каждого пятисекундного интервала (4 %).

Иначе говоря, один раз в секунду (или в 5 секунд) мы сохраняем в буфере все события, произошедшие в течение 100 мс (соответственно 200 мс) и порождаем список событий за этот период. Это станет понятно, если взглянуть на пример целиком. Прежде всего, нам нужен объект `Observable`, который порождает какое-нибудь значение всякий раз, как мы хотим начать буферизацию (группировку) входящих событий. Этот объект может отправлять все, что угодно, нас интересует только хронометраж. Тот факт, что мы возвращаем значение типа `Duration` из пакета `java.time`, — чистое совпадение, RxJava это значение никак не использует:

```
Observable<Duration> insideBusinessHours = Observable
    .interval(1, SECONDS)
    .filter(x -> isBusinessHour())
```

```

    .map(x -> Duration.ofMillis(100));
Observable<Duration> outsideBusinessHours = Observable
    .interval(5, SECONDS)
    .filter(x -> !isBusinessHour())
    .map(x -> Duration.ofMillis(200));

Observable<Duration> openings = Observable.merge(
    insideBusinessHours, outsideBusinessHours);

```

Сначала мы с помощью оператора `interval()` генерируем тики таймера раз в секунду, исключая те, что пришлись на нерабочее время. Таким образом, мы имеем стабильный поток тиков каждую секунду с 9:00 до 17:00. Напомню, что `interval()` возвращает возрастающие натуральные числа типа `Long`, но они нам ни к чему, поэтому для удобства заменим их фиксированными промежутками времени длительностью 100 мс. Следующий далее код создает аналогичный поток событий с интервалом в 5 секунд с 17:00 до 9:00. Если вам интересно, реализация метода `isBusinessHour()` основана на пакете `java.time`:

```

private static final LocalTime BUSINESS_START = LocalTime.of(9, 0);
private static final LocalTime BUSINESS_END = LocalTime.of(17, 0);

private boolean isBusinessHour() {
    ZoneId zone = ZoneId.of("Europe/Warsaw");
    ZonedDateTime zdt = ZonedDateTime.now(zone);
    LocalTime localTime = zdt.toLocalTime();
    return !localTime.isBefore(BUSINESS_START)
        && !localTime.isAfter(BUSINESS_END);
}

```

Поток `openings` получается объединением потоков `insideBusinessHours` и `outsideBusinessHours`. По существу, это триггер, который инструктирует оператор `buffer()`, когда начинать выборку событий из входного потока, а когда отбрасывать их. Какие конкретно значения порождает поток `openings`, совершенно не важно. Но мы должны также указать, когда прекратить буферизацию событий и отправить их дальше одним пакетом (списком). Самое очевидное решение – рассматривать каждое событие из потока `openings` как сигнал закончить текущий пакет, передать его дальше и начать следующий:

```

Observable<TeleData> upstream = //...

Observable<List<TeleData>> samples = upstream
    .buffer(openings);

```

Обратите внимание, что мы передаем оператору `buffer()` тщательно сконструированный поток `openings`. Этот код нарезает поток `upstream` событий типа `TeleData` на куски. Тики таймера из пакета `openings` отмечают начало пакета событий из потока `upstream`. В рабочее время пакет создается раз в секунду, а в нерабочее – раз в 5 секунд. Важно, что в этой версии все события из `upstream` сохраняются, т. е. попадают в тот или другой пакет. Однако есть другой вариант оператора `buffer()`, позволяющий отмечать также конец пакета:

```
Observable<List<TeleData>> samples = upstream
    .buffer(
        openings,
        duration -> empty()
    ).delay(duration.toMillis(), MILLISECONDS));
```

Вспомним, что `openings` имеет тип `Observable<Duration>`, но фактические значения событий из `openings` не важны. RxJava просто использует эти события, чтобы начать буферизацию экземпляров `TeleData`. Однако теперь мы полностью контролируем моменты начала буферизации и отправки буфера. Второй параметр – это объект `Observable`, завершение которого означает прекращение выборки, т. е. конец данного пакета. Будьте внимательны: поток `openings` порождает событие всякий раз, как мы хотели бы начать новый пакет. Для каждого события, поступившего из `openings`, мы возвращаем новый объект `Observable`, который должен завершиться в какой-то момент в будущем. Так, если `openings` порождает событие `Duration.ofMillis(100)`, то мы преобразуем его в `Observable`, который завершится тогда, когда нужно будет закончить данный пакет, т. е. через 100 мс. Отметим, что в этом случае некоторые события могут отбрасываться или дублироваться в соседних пакетах. Если второй `Observable` – тот, что отвечает за конец данного пакета, – появляется раньше события, открывающего следующий пакет, то все события, случившиеся в промежутке между этими двумя моментами, отбрасываются. Это как раз наш случай: мы начинаем заполнение буфера раз в секунду (или раз в пять секунд в нерабочее время), а прекращаем заполнение и отправляем буфер дальше по истечении 100 миллисекунд (или соответственно 200). Большинство событий попадают в промежуток между периодами буферизации и, стало быть, отбрасываются.

Оператор `buffer()` исключительно гибкий и довольно сложный. Обязательно поэкспериментируйте с ним и досконально разберитесь с приведенным примером. Он используется для управляемого агрегирования входящих событий: группировки, выборки и организации скользящих окон. Но поскольку `buffer()` по необходимости создает промежуточный список `List` перед закрытием и передачей дальше текущего буфера, он может возлагать избыточную нагрузку на сборщик мусора и потреблять много памяти (см. раздел «Потребление памяти и утечки» главы 8). Поэтому был предложен оператор `window()`.

Скользящее окно

При работе с оператором `buffer()` мы снова и снова создаем экземпляры списка `List`. А зачем нужны эти промежуточные списки? Нельзя ли как-нибудь потреблять события на лету? Именно для этого предназначен оператор `window()`. Старайтесь всюду, где возможно, использовать `window()`, а не `buffer()`, потому что он более предсказуем в плане потребления памяти. Оператор `window()` очень похож на `buffer()`: у него такие же перегруженные варианты, в т. ч. для выполнения следующих действий:

- получает `int`, группирует входящие события в списке фиксированной длины;

- получает единицу времени, группирует события, поступившие в течение интервала времени заданной длительности;
- получает объекты `Observable`, отмечающие начало и конец каждого пакета.

Но тогда в чем разница? Помните пример подсчета количества событий в секунду из данного источника? Давайте взглянем на него по-новому:

```
Observable<KeyEvent> keyEvents = //...
Observable<Integer> eventPerSecond = keyEvents
    .buffer(1, SECONDS)
    .map(List::size);
```

Мы собираем в пакеты события из источника `Observable<KeyEvent>`, произошедшие в течение каждой секунды, и помещаем эти пакеты в объект типа `Observable<List<KeyEvent>>`. На следующем шаге список `List` отображается на его размер. Это довольно расточительно, особенно если каждую секунду происходит много событий. А вот другое решение:

```
Observable<Observable<KeyEvent>> windows = keyEvents.window(1, SECONDS);
Observable<Integer> eventPerSecond = windows
    .flatMap(eventsInSecond -> eventsInSecond.count());
```

Оператор `window()`, в отличие от `buffer()`, возвращает объект типа `Observable<Observable<KeyEvent>>`. Что мы здесь имеем? Вместо того чтобы получать списки фиксированного размера по одному пакету (буферу) в каждом, мы получаем поток потоков. Как только начинается новый пакет (в нашем примере каждую секунду), во внешнем потоке появляется новое значение типа `Observable<KeyEvent>`. Мы можем преобразовывать внутренние потоки и дальше, но, чтобы избежать двойного обертывания, применяем оператор `flatMap()`. Этот оператор получает один буфер (`Observable<KeyEvent>`) в качестве аргумента и должен вернуть другой `Observable`. Оператор `count()` (см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3) преобразует `Observable<T>` в объект `Observable<Integer>`, который порождает ровно один элемент, представляющий число событий в исходном `Observable`. Следовательно, для каждого пакета событий в течение одной секунды, мы порождаем число событий за эту секунду. Но никакой внутренней буферизации нет и в помине: оператор `count()` подсчитывает события на лету.

Пропуск устаревших событий с помощью `debounce()`

Операторы `buffer()` и `window()` собирают события в группы, чтобы можно было обработать их пакетно. Оператор `sample()` периодически выбирает одно случайное событие. Эти операторы не учитывают, сколько времени прошло между событиями. Но во многих случаях событие можно отбросить, если вскоре после него происходит другое событие. Представим себе поток биржевых цен, генерируемый торговой площадкой:

```
Observable<BigDecimal> prices = tradingPlatform.pricesOf("NFLX");  
Observable<BigDecimal> debounced = prices.debounce(100, MILLISECONDS);
```

Оператор `debounce()` (псевдоним: `throttleWithTimeout()`) отбрасывает все события, за которыми с небольшим промежутком следует другое событие. Иначе говоря, если в некотором временном окне после данного события нет никаких других, то это событие передается дальше. В примере выше поток `prices` содержит цены акций компании "NFLX" в моменты их изменения. Иногда цены меняются очень часто, десятки раз в секунду. Для каждого изменения цены нужно произвести вычисление, занимающее много времени. Но если появляется новая цена, то результат этого вычисления уже не имеет значения, и надо начинать все сначала. Поэтому мы хотели бы отбрасывать событие, если почти сразу после него идет следующее (и подавляет его).

Оператор `debounce()` ждет короткое время (в примере выше 100 мс) на случай, если вдруг появится следующее событие. Этот процесс повторяется, т. е. если второе событие отстоит от первого менее чем на 100 мс, то RxJava задержит его отправку в надежде, что вскоре появится третье. И в этом случае у нас есть возможность гибко контролировать время ожидания для каждого события в отдельности. Например, мы можем игнорировать изменение цены, если следующее изменение происходит менее чем через 100 мс. Но если цена оказывается выше 150 долларов, то мы хотим безо всяких колебаний отправить его дальше – и побыстрее. Быть может, потому что события некоторых типов должны обрабатываться незамедлительно, т. к. они открывают интересные рыночные возможности. Это легко реализовать с помощью перегруженного варианта `debounce()`:

```
prices  
    .debounce(x -> {  
        boolean goodPrice = x.compareTo(BigDecimal.valueOf(150)) > 0;  
        return Observable  
            .empty()  
            .delay(goodPrice? 10 : 100, MILLISECONDS);  
    })
```

К каждому обновлению цены `x` применяется некоторая логика (`> $150`), чтобы понять, представляет ли новая цена интерес. Затем для каждого обновления мы возвращаем новый – пустой – объект `Observable`. Он не должен порождать никаких элементов; важно только его завершение. Для интересных цен он порождает уведомление о завершении через 10 мс, а для остальных – через 100 мс. Для каждого полученного события оператор `debounce()` подписывается на этот объект `Observable` и ждет его завершения. Если он сначала завершается, то событие передается дальше. В противном случае – если до его завершения появилось входящее событие – цикл повторяется.

В нашем примере при появлении цены `x`, равной 140 долларов, оператор `debounce()` создает новый `Observable` с завершением, отложенным на 100 мс. Если до его завершения не придет никаких событий, то событие с ценой 140 долларов будет отправлено дальше. Но допустим, что пришло новое обновление цены `x`, рав-

ное 151 доллару. На этот раз, когда `debounce()` попросит у нас объект `Observable` (в описании API он называется `debounceSelector`), мы вернем поток, который завершается гораздо быстрее, через 10 мс. Таким образом, в случае интересной цены (выше 150 долларов) мы готовы ждать следующего обновления всего 10 мс. Если вы еще не поняли, как работает `debounce()`, то можете поэкспериментировать со следующим эмулятором биржевых цен:

```
Observable<BigDecimal> pricesOf(String ticker) {
    return Observable
        .interval(50, MILLISECONDS)
        .flatMap(this::randomDelay)
        .map(this::randomStockPrice)
        .map(BigDecimal::valueOf);
}

Observable<Long> randomDelay(long x) {
    return Observable
        .just(x)
        .delay((long) (Math.random() * 100), MILLISECONDS);
}

double randomStockPrice(long x) {
    return 100 + Math.random() * 10 +
        (Math.sin(x / 100.0)) * 60.0;
}
```

В этом коде мы видим композицию нескольких потоков. Сначала генерируется последовательность значений типа `long`, разделенных фиксированными интервалами длительностью 50 мс. Затем каждое событие независимо задерживается на случайную величину от 0 до 100 мс. И наконец, бесконечная возрастающая последовательность чисел преобразуется в синусоидальную волну (с помощью `Math.sin()`) со случайным дрожанием. Таким образом, мы моделируем колебания биржевых цен со временем. Если применить к этому потоку оператор `debounce()`, то мы заметим, что пока цена низкая, события порождаются относительно редко, потому что мы готовы ждать следующее событие 100 мс, и такая ситуация встречается часто. Но как только цена достигает 150 долларов, порог ожидания в `debounce()` снижается до 10 мс, так что практически каждая интересная цена передается дальше.

Предотвращение зависания в `debounce()`

Легко представить себе ситуацию, когда оператор `debounce()` задерживает вообще все события, потому что они появляются слишком часто, не давая никакой передышки:

```
Observable
    .interval(99, MILLISECONDS)
    .debounce(100, MILLISECONDS)
```

Такой источник не порождает ни одного события, потому что `debounce()` ждет более свежее событие в течение 100 мс. Но как раз за 1 мс до истечения срока

ожидания появляется новое событие, и таймер `debounce` взводится заново. Следовательно, мы имеем `Observable`, который порождает события так часто, что ни одно из них до нас не доходит! Можно, конечно, сказать, что так и задумано, но на практике мы все-таки хотим хотя бы иногда видеть события, даже в случае затопления. Чтобы предотвратить такую ситуацию, придется проявить изобретательность.

Сначала нужно обнаружить, что в течение долгого времени не было событий. В разделе «Таймаут в случае отсутствия событий» главы 7 будет описан оператор `timeout()`, благодаря которому эту часть реализовать нетрудно:

```
Observable
    .interval(99, MILLISECONDS)
    .debounce(100, MILLISECONDS)
    .timeout(1, SECONDS);
```

Теперь мы, по крайней мере, получаем исключение, сигнализирующее о том, что входной поток истощился. Забавно, что на самом деле все обстоит ровно наоборот – предшествующий оператор `interval()` порождает события настолько часто, что `debounce()` не пропускает их дальше, но это мы отвлеклись. Если события появляются слишком часто, мы придерживаем их, дожидаясь момента тишины. Но если тишина продолжается слишком долго (более одной секунды), то возбуждается исключение `TimeoutException`. Но мы хотели бы не получать исключения постоянно, а увидеть хоть какое-нибудь значение из входного потока и продолжить. С первой частью справиться легко:

```
ConnectableObservable<Long> upstream = Observable
    .interval(99, MILLISECONDS)
    .publish();
upstream
    .debounce(100, MILLISECONDS)
    .timeout(1, SECONDS, upstream.take(1));
upstream.connect();
```



Тип `ConnectableObservable`

Тип `ConnectableObservable` в сочетании с операторами `publish()` и `connect()` необходим здесь для того, чтобы преобразовать холодный объект `Observable.interval()` в горячий (о том, почему `interval()` холодный и что это означает, см. раздел «Горячие и холодные объекты `Observable`» главы 2). Посредством последовательных вызовов `publish()` и `connect()` (см. раздел «Жизненный цикл `ConnectableObservable`» главы 2) мы заставляем оператор `interval()` начать порождение событий немедленно, не дожидаясь подписчиков. Это означает, что если мы подпишемся на такой `Observable` спустя много секунд, то начнем получать поток событий с середины, и что все подписчики будут видеть одни и те же события в одно и то же время. По умолчанию же `interval()` – холодный `Observable`, т. е. каждый подписчик вне зависимости от момента подписки будет получать события, начиная с нулевого.

У оператора `timeout()` имеется перегруженный вариант, который принимает объект `Observable`, получающий управление в случае таймаута. К сожалению, в приведенном выше коде есть тонкая ошибка. В случае таймаута мы наивно выбираем из входного потока первое встретившееся значение и завершаемся. Но на самом-то деле нам нужно продолжить порождение событий из потока `upstream`, по-прежнему с поддержкой `debounce()`.

Другой подход выглядит лучше:

```
upstream
  .debounce(100, MILLISECONDS)
  .timeout(1, SECONDS, upstream)
  .take(1)
  .concatWith(
    upstream.debounce(100, MILLISECONDS))
```

На первый взгляд, все нормально. В исходном источнике после применения к нему `debounce()` происходит таймаут. В этот момент мы порождаем первое же встретившееся событие и продолжаем получать события из того же источника, к которому, как и раньше, применяется `debounce()`. Однако после первого таймаута мы переключаемся на резервный `Observable`, для которого оператора `timeout()` уже нет. Вот исправление, сделанное на скорую руку и, к тому же, недальновидное:

```
upstream
  .debounce(100, MILLISECONDS)
  .timeout(1, SECONDS, upstream)
  .take(1)
  .concatWith(
    upstream
      .debounce(100, MILLISECONDS)
      .timeout(1, SECONDS, upstream))
```

Но теперь мы забыли указать резервный `Observable` во внутреннем операторе `timeout()`. Ну хватит, повторяющаяся картина уже видна. Так вместо того чтобы бесконечно повторять последовательность вида `upstream → debounce → timeout() → upstream → ...`, не лучше ли воспользоваться рекурсией?

```
import static rx.Observable.defer;

Observable<Long> timedDebounce(Observable<Long> upstream) {
    Observable<Long> onTimeout = upstream
        .take(1)
        .concatWith(defer(() -> timedDebounce(upstream)));
    return upstream
        .debounce(100, MILLISECONDS)
        .timeout(1, SECONDS, onTimeout);
}
```

Определение объекта `onTimeout`, получающего управление в случае таймаута, довольно хитроумное. Мы объявляем, что он сначала получает одно событие из потока `upstream` (исходного источника), а затем рекурсивно вызываем метод

`timedDebounce()`. Оператор `defer()` необходим, чтобы избежать бесконечной рекурсии. Оставшаяся часть `timedDebounce()` проста: взять исходный источник, применить к нему оператор `debounce()` и добавить `timeout()` с указанием объекта `onTimeout` в качестве резервного. Этот объект делает как раз то, что нужно: применяет `debounce()`, добавляет `timeout()` с резервным объектом – и так далее рекурсивно.



Не впадайте в уныние, если с первого раза понять этот код не удалось. Это довольно сложный пример, демонстрирующий мощь композиции потоков в сочетании с ленивостью и рекурсией. Вам вряд ли когда-нибудь понадобится такой уровень сложности, но, разобравшись как этот код работает, вы получите удовольствие. Поэкспериментируйте с ним и посмотрите, как даже небольшие изменения радикально изменяют характер взаимодействия потоков.

Противодавление

Механизм противодавления крайне важен для построения надежных приложений с малым временем отклика. По существу, это канал обратной связи от потребителя к производителю. Потребитель может указать, сколько данных он готов обработать в каждый момент времени. Это позволяет избежать перегрузки потребителей или промежуточного уровня передачи сообщений при высокой нагрузке. Они просят уменьшить количество сообщений, а уж производитель должен решить, как это сделать.

Проблема потребителей, не поспевающих за производителями, может возникнуть в любой системе, где обмен данными основан на передаче сообщений (или событий). В зависимости от реализации она может проявляться разными способами. Если коммуникационный канал каким-то образом синхронизирует производителей и потребителей (например, используется класс `ArrayBlockingQueue`), то производитель блокируется, если потребитель не справляется с нагрузкой. Это приводит к появлению связи между производителем и потребителем, которые вообще-то должны быть абсолютно независимыми. Обычно передача сообщений подразумевает асинхронную обработку, но это предположение оказывается несостоятельным, если производитель ни с того ни с сего принужден ждать потребителя. Хуже того, производитель может оказаться потребителем другого производителя на более высокой ступени иерархии, так что задержка будет распространяться снизу вверх.

С другой стороны, если среда передачи между двумя сторонами неограниченна, то ... то она все равно ограничена, только факторами, которые мы хуже контролируем. Бесконечная очередь типа `LinkedBlockingQueue` позволяет производителю намного опережать потребителей без блокирования. Но только до тех пор, пока очередь не займет всю память, что приведет к краху приложения. Если среда допускает внешнее хранение, как, например брокер сообщений JMS, то та же самая

проблема может проявиться как переполнение диска, хотя это и менее вероятно. Гораздо чаще встречается ситуация, когда промежуточный уровень обмена сообщениями не может справиться с тысячами, а то и миллионами ожидающих потребления сообщений. Некоторые специализированные брокеры, например Kafka (<http://kafka.apache.org/>) технически могут хранить сотни миллионов сообщений, ждущих внимания со стороны отставших потребителей. Но это означает гигантскую временную задержку между порождением и потреблением сообщения.

Хотя системы, управляемые сообщениями, вообще говоря, считаются более надежными и масштабируемыми, проблема чересчур энергичных производителей остается нерешенной. Однако предпринимается ряд усилий как-то разрешить этот интеграционный вопрос. Выборка и отбрасывание событий (оператор `sample()` и другие), а также пакетирование (`window()` и `buffer()`) – это способы снижения нагрузки со стороны производителя вручную. Когда `observable` генерирует больше событий, чем программа в состоянии потребить, применение выборки или пакетирования способно повысить пропускную способность подписчиков. Но все же ощущалась потребность в более надежном и систематическом подходе, поэтому инициатива Reactive Streams (<http://www.reactive-streams.org/>) была рождена по необходимости. Этот небольшой набор интерфейсов и семантических принципов, призванный формализовать проблему и предложить систематический алгоритм координации производителя и потребителя, получил название *противодавление*.

Противодавление – простой протокол, позволяющий потребителю указать, сколько данных он способен потребить в данный момент. Тем самым предоставляется канал обратной связи с производителем. Производители получают запросы от потребителей и предотвращают затопление сообщениями. Конечно, этот алгоритм будет работать только для таких производителей, которые умеют ограничивать себя. Это возможно, например, если производитель получает сообщения из статической коллекции или вытягивает из какого-то источника, пользуясь механизмом, аналогичным `Iterator`. Если же производитель не контролирует частоту порождаемых им данных (источник внешний или горячий), то противодействие не поможет.

Противодавление в RxJava

Механизм Reactive Streams призван решить очень общую задачу и не зависит от технологии, но мы сосредоточимся на RxJava и на том, как в ней реализовано противодействие. В этом разделе мы в качестве примера будем использовать мытье тарелок в небольшом ресторанчике³. Тарелки моделируются большими объектами, имеющими идентификатор:

```
class Dish {
    private final byte[] oneKb = new byte[1_024];
    private final int id;
```

```
Dish(int id) {
```

³ Этот пример подсказан статьей <https://www.lightbend.com/blog/7-ways-washing-dishes-and-message-driven-reactive-systems>.

```
this.id = id;
System.out.println("Создана: " + id);
}

public String toString() {
    return String.valueOf(id);
}
}
```

Буфер `onekb` моделирует дополнительно занятую память. Тарелки передают на кухню официанты, этот поток моделируется объектом `Observable`:

```
Observable<Dish> dishes = Observable
    .range(1, 1_000_000_000)
    .map(Dish::new);
```

Оператор `range()` порождает новые значения в максимально быстром темпе. И что произойдет, если мытье тарелки занимает некоторое время, очевидно, большее времени, необходимого для ее порождения?

```
Observable
    .range(1, 1_000_000_000)
    .map(Dish::new)
    .subscribe(x -> {
        System.out.println("Моется: " + x);
        sleepMillis(50);
    });
```

Как ни странно, ничего страшного. Взглянув на распечатку, мы увидим, что `range()` точно синхронизирован с подписчиком:

```
Создана: 1
Моется: 1
Создана: 2
Моется: 2
Создана: 3
Моется: 3
...
Создана: 110
Моется: 110
...
```

Это не должно вызывать удивления. Оператор `range()` по умолчанию не является асинхронным, поэтому всякое порождаемое им значение передается напрямую подписчику в том же потоке. Если подписчик работает медленно, то `Observable` не сможет увеличить темп порождения элементов. Оператор `range()` не может вызвать метод `onNext()` подписчика, пока не закончил работать предыдущий вызов. Но это возможно только потому, что производитель и потребитель работают в одном потоке, так что между ними имеется неявная связь. В каком-то смысле эту связь можно назвать очередью длины 1. То есть имеет место непредусмотренный алгоритм *рандеву*. Представьте себе официанта, который не может оставить

для мытья новые тарелки, пока предыдущие не будут вымыты. Но если официант ждет возможности отдать тарелки на кухню, то гостей никто не обслуживает. А раз гостей никто не обслуживает, то новые гости не могут войти в ресторан. Так один блокирующий компонент может привести к зависанию всей системы. Однако в реальной жизни между потоками производителя и потребителя обычно проведена граница: `Observable` порождает события в одном потоке, а `Subscriber` потребляет их в другом:

```
dishes
    .observeOn(Schedulers.io())
    .subscribe(x -> {
        System.out.println("Моется: " + x);
        sleepMillis(50);
    });
```

Притормозим и подумаем о том, что здесь произойдет, компилировать и запустить программу пока не будем. Первая мысль – случится катастрофа, потому что поток `dishes` порождает события очень быстро – в темпе оператора `range()`, тогда как подписчик медленный – потребляет только 20 тарелок в секунду. И значит, заключаем мы, исключение `OutOfMemoryError` неизбежно, поскольку где-то же необработанные события должны накапливаться. По счастью, противодействие в этом случае спасает, и `RxJava` обеспечивает нам кое-какую защиту. Вывод программы выглядит неожиданно:

```
Создана: 1
Создана: 2
Создана: 3
...
Создана: 128
Моется: 1
Моется: 2
...
Моется: 128
Создана: 129
...
Создана: 223
Создана: 224
Моется: 129
Моется: 130
...
```

Сначала оператор `range()` почти мгновенно порождает порцию из 128 тарелок. Затем мы видим медленный процесс поочередного мытья тарелок. Почему-то оператор `range()` ведет себя тихо. После того как вымыта последняя из 128 тарелок, `range()` порождает следующую порцию – 96 тарелок, затем снова следует медленный процесс мытья⁴. Очевидно, должен существовать какой-то хитроумный механизм, не дающий `range()` порождать слишком много событий, и этот механизм

⁴ Не обращайтесь внимания на конкретные цифры, важно лишь, что кто-то периодически запрашивает порции событий.

контролируется подписчиком. Пока мы не понимаем, где этот механизм находится, поэтому попробуем реализовать оператор `range()` самостоятельно:

```
Observable<Integer> myRange(int from, int count) {
    return Observable.create(subscriber -> {
        int i = from;
        while (i < from + count) {
            if (!subscriber.isUnsubscribed()) {
                subscriber.onNext(i++);
            } else {
                return;
            }
        }
        subscriber.onCompleted();
    });
}
```

А теперь используем `myRange()` вместе с `observeOn()`:

```
myRange(1, 1_000_000_000)
    .map(Dish::new)
    .observeOn(Schedulers.io())
    .subscribe(x -> {
        System.out.println("Моется: " + x);
        sleepMillis(50);
    },
        Throwable::printStackTrace
    );
```

Все кончается печально, мы так и не сумеем помыть ни одной тарелки:

```
Создана: 1
Создана: 2
Создана: 3
...
Создана: 7177
Создана: 7178
rx.exceptions.MissingBackpressureException
    at rx.internal.operators...
    at rx.internal.operators...
```

Что такое `MissingBackpressureException`, я объясню позже. А пока надеюсь, что эта демонстрация убедительно доказывает существование некоего скрытого механизма, который в нашей реализации `range()` отсутствует.

Встроенное противодавление

В предыдущих главах мы видели, как события проходят от исходного `Observable` через последовательность операторов к подписчику. Нигде не было и следа канала обратной связи, если не считать самой подписки. С момента вызова `subscribe()` (который в каком-то смысле распространяется вверх по цепочке) все события и уведомления проходят вниз, никакой петли обратной связи на поверхности не на-

блюдается. Из-за такого отсутствия обратной связи производитель (самый верхний Observable) может порождать столько событий, что подписчик захлебнется. И в результате приложение «грохнется» с исключением `OutOfMemoryError` или в лучшем случае будет сильно тормозить.

Механизм противодавления позволяет как конечным подписчикам, так и всем промежуточным операторам запрашивать у производителя определенное число событий и не более. По умолчанию исходный холодный Observable порождает события так быстро, как может. Но при наличии подобных запросов снизу он должен каким-то образом «замедлиться» и породить ровно столько событий, сколько запрошено. Отсюда и магическое число 128, которое мы видели в примере с `observeOn()`. Но сначала посмотрим, как конечный подписчик может управлять противодавлением.

В момент подписки у нас есть возможность реализовать обратные вызовы `onNext()`, `onCompleted()` и `onError()` (см. раздел «Подписка на уведомления от объекта Observable» главы 2). Оказывается, существует еще один метод обратного вызова: `onStart()`.

```
Observable
    .range(1, 10)
    .subscribe(new Subscriber<Integer>() {

        @Override
        public void onStart() {
            request(3);
        }

        // onNext, onCompleted, onError ...
    });
```

RxJava вызывает метод `onStart()` перед отправкой подписчику любого события или уведомления – в точности как вы, наверное, и предположили. Технически можно было бы воспользоваться конструктором объекта `Subscriber`, но конструкторы анонимных внутренних классов в Java выглядят жутковато:

```
.subscribe(new Subscriber<Integer>() {

    {{
        request(3);
    }}

    // onNext, onCompleted, onError ...
});
```

Впрочем, мы отвлеклись. Вызов `request(3)` из объекта `Subscriber` говорит источнику, сколько элементов мы хотим получить сначала. Если опустить этот вызов или написать `request(Long.MAX_VALUE)`, то будет считаться, что мы запрашиваем максимально возможное число событий. Именно поэтому вызывать `request()` нужно как можно раньше, иначе поток начнет порождать события, и мы уже не сможем его притормозить. А поскольку мы запросили всего три события, оператор

`range()` послушно приостанавливает порождение, отправив события 1, 2 и 3. Наш метод `onNext()` будет вызван только три раза, хотя `range()` еще не завершился. Подписчик полностью управляет тем, сколько данных хочет получить. Например, можно было бы запрашивать элементы по одному:

```
Observable
    .range(1, 10)
    .subscribe(new Subscriber<Integer>() {

        @Override
        public void onStart() {
            request(1);
        }

        @Override
        public void onNext(Integer integer) {
            request(1);
            log.info("Следующий {}", integer);
        }

        // onComplete, onError...
    });
```

Пример глуповатый, потому что ведет он себя, как обыкновенный подписчик без какого бы то ни было противодавления. Однако из него видно, как можно использовать противодавление. Легко представить себе подписчика, который сначала записывает в буфер сколько-то событий, а затем запрашивает их порциями, когда сочтет удобным. Подписчик мог бы решить, что стоит немного подождать, прежде чем получать новые события, пусть даже ему и нечем заняться, например, для того чтобы уменьшить нагрузку на следующие за ним компоненты. В примере с рестораном официант моделируется объектом `Observable<Dish>`, который продолжает таскать тарелки, а вызов `request(N)` означает готовность кухонного персонала вымыть еще сколько-то тарелок. Хороший официант не станет оставлять тарелки без запроса со стороны кухни.

При всем при том прямой вызов `request(N)` в клиентском коде – вещь редкая. Чаще различные операторы, которые мы вставляем между источником и конечным подписчиком, сами применяют противодавление, чтобы контролировать интенсивность потока данных в конвейере. Например, `observeOn()` должен подписаться на входной `Observable` и планировать все получаемые события с помощью конкретного диспетчера, например, `io()`. Но что, если входной поток порождает события в таком темпе, что ни диспетчер, ни подписчик за ним не успевают? Подписчик, создаваемый оператором `observeOn()`, поддерживает противодавление и в начале запрашивает только 128 значений⁵. Входной `Observable`, поддерживающий противодавление, порождает указанное число событий и приостанавливается – как, например, `range()`. Обнаружив, что эта порция событий успешно обработана подписчиком, оператор `observeOn()` запросит следующую. Таким образом, несмо-

⁵ Это значение можно изменить с помощью системного свойства `rx.ring-buffer.size`.

тря на то, что производитель и потребитель работают в разных потоках и по природе своей асинхронны, потребитель никогда не задыхается под грузом событий.

`observeOn()` — не единственный оператор, поддерживающий противодействие. Этим могут похвастаться еще десятки операторов. Например, `zip()` буферизует только фиксированное число событий из каждого входного `Observable`. Поэтому `zip()` спокойно относится к излишней активности одного из потоков. То же касается большинства используемых операторов.

Производители и исключение `MissingBackpressureException`

Работая с замороженной реализацией `range()`, мы уже столкнулись с исключением `MissingBackpressureException`. Но что оно означает и как его интерпретировать? Представьте подписчика (созданного вами или чаще каким-то оператором), который точно знает, сколько элементов хочет получить, например: `buffer(N)` или `take(N)`. Другой пример такого оператора дает `observeOn()`. Он обязан проявлять строгость в этом отношении: если входной `Observable` по какой-то причине порождает больше элементов, то внутренний буфер `observeOn()` переполняется, о чем он и сообщает с помощью исключения `MissingBackpressureException`. Но почему входной `Observable` порождает больше элементов, чем запрошено? Да потому что он игнорирует вызовы `request()`. Давайте вернемся к нашей простой реализации оператор `range()`:

```
Observable<Integer> myRange(int from, int count) {
    return Observable.create(subscriber -> {
        int i = from;
        while (i < from + count) {
            if (!subscriber.isUnsubscribed()) {
                subscriber.onNext(i++);
            } else {
                return;
            }
        }
        subscriber.onCompleted();
    });
}
```

Единственный способ остановить его — отписаться, но мы-то хотим не отписываться, а лишь немного притормозить поток. Последующие операторы точно знают, сколько событий хотят получить, но наш источник игнорирует их пожелания. Низкоуровневый механизм учета запрошенного числа событий объявлен в интерфейсе `rx.Producer` и включен в метод `create()`. Напомню, что метод `onSubscribeRange` вызывается всякий раз, как кто-то подписывается на объект `Observable`. Обычно из него напрямую вызывается `onNext()`, но не в том случае, когда в расчет принимается противодействие:


```
Observable<Integer> myRangeWithBackpressure(int from, int count) {
    return Observable.create(new OnSubscribeRange(from, count));
}

class OnSubscribeRange implements Observable.OnSubscribe<Integer> {

    // конструктор ...

    @Override
    public void call(final Subscriber<? super Integer> child) {
        child.setProducer(new RangeProducer(child, start, end));
    }
}

class RangeProducer implements Producer {

    @Override
    public void request(long n) {
        // здесь должен быть вызов onNext() дочернего подписчика
    }
}
```

Это набросок настоящей реализации `range()` в RxJava. Реализация интерфейса `Producer` – непростая задача: он должен иметь внутреннее состояние, быть потокобезопасным и работать чрезвычайно быстро. Поэтому обычно мы не пишем производители самостоятельно, но понимать, как они работают, полезно (о том, как самостоятельно реализовать противодавление, см. раздел «Учет запрошенного объема данных» ниже)⁶. На внутреннем уровне противодавление переворачивает принципы Rx с ног на голову. Объект `Observable`, порождаемый оператором `range()` (и многими другими встроенными операторами), больше не занимается энергичной отправкой данных подписчикам. Вместо этого он просыпается в ответ на запросы данных (вызовы `request(N)` подписчиком) и только тогда порождает события. Кроме того, он порождает не больше, чем запрошено.

Взгляните, как задается производитель `Producer` для дочернего подписчика `child`; этот `Producer` впоследствии будет косвенно вызываться, когда подписчик обращается к методу `request()`. Именно так создается канал обратной связи от подписчика к источнику типа `Observable`. Объект `Observable` инспектирует своего подписчика о том, как он может запросить определенный объем данных. По сути дела, `Observable` переходит с модели проталкивания на модель вытягивания-проталкивания, когда клиенты факультативно могут запрашивать ограниченное число событий. А что делать, если какой-то сторонний `Observable` не организовал такой канал? В том случае, когда RxJava обнаруживает, что имеет дело с источником, не поддерживающим противодавление, она вправе в любой возбудить исключение `MissingBackpressureException`. Однако существуют операторы из семейства `onBackpressure*`(), способные до некоторой степени эмулировать противодавление.

⁶ О том, насколько сложным и оптимизированным может оказаться даже простейший производитель, см., например, статью по адресу <https://github.com/ReactiveX/RxJava/blob/1.x/src/main/java/rx/internal/operators/OnSubscribeRange.java>.

Простейший оператор `onBackpressureBuffer()` безусловно буферизует все входящие события и отдает подписчикам только запрошенное количество:

```
myRange(1, 1_000_000_000)
    .map(Dish::new)
    .onBackpressureBuffer()
    .observeOn(Schedulers.io())
    .subscribe(x -> {
        System.out.println("Моется: " + x);
        sleepMillis(50);
    });
```

Как всегда, читаем снизу вверх. Сначала `subscribe()` поднимается вверх до оператора `observeOn()`. `observeOn()` также должен подписаться, но не может просто начать потреблять произвольное число событий. Поэтому он запрашивает фиксированное их количество (128), чтобы избежать переполнения очереди диспетчера `io()`. Оператор `onBackpressureBuffer()` боится от источников, игнорирующих противодавление. Получив от подписчика запрос `request(128)`, он передает его дальше наверх и ничего не делает, если через него прошло только 128 элементов. Но если `Observable` проигнорировал запрос и продолжать отдавать данные, не обращая внимания на противодавление, то `onBackpressureBuffer()` сохраняет излишки в неограниченном внутреннем буфере. Когда от подписчика приходит следующий запрос, `onBackpressureBuffer()` сначала отдает события из буфера и, лишь опустошив его, просит у входного `Observable` еще. Этот механизм позволяет оператору `observeOn()` работать так, будто `myRange()` поддерживает противодавление, хотя на самом деле ограничением интенсивности занимается `onBackpressureBuffer()`. К сожалению, неограниченный внутренний буфер – не та вещь, к которой можно относиться легкомысленно:

```
Создана: 1
Создана: 2
Создана: 3
Создана: 4
Создана: 8
Создана: 9
Моется: 1
Создана: 10
Создана: 11
...
Создана: 26976
Создана: 26977
Моется: 15
Exception in thread "main" java.lang.OutOfMemoryError: ...
Моется: 16
    at java.util.concurrent.ConcurrentLinkedQueue.offer...
    at rx.internal.operators.OperatorOnBackpressureBuffer...
...
```

Конечно, у вас может получиться другой результат: если события поменьше, а памяти побольше, то `onBackpressureBuffer()` может и сработать. Но на практике

никогда не следует рассчитывать на наличие неограниченных ресурсов. Ни память, ни SSD-диск не бесконечны. По счастью, существует перегруженный вариант `onBackpressureBuffer(N)`, который принимает максимальный размер буфера:

```
.onBackpressureBuffer(1000, () -> log.warn("Буфер полон"))
```

Второй параметр необязателен; это функция обратного вызова, которая вызывается, когда буфер на 1000 элемент заполнен, т. е. когда, несмотря на буферизацию, подписчик не способен обрабатывать события в нужном темпе. Никакой возможности восстановления после ошибки не предусмотрено, поэтому ожидайте исключения `MissingBackpressureException` сразу после предупреждения. Но над буфером у нас хотя бы есть контроль, в отличие от оборудования или операционной системы.

Альтернативой `onBackpressureBuffer()` является оператор `onBackpressureDrop()`, который просто отбрасывает все события, поступившие без предварительного `request()`. Представьте официанта, который продолжает носить тарелки на кухню. Оператор `onBackpressureBuffer()` можно сравнить с бесконечным столом, на который складываются немытые тарелки. С другой стороны, `onBackpressureDrop()` — это официант, который просто выбрасывает грязные тарелки, если в данный момент их невозможно вымыть. Такую модель бизнеса не назовешь неистощительной, но, по крайней мере, ресторан может продолжить обслуживание клиентов:

```
.onBackpressureDrop(dish -> log.warn("Выбрасывается {}"), dish))
```

Необязательная функция обратного вызова уведомляет о том, что событие отброшено, потому что пришло без запроса. Полезно было бы запоминать, сколько событий отброшено, этот показатель может оказаться важен. Наконец, существует оператор `onBackpressureLatest()`, который похож на `onBackpressureDrop()`, но сохраняет ссылку на последнее отброшенное событие, так что если впоследствии снизу придет запоздавший запрос `request()`, то последнее входящее событие будет отправлено. Семейство методов `onBackpressure*`() служит мостом между операторами и подписчиками, запрашивающими противодавление, и объектами `Observable`, которые его не поддерживают. Но лучше все-таки использовать или создать самостоятельно источники, поддерживающие противодавление изначально.

Учет запрошенного объема данных

Существует много способов построить `Observable`, поддерживающий противодавление. Проще всего воспользоваться встроенными фабричными методами типа `range()` или `from(Iterable<T>)`. Последний создает источник на основе `Iterable`, но со встроенной поддержкой противодавления. Такой `observable` не порождает все значения из `Iterable` сразу, а делает это постепенно, по мере поступления запросов от потребителей. Это вовсе не означает, что все данные должны быть сначала загружены в список `List<T>` (расширяющий `Iterable<T>`). Тип `Iterable` — это, на самом деле, фабрика итераторов `Iterator`, поэтому можно безопасно загружать данные на лету.

Интересный пример Observable с поддержкой противодавления дает обертывание объекта JDBC ResultSet потоком. Отметим, что класс ResultSet предназначен для вытягивания, как и Observable, поддерживающий противодавление. Но это не Iterable и не Iterator, поэтому сначала мы должны преобразовать его в Iterator<Object[]>, где Object[] – нетипизированное представление одной строки из базы данных.

```
public class ResultSetIterator implements Iterator<Object[]> {

    private final ResultSet rs;

    public ResultSetIterator(ResultSet rs) {
        this.rs = rs;
    }

    @Override
    public boolean hasNext() {
        return !rs.isLast();
    }

    @Override
    public Object[] next() {
        rs.next();
        return toArray(rs);
    }
}
```



Обработка ResultSet

Имейте в виду, что рассмотрение ResultSet как Iterator (и особенно как Iterable) – дырявая абстракция. Во-первых, ResultSet – разрушающий объект; этим он похож на Iterator, но не на Iterable. Обойти Iterator можно только один раз, и к ResultSet это тоже часто относится. Во-вторых, Iterable является фабрикой новых итераторов, тогда как показанный выше конвертор всегда возвращает Iterator, основанный на одном и том же ResultSet. Это означает, что если дважды вызвать метод iterator(), то мы получим итераторы, возвращающие разные значения, поскольку они конкурируют за один и тот же ResultSet. Наконец, ResultSet необходимо закрывать после использования, а к Iterator такое требование не предъявляется. Полагаться на то, что клиентский код, читающий итератор до конца, произведет заключительную очистку – чрезмерный оптимизм.

Это упрощенный конвертор без обработки ошибок, заимствованный из класса ResultSetIterator, входящего в состав библиотеки с открытым исходным кодом Apache Commons DbUtils⁷. В этом классе есть также упрощенное преобразование к типу Iterable<Object[]>:

⁷ <https://commons.apache.org/proper/commons-dbutils/apidocs/org/apache/commons/dbutils/ResultSetIterator.html>.

```
public static Iterable<Object[]> iterable(final ResultSet rs) {
    return new Iterable<Object[]>() {

        @Override
        public Iterator<Object[]> iterator() {
            return new ResultSetIterator(rs);
        }
    };
}
```

Имея эти конверторы, мы наконец можем построить `Observable<Object[]>`, обертывающий `ResultSet` и поддерживающий противодавление:

```
Connection connection = //...
PreparedStatement statement =
    connection.prepareStatement("SELECT ...");
statement.setFetchSize(1000);
ResultSet rs = statement.executeQuery();
Observable<Object[]> result =
    Observable
        .from(ResultSetIterator.iterable(rs))
        .doAfterTerminate(() -> {
            try {
                rs.close();
                statement.close();
                connection.close();
            } catch (SQLException e) {
                log.warn("Ошибка при закрытии", e);
            }
        });
```

Объект `result` поддерживает противодавление изначально, потому что его поддерживает встроенный оператор `from()`. Поэтому пропускная способность подписчика больше не важна, и исключение `MissingBackpressureException` пропало. Отметим, что вызов `setFetchSize()` обязателен; иначе некоторые JDBC-драйверы могут попытаться загрузить все записи в память, что совершенно неэффективно, если требуется надстроить поток над большим результирующим набором.

Как уже отмечалось, на нижнем уровне поддержки противодавления находится реализация интерфейса `Producer`. Но это чрезвычайно тонкое и чреватое ошибками дело, поэтому был создан вспомогательный класс `SyncOnSubscribe`. Эта реализация `Observable.OnSubscribe` основана на вытягивании, и в нее прозрачно встроена поддержка противодавления. Начнем с простейшего случая: объект `Observable` без состояния — на практике такое редко встретишь. Такой `Observable` не сохраняет никакое состояние между вызовами `onNext()`. Но даже простейшие операторы — `range()` или `just()` — должны помнить, какие элементы уже были порождены. Один из немногих полезных `Observable` без состояния порождает случайные числа:

```
import rx.observables.SyncOnSubscribe;

Observable.OnSubscribe<Double> onSubscribe =
```

```
SyncOnSubscribe.createStateless(
    observer -> observer.onNext(Math.random())
);
```

```
Observable<Double> rand = Observable.create(onSubscribe);
```

Объект `rand` – это обычный `Observable`, который можно преобразовывать, комбинировать и на который можно подписаться. Но под капотом скрывается полноценная поддержка противодействия. Если конечный подписчик или любой оператор в конвейере запросит ограниченное количество событий, то этот `Observable` послушно выполнит приказ. Единственное, что нужно предоставить методу `createStateless()` – лямбда-выражение, которое будет вызываться для каждого запрошенного события; так, если снизу поступает запрос `request(3)`, то это выражение будет вызвано три раза – в предположении, что при каждом вызове порождается только одно событие. Между вызовами не сохраняется никакого контекста (состояния).

Теперь построим оператор с внутренним состоянием. Этот вариант `SyncOnSubscribe` допускает неизменяемую переменную состояния, которая передается между вызовами. Кроме того, каждый вызов должен возвращать новое значение состояния. В качестве примера создадим неограниченный генератор натуральных чисел, начиная с нуля. Такой оператор весьма полезен, если требуется применить `zip()` к последовательности произвольной длины и монотонно возрастающей последовательности натуральных чисел. Оператор `range()` тоже будет работать, но он требует задания верхней границы, что не всегда удобно:

```
Observable.OnSubscribe<Long> onSubscribe =
    SyncOnSubscribe.createStateful(
        () -> 0L,
        (cur, observer) -> {
            observer.onNext(cur);
            return cur + 1;
        }
    );
```

```
Observable<Long> naturals = Observable.create(onSubscribe);
```

На этот раз мы передали два лямбда-выражения фабричному методу `createStateful()`. Первое лениво создает начальное состояние – в данном случае ноль. Второе выражение важнее: предполагается, что оно порождает один элемент, как-то зависящий от текущего состояния, и возвращает новое состояние. Поскольку ожидается, что состояние неизменно, этот метод позволяет вернуть новое состояние, а не изменять текущее. Объект `naturals` легко переписать, так чтобы он возвращал значение типа `BigInteger`, предотвратив тем самым гипотетическое переполнение. Этот `Observable` может порождать бесконечно число возрастающих натуральных чисел, но в полной мере поддерживает противодействие. Следовательно, темп порождения им событий можно настроить под требования подписчиков. Сравните с наивной реализацией, которая несомненно много проще, но «падает», когда подписчики работают медленно:

```
Observable<Long> naturals = Observable.create(subscriber -> {
    long cur = 0;
    while (!subscriber.isUnsubscribed()) {
        System.out.println("Порождено: " + cur);
        subscriber.onNext(cur++);
    }
});
```

Если вы предпочитаете единственную переменную состояния, которая изменяется при обходе (как объект `ResultSet` из `JDBC`), то у `SyncOnSubscribe` найдется метод и на такой случай. Следующий код *не* компилируется из-за контролируемых исключений, но мы хотим сначала продемонстрировать общий принцип:

```
ResultSet resultSet = //...
```

```
Observable.OnSubscribe<Object[]> onSubscribe = SyncOnSubscribe.createSingleState(
    () -> resultSet,
    (rs, observer) -> {
        if (rs.next()) {
            observer.onNext(toArray(rs));
        } else {
            observer.onCompleted();
        }
        observer.onNext(toArray(rs));
    },
    ResultSet::close
);
```

```
Observable<Object[]> records = Observable.create(onSubscribe);
```

Необходимо реализовать три функции обратного вызова.

- Генератор состояния. Это лямбда-выражение вызывается один раз и порождает переменную состояния, которая будет передаваться в качестве аргумента последующим выражениям.
- Генератор следующего значения, обычно на основе состояния. Этот обратный вызов вправе изменять состояние, переданное в первом аргументе.
- Третья функция обратного вызова вызывается в момент отписки. Именно здесь можно было бы закрыть `ResultSet`.

Ниже приведена более полная реализация, включающая обработку ошибок. Отметим, что ошибки, возникающие в момент отписки, очень трудно передать вниз.

```
Observable.OnSubscribe<Object[]> onSubscribe = SyncOnSubscribe.createSingleState(
    () -> resultSet,
    (rs, observer) -> {
        try {
            rs.next();
            observer.onNext(toArray(rs));
        } catch (SQLException e) {
            observer.onError(e);
        }
    }
);
```

```

    },
    rs -> {
      try {
        // Также закрыть Statement, Connection и т. д.
        rs.close();
      } catch (SQLException e) {
        log.warn("Ошибка при закрытии", e);
      }
    }
  );
};

```

`SyncOnSubscribe` — полезное средство, позволяющее писать объект `Observable` с поддержкой противодавления⁸. Он лишь немногим сложнее `Observable.create()`, но преимущества поддержки противодавления, контролируемого подписчиками, трудно переоценить. Старайтесь не использовать оператор `create()` напрямую, отдавая предпочтение встроенным фабрикам типа `from()` или `SyncOnSubscribe`.

Противодавление — на удивление мощный механизм контроля над интенсивностью потока, порождаемого `Observable`, со стороны подписчиков. Конечно, с каналом обратной связи сопряжены некоторые накладные расходы, но достоинства слабо связанных и вместе с тем управляемых производителей и потребителей многократно перевешивают. Противодействие часто сочетается с пакетированием, так что издержки минимальны, но если подписчик действительно медленный (даже в течение коротких промежутков времени), то его медлительность сразу принимается во внимание, и система в целом сохраняет стабильность. Отсутствие поддержки противодавления можно до некоторой степени компенсировать оператором из семейства `onBackpressure*`(), если только это не продолжается слишком долго.

При создании собственных `Observable` уделяйте внимание правильности обработки запросов противодавления. Помните, что вы не можете контролировать пропускную способность подписчиков. Следует также избегать длительной работы в подписчике, перепоручая ее оператору `flatMap()`. Например, можно было бы сохранять события в базе данных внутри метода `subscribe()`:

```
source.subscribe(this::store);
```

Но лучше постарайтесь сделать метод `store` более реактивным (пусть он возвращает объект типа `Observable<UUID>` с идентификатором сохраненной записи), а метод `subscribe()` используйте только для того, чтобы активировать подписку и побочные эффекты:

```

source
  .flatMap(this::store)
  .subscribe(uuid -> log.debug("Сохранено: {}", uuid));

```

Можно пойти еще дальше и собрать `uuid` в пакет, чтобы уменьшить нагрузку на библиотеку протоколирования:

⁸ Если требуется еще более реактивный инструментарий, обратите внимание на класс `AsyncOnSubscribe`, который в принципе похож, но функции обратного вызова, генерирующие следующий элемент для подписчика, могут быть асинхронными.


```
source
  .flatMap(this::store)
  .buffer(100)
  .subscribe(
    hundredUids -> log.debug("Stored: {}", hundredUids))
```

Избежав длительной работы в методе `subscribe()`, мы уменьшаем необходимость в противодавлении, тем не менее, всегда имеет смысл подумать о нем заранее. В документации имеются сведения о том, поддерживает оператор противодавление или нет. Если эта информация опущена, то, скорее всего, противодавление никак не отражается на операторе, как, например, в случае `map()`.

Резюме

Важный урок, который нужно извлечь из данной главы: избегайте порождения событий вручную методом `Observable.create()`. Если необходимо реализовать `Observable` самостоятельно, присмотритесь к многочисленным фабричным методам, которые уже поддерживают противодавление. Кроме того, тщательно анализируйте предметную область; быть может, входящие события можно без ущерба для дела пропускать или собирать в пакеты, чтобы уменьшить общую нагрузку на потребителя.



Глава 7.

Тестирование и отладка

Дочитав до этого места, вы уже, наверное, понимаете основные принципы программирования с применением реактивных расширений. Мы рассмотрели подписку, наиболее употребительные операторы, использование преимуществ RxJava в существующих приложениях и разработку полностью реактивных программных систем. Но чтобы взять все лучшее от реактивного программирования, мы должны копнуть еще глубже. Эта глава посвящена нескольким нетривиальным, но важным аспектам и принципам, в том числе:

- декларативная обработка ошибок, включая повторные попытки (см. раздел «Обработка ошибок»);
- виртуальное время и тестирование (см. раздел «Виртуальное время»);
- мониторинг и отладка потоков `Observable` (см. раздел «Мониторинг и отладка»).

Одного лишь понимания библиотеки или каркаса недостаточно для успешного развертывания в производственной среде. Вышеупомянутые аспекты – важное условие создания стабильных и отказоустойчивых приложений.

Обработка ошибок

В «Реактивном манифесте» (<http://www.reactivemanoifesto.org/>) перечислены четыре характеристики, которыми должны обладать реактивные системы: отзывчивость, отказоустойчивость, эластичность и управляемость сообщениями. Рассмотрим две из них.

Отзывчивость

Система отвечает своевременно, если дать ответ вообще возможно. [...] Отзывчивость означает, что проблемы обнаруживаются быстро и разрешаются эффективно. [...] малое и относительно постоянное время отклика [...] упрощает обработку ошибок.

Отказоустойчивость

Система остается отзывчивой в условиях отказов. [...] части системы могут выходить из строя и восстанавливаться, но система в целом остается рабо-

тоспособной. [...] На клиента какого-то компонента не возлагается обязанность обрабатывать его отказы.

В этом разделе мы объясним, почему первые две характеристики, отзывчивость и отказоустойчивость, важны, и как RxJava их обеспечивает. Вы уже знакомы с функцией обратного вызова `onError()`, задаваемой в момент подписки на объект `Observable`. Но это лишь верхушка айсберга и зачастую не лучший подход к обработке ошибок.

А где же мои исключения?

Традиционно в Java средством сообщить об ошибке являются исключения. В этом языке есть два вида исключений.

- Неконтролируемые исключения (например, `NullPointerException`), которые необязательно указывать в объявлении метода, хотя это и не запрещается.
- Контролируемые исключения, которые обязательно должны быть объявлены и обработаны, иначе код не откомпилируется. К этому виду относится любой тип `Throwable`, не расширяющий `RuntimeException` или `Error`, например `IOException`.

У обоих видов есть свои плюсы и минусы. Неконтролируемые исключения легко добавить, не нарушая обратную совместимость на этапе компиляции. Кроме того, клиентский код с неконтролируемыми исключениями выглядит чище, т. к. ему не нужно (хотя и можно) заниматься обработкой ошибкой. С другой стороны, контролируемые исключения явно показывают, чего можно ожидать от метода. Разумеется, любой метод может возбуждать какие угодно исключения, но контролируемые считаются частью API, и подразумевается, что они должны быть обработаны явно. Хотя контролируемые исключения невозможно не заметить и, с точки зрения написания не содержащего ошибок кода, они кажутся шагом вперед, на практике оказалось, что пользоваться ими неудобно, а понятность кода только ухудшается. Даже официальные Java API переходят на неконтролируемые исключения, например, прежнее (контролируемое) исключение `JMSEException` в версии JMS 2.0 заменено неконтролируемым исключением `JMSRuntimeException`.

В RxJava принят совершенно иной подход. Прежде всего, в стандартном Java исключения можно рассматривать как отдельное измерение в системе типов. У метода существует тип возвращаемого значения и возбуждаемые исключения, и одно из них никак не связано. Метод, открывающий файл `file`, может либо вернуть значение типа `InputStream`, либо возбудить исключение `FileNotFoundException`. Но что, если это исключение не объявлено в сигнатуре метода? А следует ли ожидать еще каких-нибудь исключений? Исключения – это в некотором смысле альтернативный путь выполнения. В их основе лежит идея о том, что ошибки всегда неожиданны и не являются частью нормального хода работы. В RxJava ошибки – это просто еще один вид уведомления. Всякий тип `Observable<T>` описывает последо-

вательность событий типа `T`, которая может (хотя это не обязательно) завершаться уведомлением о завершении или ошибке. А значит, ошибки – неявная часть любого потока, и, хотя мы не обязаны их обрабатывать, есть много операторов, которые элегантно описывают декларативный способ обработки. Кроме того, явный блок `try-catch`, окружающий `Observable`, не перехватывает никакие ошибки, они распространяются только с помощью уведомлений.

Но прежде чем приступить к изучению операторов RxJava для декларативной обработки ошибок, мы должны понять эвристики, применяемые, когда ошибки не обрабатываются вовсе. В Java исключение может произойти почти где угодно, и разработчики библиотек должны либо обрабатывать их, либо, по крайней мере, как-то о них сообщать. Самая распространенная проблема – метод `subscribe()`, в котором не определен обратный вызов `onError()`:

```
Observable
    .create(subscriber -> {
        try {
            subscriber.onNext(1 / 0);
        } catch (Exception e) {
            subscriber.onError(e);
        }
    })
    // ОШИБКА, отсутствует обратный вызов onError()
    .subscribe(System.out::println);
```

В методе `create()` мы сознательно совершили действие, которое приводит к исключению `ArithmeticException`, и вызвали метод подписчика `onError()`. Однако наш подписчик не предоставил реализацию `onError()`. По счастью, RxJava пытается спасти ситуацию, возбудив исключение `OnErrorNotImplementedException`, обертывающее исходное исключение `ArithmeticException`. Но в каком потоке оно будет возбуждено? Трудный вопрос. Если `Observable` синхронный (как в примере выше), то метод `create()` косвенно вызывается в потоке клиента и, значит, там же возбуждается исключение `OnErrorNotImplementedException`. Поэтому поток, в котором был вызван метод `subscribe()`, получит это исключение.

Ситуация усложняется, если мы забудем указать обработчик ошибок, а `Observable` асинхронный. В таком случае поток, в котором был вызван метод `subscribe()`, мог давно завершиться к моменту возбуждения `OnErrorNotImplementedException`. Коли так, исключение возбуждается в том потоке, где следовало бы вызвать `onError()`. Этот поток может принадлежать диспетчеру, указанному в операторе `subscribeOn()` или последнем `observeOn()`. Диспетчер вправе распорядиться неожиданным исключением как ему угодно, обычно он просто выводит трассировку стека в поток стандартного вывода. Это решение далеко от идеала: такие исключения проходят мимо обычного кода протоколирования и в худшем случае вообще остаются незамеченными. Поэтому, когда `subscribe()` прослушивает только значения, но не ошибки, мы можем предположить, что ошибки пропускаются, а это очень плохо. Даже если вы не ожидаете никаких исключений (а такое бывает редко), по крайней мере, включите протоколирование ошибок.

```
private static final Logger log = LoggerFactory.getLogger(My.class);

//....

.subscribe(
    System.out::println,
    throwable -> log.error("That escalated quickly", throwable));
```

Есть много других мест, в которых могут возникать ошибки. Прежде всего, возьмите за правило окружать лямбда-выражение, передаваемое методу `create()`, блоком `try-catch()`:

```
Observable.create(subscriber -> {
    try {
        subscriber.onNext(1 / 0);
    } catch (Exception e) {
        subscriber.onError(e);
    }
});
```

Если вы все-таки забудете о `try-catch` и позволите `create()` возбудить исключение, то RxJava сделает все, что в ее силах, а именно отправит исключение дальше в виде уведомления `onError()`:

```
Observable.create(subscriber -> subscriber.onNext(1 / 0));
```

Два приведенных выше фрагмента семантически эквивалентны. Исключения, возбуждаемые в методе `create()`, перехватываются RxJava и преобразуются в уведомления об ошибках. И все же рекомендуется явно распространять исключения с помощью `subscriber.onError()` всюду, где это возможно. А еще лучше использовать оператор `fromCallable()`:

```
Observable.fromCallable(() -> 1 / 0);
```

Еще исключения могут возникнуть в любом операторе, принимающем пользовательский код, например, в виде лямбда-выражения. Это относится к операторам `map()`, `filter()`, `zip()` и многим, многим другим. Они должны быть готовы не только к входящим уведомлениям об ошибках, но и к исключениям, возбуждаемым пользовательскими функциями отображения или предикатами. В качестве примера рассмотрим следующие некорректные операторы отображения и фильтрации:

```
Observable
    .just(1, 0)
    .map(x -> 10 / x);
```

```
Observable
    .just("Lorem", null, "ipsum")
    .filter(String::isEmpty);
```

В первом случае для некоторых элементов возбуждается уже знакомое исключение `ArithmeticException`, а во втором — `NullPointerException`, когда `filter()`

вызывает предикат. Все лямбда-выражения, передаваемые функциям высшего порядка, в частности `map()` или `filter()`, должны быть чистыми, а возбуждение исключения – это нечистый побочный эффект. RxJava снова делает все возможное, чтобы обработать неожиданные исключения, и ведет себя в точности так, как мы ожидаем. Если какой-то оператор в конвейере возбуждает исключение, то оно преобразуется в уведомление об ошибке и передается дальше. Но не полагайтесь только на усилия RxJava исправить некорректный пользовательский код; если подозреваете, что лямбда-выражение потенциально может возбудить исключение, обрабатывайте его явно с помощью `flatMap()`:

```
Observable
    .just(1, 0)
    .flatMap(x -> (x == 0) ?
        Observable.error(new ArithmeticException("Zero :-(") :
        Observable.just(10 / x)
    );
```

`flatMap()` – очень многогранный оператор; совсем необязательно, что он определяет следующий шаг асинхронного вычисления. `Observable` – это контейнер для значений и ошибок, поэтому если вы хотите декларативно выразить даже совсем простое вычисление, которое может завершиться ошибкой, оберните его объектом `Observable` – это будет правильное решение.

Декларативная замена *try-catch*

Ошибки распространяются по конвейеру `Observable` во многом, как обычные события. Мы уже понимаем, откуда они берутся, теперь надо разобраться, как их декларативно обработать. Обычно мы имеем дело с комбинацией нескольких операторов и объектов `observable` на верхнем уровне. Рассмотрим простой пример конструирования страхового договора по каким-то данным:

```
Observable<Person> person = //...
Observable<InsuranceContract> insurance = //...
Observable<Health> health = person.flatMap(this::checkHealth);
Observable<Income> income = person.flatMap(this::determineIncome);
Observable<Score> score = Observable
    .zip(health, income, (h, i) -> asses(h, i))
    .map(this::translate);
Observable<Agreement> agreement = Observable.zip(
    insurance,
    score.filter(Score::isHigh),
    this::prepare);
Observable<TrackingId> mail = agreement
    .filter(Agreement::postalMailRequired)
    .flatMap(this::print)
    .flatMap(printHouse::deliver);
```

Здесь мы видим несколько шагов бизнес-процесса: загрузка данных о физическом лице `Person`, поиск имеющегося страхового договора `InsuranceContract`,

определение состояния здоровья `Health` и уровня дохода `Income` по `Person` (конкурентное разветвление) и последующее объединение результатов для вычисления и преобразования оценки в баллах `Score`. Наконец, `InsuranceContract` соединяется с `Score` (но лишь если оценка достаточно высока) и производится какая-то пост-обработка, например, отправка письма по электронной почте. Как вы уже знаете, пока никаких вычислений не производилось; мы только объявили операции, но начнутся они не раньше, чем кто-нибудь подпишется. Но что, если какой-то шаг конвейера породит уведомление об ошибке? Никакой обработки ошибок в коде не видно, но ошибки-то все равно распространяются, и это очень удобно.

Все операторы, которые нам до сих пор встречались, работают со значениями, а ошибки полностью игнорируют. И это правильно: обыкновенные операторы преобразуют значения, которые видят, а уведомления о завершении и ошибке пропускают дальше. Следовательно, ошибка, возникшая в любом объекте `Observable`, дойдет до всех расположенных после него подписчиков. И это хорошо, если бизнес-процесс требует успешного завершения абсолютно всех шагов. Но иногда некоторые ошибки можно обработать, подставив значения по умолчанию или обратившись к вспомогательному источнику данных.

Замена ошибки фиксированным значением с помощью `onErrorReturn()`

Простейший оператор обработки ошибок в `RxJava` – `onErrorReturn()`: он заменяет ошибку фиксированным значением:

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .onErrorReturn(error -> Income.no())

//...

private Observable<Income> determineIncome(Person person) {
    return Observable.error(new RuntimeException("Foo"));
}

class Income {
    static Income no() {
        return new Income(0);
    }
}
```

Оператор `onErrorReturn()` не нуждается в особых пояснениях. Пока мимо проходят нормальные события, он не делает ничего. Но увидев уведомление об ошибке, он заменяет его фиксированным значением – в данном случае `Income.no()`. Это текущая и приятная для чтения альтернатива блоку `try-catch`, который императивно возвращает фиксированное значение в части `catch`:

```
try {
    return determineIncome(Person person)
```

```

} catch (Exception e) {
    return Income.no();
}

```

Вы, конечно, заметили, что в этом примере исходное исключение проглатывается, а возвращается только фиксированное значение. Возможно, так и задумано, но в общем случае рекомендуется все-таки хотя бы протоколировать возникающие исключения. Так ведут себя все операторы обработки ошибок в RxJava – если некоторое исключение обрабатывается декларативно, то оно проглатывается. Это нужно обязательно иметь в виду; нет ничего хуже сбоящей системы, в журнале которой нет никакого упоминания об ошибках. Оператору `onErrorReturn()` ошибка передается в качестве аргумента, но мы его радостно проигнорировали. Можно либо запротоколировать исключение внутри `onErrorReturn()`, либо воспользоваться более специализированными диагностическими операторами, которые рассматриваются в разделе «Мониторинг и отладка» ниже. А пока просто запомните, что все операторы обработки ошибок в RxJava оставляют протоколирование исключений и мониторинг под вашу ответственность.

Ленивое вычисление замещающего значения с помощью `onErrorResumeNext()`

Возврат фиксированного значения в операторе `onErrorReturn()` иногда можно считать хорошим решением, но чаще требуется лениво вычислить какое-то значение, замещающее ошибку. Есть два возможных сценария.

- Основной путь генерации потока данных завершился с ошибкой (пришло уведомление `onError()`), поэтому мы переключаемся на вспомогательный источник, который ничем не хуже, но по какой-то причине мы трактуем его как резервный (быть может, он медленнее, дороже или еще что-то).
- В случае ошибки мы хотели бы заменить реальные данные менее дорогостоящей, более стабильной, но, возможно, устаревшей информацией. Например, если извлечь актуальные данные не получилось, мы можем взять устаревшие из кэша. Другой распространенный пример – немного ухудшить условия работы пользователя, скажем, вместо персональных рекомендаций предложить посетителю Интернет-магазина глобальный список бестселлеров.

Понятно, что обработка ошибки сама по себе может быть накладной процедурой, которая также может завершиться ошибкой. Поэтому необходимо как-то инкапсулировать эту логику в ленивую и предпочтительно асинхронную обертку. И что бы это могло быть? Ну, разумеется, `Observable`!

```

Observable<Person> person = //...
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .onErrorResumeNext(person.flatMap(this::guessIncome));

//...

private Observable<Income> guessIncome(Person person) {

```



```
//...
}
```

Оператор `onErrorResumeNext()` заменяет уведомление об ошибке другим потоком. Если мы подпишемся на `Observable`, охраняемый оператором `onErrorResumeNext()` в случае ошибки, то `RxJava` автоматически переключится с главного `Observable` на резервный, переданный в качестве аргумента. В нашем примере, если в потоке `income` возникнет ошибка, то уведомление о ней будет перехвачено, и библиотека сама подпишется на поток `guessIncome()`, предположительно, менее точный, но более надежный, быстрый или дешевый. Интересно, что `onErrorResumeNext()` можно заменить оператором `concatWith()` в предположении, что `determineIncome` всегда порождает ровно одно значение или ошибку:

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .flatMap(
        Observable::just,
        th -> Observable.empty(),
        Observable::empty)
    .concatWith(person.flatMap(this::guessIncome))
    .first();
```

Здесь используется еще не встречавшийся нам вариант оператора `flatMap()`, принимающий три лямбда-выражения вместо одного:

- первый аргумент позволяет заменить каждый элемент входного `Observable` новым `Observable` — именно так оператор `flatMap()` использовался в этой книге раньше;
- второй аргумент заменяет потенциальное уведомление об ошибке другим потоком. В данном случае мы хотим игнорировать входящие ошибки, поэтому просто переключаемся на пустой `Observable`;
- наконец, если входной поток нормально завершится, то мы сможем заменить уведомление о завершении другим потоком.

Оператор `first()` здесь играет решающую роль. Он означает, что мы ждем только первое событие. Если ошибки не было, то мы получим результат `determineIncome`, и `RxJava` так никогда и не подпишется на результат `guessIncome()`. Если же ошибка была, то первый `Observable` не породит никаких событий, поэтому оператор `first()` запросит другой элемент и тогда произойдет подписка на резервный поток, переданный в качестве аргумента оператору `concatWith()`.

Надеюсь, к этому моменту вы уже понимаете, что `concatWith()` в этом примере вообще не нужен; достаточно оператора `flatMap()` в его наиболее сложной форме. Даже оператор `first()` больше не нужен. Проанализируйте следующий код:

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .flatMap(
        Observable::just,
        th -> person.flatMap(this::guessIncome),
        Observable::empty);
```

Здесь есть интересная особенность: мы можем вернуть из обратного вызова `onError()` другой объект `Observable`, зависящий от аргумента `th` типа `Throwable`. Теоретически можно было бы возвращать разные резервные потоки в зависимости от типа исключения или сопровождающего его сообщения. У оператора `onErrorResumeNext()` имеется перегруженный вариант как раз для этой цели:

```
Observable<Income> income = person
    .flatMap(this::determineIncome)
    .onErrorResumeNext(th -> {
        if (th instanceof NullPointerException) {
            return Observable.error(th);
        } else {
            return person.flatMap(this::guessIncome);
        }
    });
```

Хотя возможностей `flatMap()` достаточно для гибкой обработки ошибок, `onErrorResumeNext()` выразительнее и понятнее, поэтому лучше пользоваться им.

Таймаут в случае отсутствия событий

RxJava предоставляет несколько операторов для обработки уведомлений об исключениях от предшествующего объекта `Observable`. Но знаете ли вы, что даже хуже ошибки? Молчание. Когда система, к которой вы подключаетесь, возвращает ошибку, это сравнительно просто предвидеть, обработать, автономно протестировать и т. д. А если `Observable`, на который мы подписались, не отдает вообще ничего, хотя мы ожидали получить результат практически сразу? Такой сценарий гораздо хуже ошибки. Он оказывает влияние на задержку системы в целом и выглядит, как зависание, без каких бы то ни было ключей в журналах.

К счастью, в RxJava имеется встроенный оператор `timeout()`, который прослушивает входной поток `Observable` и следит за тем, сколько времени прошло с момента последнего события или подписки. Если время между двумя последовательными событиями превысило указанную величину, то `timeout()` публикует уведомление об ошибке `TimeoutException`. Чтобы лучше понять, как работает `timeout()`, рассмотрим сначала `Observable`, который порождает всего одно событие спустя некоторое время. Для демонстрации создадим `Observable`, возвращающий событие `Confirmation` через 200 мс. Для моделирования задержки добавим оператор `delay(100, MILLISECONDS)`. Мы хотели бы также смоделировать дополнительную задержку между событием и уведомлением о завершении. Для этого предназначен оператор `empty()`, который обычно завершается немедленно, но при наличии `delay()` ждет указанное время перед отправкой уведомления о завершении. Комбинация обоих потоков выглядит так:

```
Observable<Confirmation> confirmation() {
    Observable<Confirmation> delayBeforeCompletion =
        Observable
            .<Confirmation>empty()
            .delay(200, MILLISECONDS);
```

```
return Observable
    .just(new Confirmation())
    .delay(100, MILLISECONDS)
    .concatWith(delayBeforeCompletion);
}
```

Теперь протестируем оператор `timeout()` в его простейшем варианте:

```
import java.util.concurrent.TimeoutException;

//...

confirmation()
    .timeout(210, MILLISECONDS)
    .forEach(
        System.out::println,
        th -> {
            if ((th instanceof TimeoutException)) {
                System.out.println("Слишком долго");
            } else {
                th.printStackTrace();
            }
        }
    );
```

Таймаут длительностью 210 миллисекунд выбран не случайно. Задержка между подпиской и поступлением экземпляра `Confirmation` составляет ровно 100 мс, это меньше величины таймаута. А задержка между этим событием и уведомлением о завершении равна 200 мс – тоже меньше 210. Следовательно, в этом примере оператор `timeout()` прозрачен, т. е. не влияет на поток сообщений. Но стоит сделать величину таймаута чуть меньше 200 мс (скажем, 190), как он становится виден. Событие `Confirmation` отображается, но вместо уведомления о завершении мы получаем уведомление об ошибке `TimeoutException`. Время до поступления первого события гораздо меньше 200 мс, но задержка между первым событием и вторым (уведомлением о завершении) превысила 190 мс, поэтому вместо него было отправлено уведомление об ошибке. Разумеется, если задать таймаут меньше 100 мс, то мы и первого события не увидим.

Это был простейший сценарий использования `timeout()`; он полезен, когда нужно ограничить время ожидания ответа. Но иногда фиксированная величина таймаута не годится – нужно корректировать ее во время выполнения. Допустим, мы разработали алгоритм прогнозирования солнечных затмений. На выходе он возвращает объект `Observable<LocalDate>` (а что же еще!), содержащий поток будущих дат таких событий. Допустим, что алгоритм вычислительно сложный и будем моделировать это с помощью оператора `interval()` (см. раздел «Хронометраж: операторы `timer()` и `interval()`»), скрепляя фиксированный список дат с медленно прогрессирующим потоком, который генерирует `interval()`. Вызов `interval(500, 50, MILLISECONDS)` означает, что первая дата появляется через 500 мс, а каждая последующая – с интервалом 50 мс. В реальных системах это вполне обычная ситуация: первый элемент ответа приходит с относительно большой задержкой из-

за необходимости установить соединение, выполнить процедуру инициализации сеанса SSL, оптимизировать запрос и т. п. А последующие ответы либо уже готовы, либо их легко получить, поэтому задержка гораздо меньше:

```
Observable<LocalDate> nextSolarEclipse(LocalDate after) {
    return Observable
        .just(
            LocalDate.of(2016, MARCH, 9),
            LocalDate.of(2016, SEPTEMBER, 1),
            LocalDate.of(2017, FEBRUARY, 26),
            LocalDate.of(2017, AUGUST, 21),
            LocalDate.of(2018, FEBRUARY, 15),
            LocalDate.of(2018, JULY, 13),
            LocalDate.of(2018, AUGUST, 11),
            LocalDate.of(2019, JANUARY, 6),
            LocalDate.of(2019, JULY, 2),
            LocalDate.of(2019, DECEMBER, 26))
        .skipWhile(date -> !date.isAfter(after))
        .zipWith(
            Observable.interval(500, 50, MILLISECONDS),
            (date, x) -> date);
}
```

В таких ситуациях задать один фиксированный таймаут проблематично. Для первого события таймаут должен быть пессимистическим, а для последующих – более агрессивным. Как раз на это и рассчитан перегруженный вариант `timeout()`, который принимает две фабрики объектов `Observable`: одна задает таймаут для первого события, вторая – для всех остальных. Пример стоит тысячи слов:

```
nextSolarEclipse(LocalDate.of(2016, SEPTEMBER, 1))
    .timeout(
        () -> Observable.timer(1000, TimeUnit.MILLISECONDS),
        date -> Observable.timer(100, MILLISECONDS))
```

Здесь первый `Observable` порождает ровно одно событие через одну секунду – это допустимая задержка для первого события. Второй `Observable` создается для каждого события, появляющегося в потоке, и допускает тонкую настройку величины таймаута. Отметим, что мы не использовали параметр `date`. Но вполне возможно, что величина таймаута адаптируется к условиям; например, мы можем подождать следующее событие немного дольше, если предыдущее пришло позже, чем обычно. Или наоборот, для каждого последующего события таймаут уменьшается, адаптируясь к производительности подписчика.

Иногда полезно запоминать задержку каждого события, даже если таймаут не устанавливается. Именно это делает оператор `timeInterval()`: он заменяет каждое событие типа `T` событием типа `TimeInterval<T>`, которое инкапсулирует исходное и добавляет информацию о том, сколько времени прошло после предыдущего события (или подписки, если это событие первое):

```
Observable<TimeInterval<LocalDate>> intervals =
    nextSolarEclipse(LocalDate.of(2016, JANUARY, 1))
        .timeInterval();
```

Помимо метода `getValue()`, который возвращает `LocalDate`, в классе `TimeInterval<LocalDate>` имеется также метод `getIntervalInMilliseconds()`, но понять, что он делает, проще на примере результата работы предыдущей программы после подписки. Легко видеть, что первое событие пришло спустя 533 мс, а на каждое последующее потребовалось примерно 50 мс:

```
TimeInterval [intervalInMilliseconds=533, value=2016-03-09]
TimeInterval [intervalInMilliseconds=49, value=2016-09-01]
TimeInterval [intervalInMilliseconds=50, value=2017-02-26]
TimeInterval [intervalInMilliseconds=50, value=2017-08-21]
TimeInterval [intervalInMilliseconds=50, value=2018-02-15]
TimeInterval [intervalInMilliseconds=50, value=2018-07-13]
TimeInterval [intervalInMilliseconds=50, value=2018-08-11]
TimeInterval [intervalInMilliseconds=50, value=2019-01-06]
TimeInterval [intervalInMilliseconds=51, value=2019-07-02]
TimeInterval [intervalInMilliseconds=49, value=2019-12-26]
```

У оператора `timeout()` есть еще один перегруженный вариант, который принимает объект `Observable`, замещающий исходный в случае ошибки. Это поведение очень похоже на оператор `onErrorResumeNext()` (см. раздел «Ленивое вычисление замещающего значения с помощью `onErrorResumeNext()`» выше).

Повтор после ошибки

Уведомление `onError` всегда последнее, после него в потоке не может быть событий. Поэтому если нужно оповестить о не фатальных условиях, не пользуйтесь `onError`. Этот совет мало чем отличается от стандартной рекомендации избегать управления логикой работы программы с помощью исключений. В случае `Observable` стоит подумать об обертывании ошибок событиями специальных типов, которые могут встречаться в потоке неоднократно наряду с обычными событиями. Например, если вы подаете поток результатов сделок, и некоторые сделки могут завершаться неудачно по таким причинам делового характера, как недостаток финансирования, не используйте для них уведомление `onError`. Вместо этого имеет смысл создать абстрактный класс `TransactionResult` и два его конкретных подкласса, представляющих успех и неудачу. Уведомление `onError` в таком потоке означает, что случилась какая-то катастрофическая ошибка, из-за которой порождение новых событий невозможно.

Вместе с тем, `onError` может представлять спорадические отказы внешних компонентов или систем. Как ни странно, простой повтор операции часто приводит к успеху. Возможно, во внешней системе произошел кратковременный всплеск нагрузки, или она приостановила обслуживание на время сборки мусора, или перезапустилась. Повторная попытка – необходимая составная часть надежной отказоустойчивой системы. В `RxJava` имеется полноценная поддержка повтора.

Простейший вариант оператора `retry()` производит повторную подписку на отказавший `Observable` в надежде, что он снова начнет порождать нормальные события, а не ошибки. В учебных целях мы создадим `Observable`, который ведет себя нигде не годно:

```
Observable<String> risky() {
    return Observable.fromCallable(() -> {
        if (Math.random() < 0.1) {
            Thread.sleep((long) (Math.random() * 2000));
            return "OK";
        } else {
            throw new RuntimeException("Transient");
        }
    });
}
```

В 90 процентах случаев подписка на `risky()` заканчивается исключением `RuntimeException`. Если нам все же удастся попасть в ветвь "ок", то вставляется искусственная задержка длительностью от 0 до 2 секунд. Эта рискованная операция послужит нам для демонстрации `retry()`:

```
risky()
    .timeout(1, SECONDS)
    .doOnError(th -> log.warn("Will retry", th))
    .retry()
    .subscribe(log::info);
```

Напомню, что медленная система, вообще говоря, неотличима от не работающей, но часто это даже хуже, потому что мы наблюдаем дополнительную задержку. Добавление таймаутов, иногда даже агрессивных, в сочетании с механизмом повтора крайне желательно – конечно, если повтор не влечет побочных эффектов и повторяемая операция идемпотентна. Оператор `retry()` ведет себя вполне прямолинейно: переправляет дальше всё (события и уведомление о завершении), кроме уведомления об ошибке. А уведомление об ошибке проглатывается (т. е. нигде не протоколируется никакое исключение), поэтому используется обратный вызов `doOnError()` (см раздел «Обратные вызовы `doOn...()`») ниже. Встретив наше эмулированное исключение `RuntimeException` или `TimeoutException`, `retry()` попытается подписаться заново.

Однако имейте в виду, что если `Observable` кэшируется или по какой-то другой причине гарантированно возвращает одну и ту же последовательность элементов, то `retry()` работать не будет:

```
risky().cached().retry() // НЕПРАВИЛЬНО
```

Если `risky()` породил ошибку один раз, то будет порождать ее бесконечно, сколько бы раз мы ни подписывались на него заново. Чтобы обойти эту проблему, мы можем задержать создание `Observable` еще немного, воспользовавшись оператором `defer()`:

```
Observable
    .defer(() -> risky())
    .retry()
```

Даже если `Observable`, возвращенный методом `risky()`, кэшируется, `defer()` вызывает `risky()` несколько раз и, возможно, каждый раз получает новый `Observable`.

Повтор с задержкой и ограниченным числом попыток

Простой оператор `retry()` полезен, но раз за разом повторять подписку немедленно и без ограничения числа попыток опасно. Это может привести к быстрому исчерпанию ресурсов процессора или сети вследствие чрезмерной нагрузки. По существу, метод `retry()` без параметров – это цикл `while` с внутренним блоком `try`, за которым следует пустой `catch`. Прежде всего, следует ограничить число попыток, и этот механизм уже встроен:

```
risky()  
  .timeout(1, SECONDS)  
  .retry(10)
```

Целочисленный параметр `retry()` говорит, сколько раз можно повторять подписку, следовательно, `retry(0)` означает отсутствие повторов. Если входной `Observable` вернет ошибку 10 раз подряд, то последнее увиденное исключение будет передано дальше. Более гибкий вариант `retry()` оставляет решение на наше усмотрение, передавая номер попытки и фактическое исключение:

```
risky()  
  .timeout(1, SECONDS)  
  .retry((attempt, e) ->  
    attempt <= 10 && !(e instanceof TimeoutException))
```

Этот код не только ограничивает число повторных попыток значением 10, но и прекращает попытки раньше, если произошла ошибка `TimeoutException`.

Если ошибки спорадические, то имеет смысл немного подождать перед повторной попыткой. Сам оператор `retry()` не предоставляет такой возможности, но ее сравнительно несложно реализовать. Есть также более общий оператор `retryWhen()`, который принимает функцию, которой передается объект `Observable`, представляющий поток отказов. Всякий раз как на входе появляется ошибка, этот `Observable` порождает объект типа `Throwable`. На нас возлагается обязанность преобразовать этот `Observable` таким образом, чтобы он порождал некоторое произвольное событие, когда мы хотим произвести повторную попытку:

```
risky()  
  .timeout(1, SECONDS)  
  .retryWhen(failures -> failures.delay(1, SECONDS))
```

В этом примере `retryWhen()` получает `Observable`, порождающий `Throwable` при каждой ошибке во входном потоке. Мы просто задерживаем это событие на одну секунду, так чтобы в выходном потоке оно появилось на секунду позже. Для `retryWhen()` это служит сигналом повторить попытку. Если бы мы просто возвращали тот поток, который получаем (`retryWhen(x -> x)`), то `retryWhen()` вел бы себя в точности, как `retry()`, т. е. повторно подписывался сразу после ошибки. Благодаря `retryWhen()` мы также можем имитировать `retry(10)` (впрочем, не совсем... читайте дальше):

```
.retryWhen(failures -> failures.take(10))
```

Мы получаем событие при каждой ошибке. Предполагается, что возвращаемый поток порождает произвольное событие, когда нужно повторить попытку. Поэтому мы просто переправляем первые 10 ошибок, так что после каждой сразу же следует повторная попытка. Но что случится, когда в потоке `failures` появится однанадцатая ошибка? Вот тут-то и начинается самое интересное. Оператор `take(10)` порождает событие `onComplete` сразу после десятой ошибки. Следовательно, после десятой попытки `retryWhen()` получает событие завершения. Оно интерпретируется как сигнал прекратить повторы и завершить поток. Это означает, что после 10 неудачных попыток мы больше ничего не порождаем и завершаемся. Однако если мы завершим `Observable`, возвращаемый внутри `retryWhen()`, с ошибкой, то эта ошибка будет передана дальше.

Иными словами, пока мы порождаем какие-то события в потоке `Observable` внутри `retryWhen()`, они интерпретируются как запросы повтора. Но если мы отправим уведомление о завершении или об ошибке, то повторные попытки прекращаются, а само уведомление передается дальше. Сам оператор `failures.take(10)` действительно приведет к 10 повторам, но в случае еще одной ошибки дальше будет передана не она, а уведомление об успешном завершении. Взгляните на следующий код:

```
static final int ATTEMPTS = 11;

//...

.retryWhen(failures -> failures
    .zipWith(Observable.range(1, ATTEMPTS), (err, attempt) ->
        attempt < ATTEMPTS ?
            Observable.timer(1, SECONDS) :
            Observable.error(err))
    .flatMap(x -> x)
)
```

Оно выглядит довольно сложно, но делает то, что нужно. Мы скрепляем поток `failures` с последовательностью чисел от 1 до 11. Мы хотели бы выполнить не более 10 попыток, поэтому если порядковый номер попытки меньше 11, то возвращается `timer(1, SECONDS)`. Оператор `retryWhen()` запоминает это событие и повторяет попытку спустя секунду после ошибки. Но если ошибкой заканчивается десятая попытка, то мы возвращаем объект `Observable`, содержащий эту ошибку, что прекращает попытки и передает дальше последнее увиденное исключение.

В результате мы получаем очень гибкий механизм. Мы можем прекратить повтор, увидев определенное исключение или по исчерпанию заранее заданного числа попыток. Более того, мы можем изменять время между повторными попытками! Например, первый повтор производится немедленно, а задержка между последующими повторами возрастает экспоненциально¹:

```
.retryWhen(failures -> failures
    .zipWith(Observable.range(1, ATTEMPTS),
```

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


```

        this::handleRetryAttempt)
    .flatMap(x -> x)
)

//...

Observable<Long> handleRetryAttempt(Throwable err, int attempt) {
    switch (attempt) {
        case 1:
            return Observable.just(42L);
        case ATTEMPTS:
            return Observable.error(err);
        default:
            long expDelay = (long) Math.pow(2, attempt - 2);
            return Observable.timer(expDelay, SECONDS);
    }
}

```

При первой ошибке мы возвращаем `Observable`, который сразу же порождает некоторое событие, поэтому повторная попытка производится немедленно. Тип и значение события не важны (роль играет только момент его возникновения), поэтому число 42 ничуть не хуже любого другого значения. Последняя попытка приводит к отправке подписчику исключения, содержащего последнюю увиденную ошибку. Наконец, для попыток с номерами от 2 до 10 мы вычисляем задержку по такой формуле:

$$delay(attempt) = \begin{cases} 0 & \text{если } attempt = 1 \\ 2^{attempt-2} & \text{если } attempt \in \{2, 3, 4 \dots 10\} \end{cases}$$

Тестирование и отладка

Композиция потоков, особенно с включением времени, может оказаться трудным делом. По счастью, в RxJava имеется развитая поддержка автономного тестирования. Мы можем воспользоваться классом `TestSubscriber` для утверждений о порожденных событиях и, что еще важнее, в RxJava есть концепция виртуального времени. По сути дела, мы полностью контролируем течение времени, так что тесты, зависящие от времени, выполняются быстро и предсказуемо.

Виртуальное время

Время – важный фактор почти в любом приложении, и речь здесь не о задержке и времени отклика. Любое событие происходит в какой-то момент, порядок событий важен, задачи планируются на будущее. Поэтому мы тратим бесчисленные часы в поисках ошибок, случающихся только в определенных дни или в определенных часовых поясах. Похоже, не существует общепринятого способа тестирования кода, связанного со временем. Один из подходов – *тестирование на основе свойств* (property-based testing) – подразумевает генерацию сотен тестов (иногда

рандомизированных) для проверки широкого спектра входных аргументов. Например, проверим очень простое свойство: для любой даты, прибавление и последующее вычитание одного месяца дает исходную дату:

```
import spock.lang.Specification
import spock.lang.Unroll

import java.time.LocalDate
import java.time.Month

class PlusMinusMonthSpec extends Specification {

    static final LocalDate START_DATE =
        LocalDate.of(2016, Month.JANUARY, 1)

    @Unroll
    def '#date +/- 1 month gives back the same date'() {
        expect:
            date == date.plusMonths(1).minusMonths(1)
        where:
            date << (0..365).collect {
                day -> START_DATE.plusDays(day)
            }
    }
}
```

Мы воспользовались каркасом Spock (<http://spockframework.org/>), написанным на языке Groovy, чтобы быстро сгенерировать 366 различных тестовых примеров. Код в блоке `expect` выполняется для каждого значения, сгенерированного в блоке `where`. А в блоке `where` мы перебираем целые числа от 0 до 365 и генерируем все возможные даты в диапазоне от 2016-01-01 до 2016-12-31. Утверждение очевидное и простое: если к любой дате прибавить и вычесть один месяц, то должна получиться исходная дата. Тем не менее, 6 из 366 тестов не проходят:

```
date == date.plusMonths(1).minusMonths(1)
|   |   |   |   |
|   |   |   2016-02-29   2016-01-29
|   |   | 2016-01-30
|   false
2016-01-30
```

```
date == date.plusMonths(1).minusMonths(1)
|   |   |   |   |
|   |   |   2016-02-29   2016-01-29
|   |   | 2016-01-31
|   false
```

```
2016-01-31
date == date.plusMonths(1).minusMonths(1)
|   |   |   |   |
|   |   |   2016-04-30   2016-03-30
|   |   | 2016-03-31
```

```
| false  
2016-03-31
```

```
...
```

Уверен, вы сами поймете, для каких еще дат тест не проходит. Я привел этот искусственный пример только для того, чтобы продемонстрировать, насколько сложной может быть предметная область. Но тонкости календаря – не главная причина трудностей при работе со временем в вычислительных системах. RxJava пытается справиться со сложностью конкурентных вычислений, избегая внутреннего состояния и по возможности используя чистые функции. Чистая функция (или оператор) должна явно объявлять все свои входы и выходы. Тогда тестирование намного упрощается. Но зависимость от времени почти всегда скрыта и не бросается в глаза. Всякий раз, употребляя выражения вида `new Date()`, `Instant.now()`, `System.currentTimeMillis()` и т. п., мы вводим зависимость от внешнего значения, изменяющегося со временем. Мы знаем, что зависимость от синглтонов – признак плохого проектирования, особенно с точки зрения тестопригодности. Но получение текущего времени – это, по существу, зависимость от системного синглтона, доступного из любого места.

Один из паттернов, позволяющих сделать зависимость от времени более явной, – использование фиктивных системных часов. Но он требует от программиста строгой дисциплины – весь относящийся к времени код необходимо делегировать специальной службе, которую можно подменить. В Java 8 этот прием формализован путем введения абстракции `clock`, устроенной примерно так:

```
public abstract class Clock {  
  
    public static Clock system(ZoneId zone) { /* ... */ }  
  
    public long millis() {  
        return instant().toEpochMilli();  
    }  
  
    public abstract Instant instant();  
}
```

Интересно, что в RxJava есть очень похожая абстракция, которую мы уже подробно изучали: диспетчеры `Scheduler` (см. раздел «Что такое диспетчер?» главы 4). Вы спросите, как диспетчеры связаны с течением времени? Дело в том, что в RxJava всё происходит либо немедленно, либо планируется на какое-то *время* в будущем. И именно диспетчер контролирует, когда будет выполнена каждая строка кода.

Диспетчеры и автономное тестирование

У таких диспетчеров, как `io()` или `computation()` нет никаких особых возможностей, кроме запуска задач в определенные моменты времени. Но есть один специальный диспетчер `test()`, обладающий двумя интригующими методами:

`advanceTimeBy()` и `advanceTimeTo()`. Эти методы класса `TestScheduler` способны вручную переводить время, в противном случае оно оказывается застывшим. Это значит, что ни одна задача, запланированная на будущее этим диспетчером, никогда не выполнится, если мы не переведем время вперед, когда сочтем удобным.

Для примера рассмотрим последовательность событий во времени:

```
TestScheduler sched = Schedulers.test();
Observable<String> fast = Observable
    .interval(10, MILLISECONDS, sched)
    .map(x -> "F" + x)
    .take(3);
Observable<String> slow = Observable
    .interval(50, MILLISECONDS, sched)
    .map(x -> "S" + x);

Observable<String> stream = Observable.concat(fast, slow);
stream.subscribe(System.out::println);
System.out.println("Подписан");
```

Подписавшись, мы должны увидеть три события F0, F1 и F2, каждому из которых предшествует задержка 10 мс, а вслед за ними бесконечное число событий S0, S1 ..., каждое с задержкой 50 мс. Как протестировать, что мы объединили все потоки, что события появляются в правильном порядке и, самое главное, в правильное время? Ключ – в явном диспетчере `TestScheduler`, который передается всюду, где возможно:

```
TimeUnit.SECONDS.sleep(1);
System.out.println("Спустя одну секунду");
sched.advanceTimeBy(25, MILLISECONDS);

TimeUnit.SECONDS.sleep(1);
System.out.println("Спустя еще одну секунду");
sched.advanceTimeBy(75, MILLISECONDS);

TimeUnit.SECONDS.sleep(1);
System.out.println("...и еще одну");
sched.advanceTimeTo(200, MILLISECONDS);
```

Результат абсолютно предсказуемый и повторяемый, он не зависит от системного времени, кратковременных всплесков нагрузки, пауз на сборку мусора и т. д.:

```
Подписан
Спустя одну секунду
F0
F1
Спустя еще одну секунду
F2
S0
...и еще одну
S1
S2
```

Вот что здесь происходит:

1. После подписки на поток `stream` запуск задачи `f0` планируется через 10 мс. Однако мы воспользовались диспетчером `TestScheduler`, который абсолютно инертен, пока мы вручную не переведем время.
2. Сон в течение одной секунды на самом деле не нужен, его можно опустить. Поскольку `TestScheduler` не зависит от системного времени, вообще никакие события не порождаются. Вызов `sleep` помещен с единственной целью доказать, что `TestScheduler` работает. Если бы это был не `TestScheduler`, а обычный (подразумеваемый по умолчанию) диспетчер, то к этому моменту на консоли уже появилось бы несколько событий.
3. В результате вызова `advanceTimeBy(25ms)` всё, что было запланировано вплоть до 25-й миллисекунды, начинает выполняться. Поэтому на консоли появляются события `f0` (10-ая миллисекунда) и `f1` (20-ая миллисекунда).
4. В течение еще одной секунды сна на консоли не появляется ничего нового; `TestScheduler` игнорирует реальное время. Но вызов `advanceTimeBy(75ms)` (логическое время теперь сдвинулось к сотой миллисекунде) активирует события `f2` (30-ая миллисекунда) и `s0` (80-ая миллисекунда). Больше ничего не происходит.
5. Еще через секунду реального времени мы переводим время к абсолютной отметке 200 мс (`advanceTimeTo(200ms)`, в методе `advanceTimeBy()` используется относительное время). `TestScheduler` понимает, что нужно породить события `s1` (130-ая миллисекунда) и `s2` (180-ая миллисекунда). Никакие другие события не порождаются, даже если мы будем ждать вечно.

Как видим, `TestScheduler` гораздо «умнее» примитивной абстракции фиктивных часов `Clock`. Мало того что мы полностью контролируем текущее время, так еще и можем произвольно откладывать все события. Неприятность в том, что `TestScheduler` необходимо передавать всюду, в любой оператор, у которого имеется необязательный аргумент типа `Scheduler`. Для удобства во всех таких операторах по умолчанию используется диспетчер `computation()`, но, с точки зрения тестопригодности, лучше передавать диспетчер явно. Кроме того, следует подумать о внедрении зависимостей и получать диспетчеры извне.

Но одного `TestScheduler` недостаточно. Он прекрасно работает в тех автономных тестах, для которых предсказуемость – необходимое условие, поскольку спорадические ошибки в тестах очень раздражают. А в главе 8 изучаются инструменты и методы автономного тестирования принципиально асинхронных объектов `Observable`.

Автономное тестирование

Написание тестопригодного кода и наличие основательного набора тестов давно уже считается необходимой составной частью разработки, это никакая не новость. И неважно, пишете вы тесты сначала, следуя методике разработки через тестирование (TDD), или проверяете свой код с помощью интеграционных тестов на более поздней стадии, автоматизированное тестирование – техника, которая

должна войти в плоть и кровь. Поэтому все используемые инструменты (каркасы, библиотеки, платформы) должны поддерживать автоматизированные тесты, и эту характеристику следует обязательно учитывать при выборе технологии. Но оставьте страхи – в RxJava встроена великолепная поддержка автономного тестирования, несмотря на весьма сложную предметную область: асинхронная, событийно-ориентированная архитектура. Явное управление временем в сочетании с упором на чистые функции и композицию функций (уходящую корнями в функциональное программирование) немало способствуют удобству тестирования.

Проверка порожденных событий

Прежде всего, необходимо определить цели тестирования `Observable`. Имея метод, возвращающий объект `Observable`, мы, вероятно, захотим проверить следующее:

- события порождаются в правильном порядке;
- события правильно распространяются;
- результат композиции различных операторов совпадает с ожидаемым;
- события происходят в правильные моменты времени;
- поддерживается противодействие.

И многое другое. Первые два требования просты и не требуют специальной поддержки со стороны RxJava. Нужно лишь собрать все порожденные события и проверить утверждения, пользуясь той из многочисленных библиотек, которая вам больше по душе:

```
import org.junit.Test;
import static org.assertj.core.api.Assertions.assertThat;

@Test
public void shouldApplyConcatMapInOrder() throws Exception {
    List<String> list = Observable
        .range(1, 3)
        .concatMap(x -> Observable.just(x, -x))
        .map(Object::toString)
        .toList()
        .toBlocking()
        .single();

    assertThat(list).containsExactly("1", "-1", "2", "-2", "3", "-3");
}
```

В этом тесте мы преобразуем `Observable<Integer>` в `List<Integer>`, применяя хорошо известную конструкцию `toList() → toBlocking() → single()` (см. раздел «`BlockingObservable`: выход из реактивного мира» главы 4). Обычно объекты `Observable` асинхронны, поэтому, чтобы получить быстрый тест с предсказуемым результатом, такое преобразование необходимо. Использование `BlockingObservable` позволяет также без труда формулировать утверждения об уведомлениях `onError()`. Исключения просто возбуждаются повторно в момент подписки. Отметим, что контролируемые исключения обертываются объектами типа `RuntimeException` – доказать это может только хороший тест:

```
import com.google.common.io.Files;
import static java.nio.charset.StandardCharsets.UTF_8;
import static org.assertj.core.api.Assertions.failBecauseExceptionWasNotThrown;

File file = new File("404.txt");
BlockingObservable<String> fileContents = Observable
    .fromCallable(() -> Files.toString(file, UTF_8))
    .toBlocking();

try {
    fileContents.single();
    failBecauseExceptionWasNotThrown(FileNotFoundException.class);
} catch (RuntimeException expected) {
    assertThat(expected)
        .hasCauseInstanceOf(FileNotFoundException.class);
}
```

Оператор `fromCallable()` удобен, когда нужно лениво создать `Observable`, порождающий не более одного элемента. Он также поддерживает обработку ошибок и протигодавление, поэтому для создания одноэлементных потоков лучше использовать его, а не метод `observable.create()`. Можно использовать еще один вид автономных тестов для проверки нашего понимания различных операторов и их поведения. Например, что в действительности делает оператор `concatMapDelayError()`? Мы, конечно, можем разок его попробовать, но наличие автономного теста, который любой желающий может прочитать и легко понять, — неоспоримое преимущество:

```
import static rx.Observable.fromCallable;

Observable<Notification<Integer>> notifications = Observable
    .just(3, 0, 2, 0, 1, 0)
    .concatMapDelayError(x -> fromCallable(() -> 100 / x))
    .materialize();

List<Notification.Kind> kinds = notifications
    .map(Notification::getKind)
    .toList()
    .toBlocking()
    .single();

assertThat(kinds).containsExactly(OnNext, OnNext, OnNext, OnError);
```

При использовании стандартного оператора `concatMap()` преобразование второго элемента (0) привело бы к ошибке и завершению всего потока. Однако мы видим, что в результирующем потоке четыре элемента: три вызова `OnNext`, а затем `OnError`. Еще одно утверждение показало бы, что в действительности получены значения 33 ($100 / 3$), 50 и 100. Это наглядно объясняет механику `concatMapDelayError()`: если преобразование приводит к ошибке, то ошибка не передается дальше, а работа оператора продолжается. И лишь когда входной поток завершится, мы передаем уведомление `onError`, которое встретилось по дороге. В этом тесте мы не смогли бы преобразовать `Observable` в `List`, потому что это немедленно возбудило

бы исключение. В таких случаях полезен оператор `materialize()`: события любого вида (`onNext`, `onCompleted` и `onError`) обертываются однородными объектами `Notification`. Эти объекты впоследствии можно исследовать, но соответствующий код громоздкий и малопонятный. На помощь приходит класс `TestSubscriber`:

```
Observable<Integer> obs = Observable
    .just(3, 0, 2, 0, 1, 0)
    .concatMapDelayError(x -> Observable.fromCallable(() -> 100 / x));

TestSubscriber<Integer> ts = new TestSubscriber<>();
obs.subscribe(ts);

ts.assertValues(33, 50, 100);
ts.assertError(ArithmeticException.class); //Ошибка (!)
```

Класс `TestSubscriber` совсем простой: он хранит внутри все полученные события и уведомления, чтобы впоследствии их можно было опросить. `TestSubscriber` также предоставляет набор утверждений, весьма полезных в тестах. Нам нужно только создать экземпляр `TestSubscriber`, подписаться на тестируемый `Observable` и исследовать его содержимое. Странно, но показанный выше тест не проходит. Метод `assertError()` дает ошибку, поскольку мы ожидаем, что поток завершится исключением `ArithmeticException`, а на самом деле получаем исключение типа `CompositeException`, агрегирующее все три исключения `ArithmeticException`, встретившиеся по пути. Это еще одна причина, по которой очень полезно изучать поведение операторов путем их выполнения и автоматизированного тестирования.

Класс `TestSubscriber` особенно эффективен в паре с диспетчером `TestScheduler`. Типичный сценарий – чередование утверждений и перевода времени для наблюдения за потоком событий. Допустим, что имеется служба, возвращающая `Observable`. Детали ее реализация несущественны:

```
interface MyService {
    Observable<LocalDate> externalCall();
}
```

Вместо того чтобы мешать в одну кучу разные аспекты поведения, мы решили надстроить над `MyService` декоратор, который добавит функциональность тайм-аута к исходной реализации `MyService`. По причинам, о которых вы уже, наверное, догадываетесь, мы также вынесли наружу диспетчер, которым пользуется оператор `timeout()`:

```
class MyServiceWithTimeout implements MyService {

    private final MyService delegate;
    private final Scheduler scheduler;

    MyServiceWithTimeout(MyService d, Scheduler s) {
        this.delegate = d;
        this.scheduler = s;
    }
}
```



```
}  
  
@Override  
public Observable<LocalDate> externalCall() {  
    return delegate  
        .externalCall()  
        .timeout(1, TimeUnit.SECONDS,  
            Observable.empty(),  
            scheduler);  
}  
}
```

Объект типа `MyServiceWithTimeout` обортывает экземпляр `MyService` и добавляет односекундный таймаут вместе с замещающим потоком. В духе RxJava у каждого класса только одна обязанность, но их можно комбинировать – точно так же, как мы составляем композиции узкоспециализированных операторов. Пусть требуется протестировать, работает ли таймаут. В идеале автономные тесты должны быть очень быстрыми. Помните тест `PlusMinusMonthSpec`, приведенный в начале раздела «Виртуальное время» выше? Его прогон для каждого дня в XXI-ом веке (а их больше 36 тысяч) занимает примерно одну секунду. Хороший тест должен работать не дольше нескольких миллисекунд.

Односекундный таймаут – вроде бы не так уж много, но это целая вечность, когда таких тестов нужно прогнать сотни. Мы можем вынести величину таймаута наружу (это в любом случае разумно) и для автономного тестирования уменьшить, скажем, до 100 мс. В самом тесте мы могли проспать 90 мс, проверить, что таймаут еще не истек, проспать еще 20 мс и затем проверить, что таймер сработал и мы получили пустой `Observable`. К сожалению, это очень хрупкая схема, чувствительная к контекстным переключениям, паузам на сборку мусора, переменной нагрузке на систему и т. д. Короче говоря, тест может быть либо относительно стабильным, либо относительно быстрым. Но чем он быстрее, тем чаще будут случаться спорадические ошибки. «Мигающие» тесты еще хуже, чем отсутствие тестов, потому что к ним нет доверия; в конечном итоге такие тесты удаляют.

В RxJava используются синтетические управляемые часы, стопроцентно предсказуемые. Безукоризненная точность и одновременно высокая скорость тестов достигаются путем искусственного перевода времени. Сначала настроим подставку (mock) для `MyService` (с помощью библиотеки Mockito²), которая сможет вернуть любой `Observable`:

```
import static org.mockito.BDDMockito.given;  
import static org.mockito.Mockito.mock;  
  
private MyServiceWithTimeout mockReturning(  
    Observable<LocalDate> result,  
    TestScheduler testScheduler) {  
    MyService mock = mock(MyService.class);  
    given(mock.externalCall()).willReturn(result);  
    return new MyServiceWithTimeout(mock, testScheduler);  
}
```

² <http://site.mockito.org/>

Теперь напишем два автономных теста. Первый проверяет, что если вызов `externalCall()` никогда не завершается, то ровно через секунду произойдет таймаут:

```
@Test
public void timeoutWhenServiceNeverCompletes() throws Exception {
    //given
    TestScheduler testScheduler = Schedulers.test();
    MyService mock = mockReturning(
        Observable.never(), testScheduler);
    TestSubscriber<LocalDate> ts = new TestSubscriber<>();

    //when
    mock.externalCall().subscribe(ts);

    //then
    testScheduler.advanceTimeBy(950, MILLISECONDS);
    ts.assertNoTerminalEvent();
    testScheduler.advanceTimeBy(100, MILLISECONDS);
    ts.assertCompleted();
    ts.assertNoValues();
}
```

Оператор `never()` возвращает `Observable`, который никогда не завершается и не порождает ни одного значения. Так мы моделируем нестерпимо медленное обращение к `myService`. Затем идут два утверждения. Сначала мы устанавливаем время не задолго до момента истечения таймаута (950 мс) и убеждаемся, что `TestSubscriber` еще не завершился – ни нормально, ни с ошибкой. Спустя еще 100 мс, т. е. уже после таймаута, мы проверяем, что поток завершился (`assertCompleted()`) но при этом не было ни одного значения (`assertNoValues()`). Можно также воспользоваться методом `assertError()`.

Второй тест проверяет, что таймер не срабатывает, пока не пройдет заданное время:

```
@Test
public void valueIsReturnedJustBeforeTimeout() throws Exception {
    //given
    TestScheduler testScheduler = Schedulers.test();
    Observable<LocalDate> slow = Observable
        .timer(950, MILLISECONDS, testScheduler)
        .map(x -> LocalDate.now());
    MyService myService = mockReturning(slow, testScheduler);
    TestSubscriber<LocalDate> ts = new TestSubscriber<>();

    //when
    myService.externalCall().subscribe(ts);

    //then
    testScheduler.advanceTimeBy(930, MILLISECONDS);
    ts.assertNotCompleted();
    ts.assertNoValues();
    testScheduler.advanceTimeBy(50, MILLISECONDS);
}
```

```
ts.assertCompleted();  
ts.assertValueCount(1);  
}
```

Метод `advanceTimeBy()` эквивалентен сну в ожидании события с тем отличием, что на сон не тратится реальное время. Мы можем тестировать самые разные операторы, в т. ч. `buffer()`, `sample()` и т. д., нужно лишь не забывать передавать им подходящий диспетчер. И раз уж речь зашла о диспетчерах, то возникает искушение использовать `Schedulers.immediate()` (см. раздел «Что такое диспетчер?» главы 4) вместо стандартных. Этот диспетчер предотвращает всякую конкурентность, т. к. выполняет все действия в контексте вызывающего потока. В некоторых ситуациях такой подход работает, но, вообще говоря, лучше использовать `TestScheduler`, т. к. поле для его применения гораздо шире.

Очень важно не забывать о внедрении зависимостей. В противном случае нельзя будет подменить различные диспетчеры тестовым. Есть несколько решений этой проблемы, в т. ч. подключаемый модуль `RxJavaSchedulersHook`. Для `RxJava` существует ряд подключаемых модулей, которые глобально изменяют поведение библиотеки. Например, `RxJavaSchedulersHook` умеет подменять стандартный диспетчер `computation()` (и другие) тестовым:

```
private final TestScheduler testScheduler = new TestScheduler();  
  
@Before  
public void alwaysUseTestScheduler() {  
    RxJavaPlugins  
        .getInstance()  
        .registerSchedulersHook(new RxJavaSchedulersHook() {  
            @Override  
            public Scheduler getComputationScheduler() {  
                return testScheduler;  
            }  
  
            @Override  
            public Scheduler getIOScheduler() {  
                return testScheduler;  
            }  
  
            @Override  
            public Scheduler getNewThreadScheduler() {  
                return testScheduler;  
            }  
        });  
}
```

У такого глобального подхода много недостатков. Мы можем зарегистрировать `RxJavaSchedulersHook` только один раз в пределах JVM, поэтому попытка использовать аннотацию `@Before` вторично закончится печально. Это ограничение можно обойти, но тогда код становится слишком сложным. Кроме того, теряется возможность запускать тесты параллельно (обычно автономные тесты независимы, поэтому такой проблемы не возникает). Поэтому единственное масштабируемое

решение для управления временем – явная передача `TestScheduler` всюду, где возможно.

И последнее, что можно протестировать с помощью `TestSubscriber`, – противодействие. В разделе «Учет запрошенного объема данных» главы 6 мы рассмотрели две реализации бесконечного потока `Observable`, порождающего последовательность натуральных чисел. Первый – старомодное использование метода `Observable.create()` – не поддерживает противодействие:

```
Observable<Long> naturals1() {
    return Observable.create(subscriber -> {
        long i = 0;
        while (!subscriber.isUnsubscribed()) {
            subscriber.onNext(i++);
        }
    });
}
```

Рекомендуемая, хотя и более сложная реализация, поддерживает противодействие в полной мере:

```
Observable<Long> naturals2() {
    return Observable.create(
        SyncOnSubscribe.createStateful(
            () -> 0L,
            (cur, observer) -> {
                observer.onNext(cur);
                return cur + 1;
            }
        )
    );
}
```

С точки зрения функциональности, эти решения эквивалентны, оба порождают бесконечный поток, но мы можем выбрать какую-то часть. Однако `TestSubscriber` позволяет еще и проверить, поддерживает ли данный `Observable` противодействие:

```
TestSubscriber<Long> ts = new TestSubscriber<>();

naturals1()
    .take(10)
    .subscribe(ts);

ts.assertNoValues();
ts.requestMore(100);
ts.assertValueCount(10);
ts.assertCompleted();
```

Самое важное в этом примере – конструктор `TestSubscriber<>()`. Без него `TestSubscriber` просто получает все события в темпе, задаваемом источником. Но если перед подпиской подавить запрос данных, то `TestSubscriber` не будет запрашивать данные у `Observable`. Именно поэтому утверждение `assertNoValues()` не

дает ошибки, хотя источник, очевидно, породил 10 значений. Затем мы запрашиваем целых 100 элементов (на всякий случай), но Observable отдает только 10 – столько, сколько может породить. Для потока `naturals1` этот тест «падает» почти сразу с таким сообщением:

```
AssertionError: No onNext events expected yet some received: 10
```

Наш наивный Observable, конечно, знает, что нужно породить только 10 событий, хотя источник бесконечный. Оператор `take(10)` энергично отписывается, чем завершает внутренний цикл `while`. Но `naturals1` игнорирует запросы механизма противодействия, которые отправляет `TestSubscriber`, поэтому последний получает элементы, которые вовсе не хотел видеть. Если заменить источник на `naturals2`, то тест пройдет. Это еще одна причина избегать метода `Observable.create()`, предпочитая ему встроенные фабрики и класс `SyncOnSubscribe`.

В классе `TestSubscriber` еще много утверждений. Некоторые из них блокируют выполнение в ожидании завершения, например, `awaitTerminalEvent()`. Но большинство проверяет состояние подписчика в данный момент, так что мы можем наблюдать движение событий во времени.

Мониторинг и отладка

Мониторинг поведения различных потоков, взаимодействующих между собой, и поиск причин возникающих проблем – это трудная тема в RxJava. На самом деле, любую асинхронную событийно-ориентированную систему отлаживать принципиально труднее, чем систему с блокирующей архитектурой. Если в синхронной операции возникает ошибка, то исключение распространяется вверх по стеку вызовов, и мы видим точную последовательность операций, приведших к ошибке, начинающуюся от HTTP-сервера и включающую различные фильтры, аспекты, бизнес-логику и т. д. В асинхронной системе полезность стека вызовов ограничена, поскольку если событие пересекает границу потоков, то исходный стек вызовов нам уже недоступен. То же самое касается распределенных систем. В этом разделе приведено несколько рекомендаций о том, как упростить мониторинг и отладку в системах на базе RxJava.

Обратные вызовы `doOn...()`

У каждого объекта Observable имеется набор методов обратного вызова, позволяющих заглянуть в различные события, а именно:

- `doOnCompleted()`
- `doOnEach()`
- `doOnError()`
- `doOnNext()`
- `doOnRequest()`
- `doOnSubscribe()`

- `doOnTerminate()`
- `doOnUnsubscribe()`

Общего у них то, что им запрещено хоть как-то изменять состояние `Observable` и что все они возвращают тот же самый `Observable`, а, значит, являются идеальным местом для размещения протоколирования. Например, начинающие часто забывают, что код, переданный методу `Observable.create()`, выполняется для каждого подписчика. Это важно, если подписка влечет за собой побочные эффекты, скажем, обращение к сети. Чтобы выявить такие проблемы, рекомендуется протоколировать любую подписку на критически важный источник:

```
Observable<Instant> timestamps = Observable
    .fromCallable(() -> dbQuery())
    .doOnSubscribe(() -> log.info("subscribe()"));

timestamps
    .zipWith(timestamps.skip(1), Duration::between)
    .map(Object::toString)
    .subscribe(log::info);
```

Эта программа обращается с запросом к базе данных (`dbQuery()`) и получает какие-то временные ряды в виде объекта `Observable<Instant>`. Мы хотели бы немного преобразовать этот поток, вычислив промежутки времени (с помощью класса `Duration` из пакета `java.time`) между последовательными событиями `Instant`: первым и вторым, вторым и третьим и т. д. Один из способов – скрепить оператором исходный поток с самим собой, но сдвинутым на один элемент. Тогда мы соединим первый элемент со вторым, второй с третьим и т. д. до самого конца. Вот только мы забыли, что оператор `zipWith()` подписывается на все переданные ему потоки, т. е. на поток `timestamps` он подпишется дважды. Это можно обнаружить, заметив, что метод `doOnSubscribe()` вызван два раза. А приводит это к повторному запросу к базе данных – проблеме, которую мы всесторонне обсудили в главе 2.

И кстати о `zip()`: благодаря противодействию он больше не буферизует более быстрый поток бесконечно, ожидая, пока более медленный породит события. Вместо этого он запрашивает у каждого `Observable` фиксированную порцию значений и возбуждает исключение `MissingBackpressureException`, если получает больше:

```
.doOnSubscribe(() -> log.info("subscribe()"))
.doOnRequest(c -> log.info("Запрошено {}", c))
.doOnNext(instant -> log.info("Получено: {}", instant));
```

Оператор `doOnRequest()` выводит в журнал Запрошено 128, такое значение выбрал оператор `zip`. Даже если поток `Observable` бесконечный или очень большой, но ведет себя корректно, мы должны увидеть в последующем сообщении Получено... не более 128 элементов. `doOnNext()` – еще один метод обратного вызова, который может пригодиться. Также довольно часто встречается оператор `doOnError()`. Он вызывает переданную функцию обратного вызова при каждом появлении входящего уведомления об ошибке. Использовать `doOnError()` для обработки ошибок

нельзя, он предназначен только для протоколирования. Он не глотает уведомление, а отправляет его дальше:

```
Observable<String> obs = Observable
    .<String>error(new RuntimeException("Проглочено"))
    .doOnError(th -> log.warn("onError", th))
    .onErrorReturn(th -> "Замещение");
```

Оператор `onErrorReturn()` выглядит чисто, но глотать исключение в нем очень легко. Он передает исключение, которое мы хотим заместить другим значением, но протоколирование возлагается на нас. Чтобы функции оставались небольшими и допускающими композицию, мы сначала протоколируем ошибку в `doOnError()`, а затем молча обрабатываем исключение в следующей строке. Так получается немного надежнее. Пренебрежение протоколированием исключений редко бывает оправданным, и это решение следует принимать осознанно, а не по недосмотру.

Прочие операторы почти не нуждаются в пояснениях, за исключением двух.

`doOnEach()`

Вызывается для каждого уведомления `Notification`, т. е. для `onNext()`, `onCompleted()` и `onError()`. Может принимать либо лямбда-выражение, вызываемое для уведомления, либо объект типа `Observer`.

`doOnTerminate()`

Вызывается для `onCompleted()` и `onError()`. Отличить одно от другого невозможно, поэтому лучше использовать `doOnCompleted()` и `doOnError()` независимо.

Измерение и мониторинг

Обратные вызовы полезны не только для протоколирования. Различные телеметрические зонды, встроенные в приложение (простые счетчики, таймеры, гистограммы распределения и т. д.) и доступные извне, могут существенно сократить время поиска ошибки, а заодно пролить свет на работу приложения. Существует много библиотек, упрощающих сбор и публикацию метрик, одна из них – Metrics компании Dropwizard (<http://metrics.dropwizard.io/3.1.0/>). Чтобы ей воспользоваться, необходимо кое-что подготовить:

```
import com.codahale.metrics.MetricRegistry;
import com.codahale.metrics.Slf4jReporter;
import org.slf4j.LoggerFactory;

MetricRegistry metricRegistry = new MetricRegistry();
Slf4jReporter reporter = Slf4jReporter
    .forRegistry(metricRegistry)
    .outputTo(LoggerFactory.getLogger(SomeClass.class))
    .build();
reporter.start(1, TimeUnit.SECONDS);
```

Класс `MetricRegistry` – фабрика различных метрик. Кроме того, мы настраиваем генератор отчетов `Slf4jReporter`, который будет записывать снимок статисти-

ческих данных в переданный объект протоколирования SLF4J. Имеются также генераторы отчетов для систем Graphite (<http://graphite.readthedocs.io/en/latest/>) и Ganglia (<http://ganglia.info/>). После этой несложной настройки можно приступить к мониторингу потоков.

Одна из простейших метрик – счетчик `Counter`, который можно увеличивать и уменьшать. Им можно воспользоваться для подсчета числа событий в потоке:

```
final Counter items = metricRegistry.counter("items");
observable
    .doOnNext(x -> items.inc())
    .subscribe(...);
```

После подписки на этот `Observable` в объекте `items` будет храниться количество уже порожденных элементов. Полезность этой информации возрастет, если опубликовать ее на внешнем сервере мониторинга типа Graphite и строить график зависимости от времени.

Еще одна интересная метрика – сколько элементов одновременно обрабатывается в данный момент. Например, оператор `flatMap()` легко может соединить сотню и более конкурентных потоков `Observable` и подписаться сразу на все. Зная количество активных `Observable` (соединений с базой данных, веб-сокетов и т. д.), можно многое сказать о работе системы:

```
Observable<Long> makeNetworkCall(long x) {
    //...
}

Counter counter = metricRegistry.counter("counter");
observable
    .doOnNext(x -> counter.inc())
    .flatMap(this::makeNetworkCall)
    .doOnNext(x -> counter.dec())
    .subscribe(...);
```

При появлении входящего события мы увеличиваем счетчик. Если событие появилось после `flatMap()` (значит, какая-то асинхронная операция только что породила что-то), мы уменьшаем счетчик. В простаивающей системе счетчик всегда равен 0, но если входной `observable` порождает много событий, а функция `makeNetworkCall()` относительно медленная, то этот счетчик стремительно растет, и мы сразу понимаем, где «затык».

В этом примере предполагается, что `makeNetworkCall()` всегда возвращает ровно один элемент и никогда не завершается ошибкой (не вызывает `onError()`). Если же мы хотим измерить время между моментом подписки на внутренний `Observable` (когда работа началась) и моментом его завершения, то сделать это ничуть не сложнее:

```
observable
    .flatMap(x ->
        makeNetworkCall(x)
```



```
.doOnSubscribe(counter::inc)
.doOnTerminate(counter::dec)
)
.subscribe(...);
```

Timer — более сложная метрика, она измеряет промежуток времени между двумя моментами. Ценность этой метрики невозможно переоценить: мы можем измерять сетевую задержку, время выполнения запроса к базе данных, время реакции на действия пользователя и многое другое. Для измерения времени мы обычно запоминаем текущее время, выполняем какую-то длительную операцию, а затем вычисляем, сколько прошло времени между «тогда» и «сейчас». Эту последовательность действий библиотека Metrics инкапсулирует следующим образом:

```
import com.codahale.metrics.Timer;

Timer timer = metricRegistry.timer("timer");
Timer.Context ctx = timer.time();
// длительная операция...
ctx.stop();
```

API хранит время начала операции в объекте `Timer.Context` и предполагает, что хронометрируемый код блокирующий. А если мы хотим измерить время между подпиской на `Observable`, который не контролируем, и его завершением? Операторы `doOnSubscribe()` и `doOnTerminate()` здесь недостаточно, потому что мы не можем передать между ними `Timer.Context`. К счастью, RxJava обладает достаточной гибкостью для решения этой проблемы путем добавления еще одного уровня композиции:

```
Observable<Long> external = //...

Timer timer = metricRegistry.timer("timer");

Observable<Long> externalWithTimer = Observable
    .defer(() -> Observable.just(timer.time()))
    .flatMap(timerCtx ->
        external.doOnCompleted(timerCtx::stop));
```

Мы применили нехитрый трюк. Сначала мы лениво начинаем отсчет времени с помощью оператора `defer()`. В результате таймер запустится точно в момент подписки. Затем мы в каком-то смысле подменяем экземпляр `Timer.Context` тем объектом `Observable`, который хотим хронометрировать (`external`). Но прежде чем вернуть `external`, мы останавливаем работающий таймер. Эту технику можно использовать для измерения времени между подпиской и завершением любого `observable`, над которым у нас нет контроля.

Если хотите узнать о более полных решениях в части мониторинга, готовых для использования в корпоративных приложениях, поинтересуйтесь библиотекой `Hystrix`, основанной на RxJava. Мы рассмотрим ее, наряду с другими примерами, в главе 8 (см. раздел «Управление отказами с помощью `Hystrix`»).

Резюме

Любая реактивная библиотека или каркас в силу своей асинхронной и событийно-ориентированной природы представляет сложности для отладки ошибок. В этом смысле RxJava – не исключение, но она предоставляет инструменты, облегчающие жизнь разработчиков и эксплуатационников.

- Во-первых, RxJava перехватывает ошибки и упрощает их обработку.
- Во-вторых, она предоставляет средства для мониторинга и отладки потоков в реальном времени.
- Наконец, она располагает первоклассной поддержкой автономного тестирования.

Возможность получить полный контроль над системными часами крайне полезна для отладки зависящих от времени операторов. Поначалу отладка RxJava может показаться трудной. Но она предлагает ясный API и строгий контракт – в противоположность якобы более простому блокирующему коду, для которого характерны скрытые состояния гонки и низкая пропускная способность.



Глава 8.

Практические примеры

В этой главе приведено несколько примеров использования RxJava в реальных приложениях. API реактивных расширений обладает богатыми возможностями, но где-то должен присутствовать источник типа `Observable`. Создать `Observable` с нуля не так-то просто из-за необходимости поддержать противодействие и контракт Rx. Но есть и хорошие новости: существует много библиотек и каркасов, поддерживающих RxJava изначально. Кроме того, RxJava весьма полезна на некоторых по сути своей асинхронных платформах.

В этой главе мы увидим, как RxJava позволяет улучшить дизайн и расширить возможности существующих архитектур. Мы рассмотрим также более сложные проблемы, которые могут возникать при развертывании реактивных приложений в производственной системе, в частности, утечки памяти. Прочитав эту главу, вы убедитесь, что RxJava – достаточно зрелый и многогранный продукт для применения в различных ситуациях, возникающих в современных приложениях.

Применение RxJava в разработке программ для Android

RxJava очень популярна у разработчиков для Android. Во-первых, графические интерфейсы изначально являются событийно-ориентированными, причем события поступают из различных источников, в т. ч. от клавиатуры и мыши. Во-вторых, Android, равно как Swing и многие другие графические среды, очень строго относятся к использованию потоков. Главный поток Android не должен блокироваться, т. к. это приводит к зависанию пользовательского интерфейса, однако любое обновление пользовательского интерфейса должно производиться из главного потока. Эти проблемы будут рассмотрены в разделе «Диспетчеры в Android». Но самый главный вопрос, о котором нужно помнить при использовании RxJava в Android, описан в следующем разделе – это утечки памяти и их предотвращение.

Предотвращение утечек памяти в компонентах Activity

Один из подводных камней, специфичных для Android, – утечки памяти в компонентах Activity (операциях). Это происходит, когда объект Observer хранит сильную ссылку на какой-то компонент пользовательского интерфейса (GUI), который, в свою очередь, ссылается на экземпляр родительской Activity. При повороте экрана мобильного устройства или нажатии кнопки возврата ОС Android уничтожает текущий экземпляр Activity и в конечном итоге пытается убрать его в мусор. Объекты Activity довольно велики, поэтому важно освобождать их как можно скорее. Но если ваш Observer хранит ссылку на Activity, то убрать его в мусор не получится, а это приводит к утечке памяти и, как результат, к снятию всего приложения операционной системой. Рассмотрим такой якобы безобидный код:

```
public class MainActivity extends AppCompatActivity {

    private final byte[] blob = new byte[32 * 1024 * 1024];

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        TextView text = (TextView) findViewById(R.id.textView);
        Observable
            .interval(100, TimeUnit.MILLISECONDS)
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(x -> {
                text.setText(Long.toString(x));
            });
    }
}
```

Поле `blob` нужно только для того, чтобы эффект утечки памяти проявился побыстрее; можете считать, что `MainActivity` – на самом деле, сложное дерево объектов. Это простое приложение кажется совершенно нормальным, но это только на первый взгляд. Каждые 100 мс оно обновляет текстовое поле, записывая в него текущее значение счетчика. Однако если дважды повернуть устройство, то приложение «грохается» с ошибкой `OutOfMemoryError`. Происходит вот что:

1. Создается объект `MainActivity`, и в методе `onCreate()` мы подписываемся на `interval()`.
2. Каждые 100 миллисекунд мы записываем в текстовое поле текущее значение счетчика. Не обращайте пока внимания на диспетчер `mainThread()`, мы опишем его в разделе «Диспетчеры в Android» ниже.
3. Ориентация устройства изменяется.
4. Объект `MainActivity` уничтожается и создается новый, метод `onCreate()` выполняется снова.
5. Теперь работает два экземпляра `Observable.interval()`, потому что мы так и не отписались от первого.

Но тот факт, что одновременно работают два интервала, причем первый остался от уже уничтоженного объекта `Activity`, — еще не самое страшное. Оператор `interval()` пользуется фоновым потоком (выделенным ему диспетчером `computation()`) для порождения событий счетчика. Эти события передаются объектам `Observer`, один из которых хранит ссылку на `TextView`, который, в свою очередь, хранит ссылку на старый объект `MainActivity`. Поток, порождающий события `interval()`, становится новым корнем GC; это значит, что ни один объект, на который он ссылается прямо или косвенно, не может быть убран в мусор. Следовательно, несмотря на то, что первый экземпляр `MainActivity` уничтожен, в мусор он не попадает, и память, выделенная для `blob`, не освобождается. Каждое изменение ориентации устройства (или любая другая ситуация, в которой Android решает уничтожить экземпляр `Activity`) увеличивает утечку памяти. Решение простое: сообщить оператору `interval()`, что он больше не нужен, отписавшись от него (см. раздел «Управление прослушивателями с помощью типов `Subscription` и `Subscriber<T>`» главы 2). В Android существует метод, подобный `onCreate()`, но вызываемый при уничтожении объектов, — `onDestroy()`:

```
private Subscription subscription;

@Override
protected void onCreate(Bundle savedInstanceState) {
    //...
    subscription = Observable
        .interval(100, TimeUnit.MILLISECONDS)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(x -> {
            text.setText(Long.toString(x));
        });
}

@Override
protected void onDestroy() {
    super.onDestroy();
    subscription.unsubscribe();
}
```

Вот и всё. Если созданный `Observable` принимает участие в жизненном цикле объекта `Activity`, не забывайте отписываться от него при уничтожении этого объекта. Вызов `unsubscribe()` разрывает связь между `Observer` и `Observable`, делая первый доступным сборщику мусора. А вместе с `Observer` можно будет убрать в мусор и весь объект `MainActivity`. Кроме того, сам оператор `interval()` перестает порождать события, потому что его никто не слушает. Двойной выигрыш.

Если вместе с `Activity` создается много объектов `Observable`, то хранить ссылки на все подписки утомительно. Для таких случаев есть удобный контейнер `CompositeSubscription`. Каждый объект `Subscription` можно поместить в `CompositeSubscription`, а в момент уничтожения отписаться ото всех сразу одним движением:

```
private CompositeSubscription allSubscriptions = new CompositeSubscription();

@Override
protected void onCreate(Bundle savedInstanceState) {
    //...
    Subscription subscription = Observable
        .interval(100, TimeUnit.MILLISECONDS)
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(x -> {
            text.setText(Long.toString(x));
        });
    allSubscriptions.add(subscription);
}

@Override
protected void onDestroy() {
    super.onDestroy();
    allSubscriptions.unsubscribe();
}
```

Стоит отметить, что отписываться от уже не используемых `Observable` следует в любом окружении. Но в мобильных устройствах, где ресурсы ограничены, это особенно важно. Теперь, зная о подводных камнях при управлении памятью на платформе Android, мы можем по-новому подойти к проектированию мобильных приложений. Прежде всего, изучим Retrofit – HTTP-клиент с встроенной поддержкой RxJava, особенно популярный на мобильных устройствах.

Библиотека Retrofit со встроенной поддержкой RxJava

Retrofit (<http://square.github.io/retrofit/>) – библиотека для отправки HTTP-запросов, особенно популярная в экосистеме Android. Она не написана специально для Android и не является единственным доступным HTTP-клиентом. Но благодаря встроенной поддержке RxJava она отлично подходит для мобильных приложений – как изначально спроектированных под RxJava, так и просто нуждающихся во взаимодействии по протоколу HTTP. Основное преимущество использования RxJava в сетевом коде – возможность легко переходить от потока к потоку. Для экспериментов с Retrofit нам понадобятся зависимости: сама библиотека, адаптер для RxJava и конвертер для Jackson – анализатора JSON:

```
compile 'com.squareup.retrofit2:retrofit:2.0.1'
compile 'com.squareup.retrofit2:adapter-rxjava:2.0.1'
compile 'com.squareup.retrofit2:converter-jackson:2.0.1'
```

Retrofit подталкивает к типобезопасному способу взаимодействия с REST-совместимыми службами тем, что просит сначала объявить Java-интерфейс без реализации. Впоследствии этот интерфейс прозрачно транслируется в HTTP-запрос. В этом упражнении мы реализуем взаимодействие с Meetup API (https://www.meetup.com/meetup_api/), популярной службой организации мероприятий.

Одна из ее окончечных точек возвращает список городов поблизости от указанного места:

```
import retrofit2.http.GET;
import retrofit2.http.Query;

public interface MeetupApi {

    @GET("/2/cities")
    Observable<Cities> listCities(
        @Query("lat") double lat,
        @Query("lon") double lon
    );
}
```

Retrofit транслирует вызов метода `listCities()` в обращение по сети. Под капотом отправляется запрос HTTP GET к ресурсу `/2/cities?lat=...&lon=...`. Обратите внимание на тип возвращаемого значения. Это строго типизированный объект `Cities`, а не строка `String` и не слабо типизированный *словарь словарей*. Еще важнее тот факт, что `Cities` поступает от `Observable`, который порождает этот объект, как только приходит ответ. Класс `Cities` отображает большинство полей, присутствующих в JSON-ответе, полученном от сервера (методы чтения и установки опущены):

```
public class Cities {
    private List<City> results;
}

public class City {
    private String city;
    private String country;
    private Double distance;
    private Integer id;
    private Double lat;
    private String localizedCountryName;
    private Double lon;
    private Integer memberCount;
    private Integer ranking;
    private String zip;
}
```

Такой подход обеспечивает удачный баланс между абстракцией (использование таких высокоуровневых концепций, как вызовы методов и строго типизированные ответы) и низкоуровневыми деталями (асинхронная природа сетевых обращений). Хотя HTTP поддерживает семантику вопрос-ответ, мы моделируем неизбежную задержку с помощью `Observable`, не скрывая ее за дырявой абстракцией блокирующего RPC (вызова удаленной процедуры). К сожалению, для взаимодействия с этим конкретным API придется настроить кое-какой связующий код. И хотя детали могут отличаться, важно понимать, какие шаги необходимы для правильного разбора JSON-ответа:

```

import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.PropertyNamingStrategy;
import retrofit2.Retrofit;
import retrofit2.adapter.rxjava.RxJavaCallAdapterFactory;
import retrofit2.converter.jackson.JacksonConverterFactory;

ObjectMapper objectMapper = new ObjectMapper();
objectMapper.setPropertyNamingStrategy(
    PropertyNamingStrategy.CAMEL_CASE_TO_LOWER_CASE_WITH_UNDERSCORES);
objectMapper.configure(
    DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);

Retrofit retrofit = new Retrofit.Builder()
    .baseUrl("https://api.meetup.com/")
    .addCallAdapterFactory(
        RxJavaCallAdapterFactory.create())
    .addConverterFactory(
        JacksonConverterFactory.create(objectMapper))
    .build();

```

Во-первых, мы должны настроить объект `ObjectMapper` из библиотеки Jackson, так чтобы он преобразовывал имена полей с подчеркиками в имена в верблюжьей нотации, принятой в Java Beans; например, имени `localized_country_name` в JSON-ответе будет соответствовать имя `localizedCountryName` в классе `City`. Во-вторых, мы хотим игнорировать поля, которым нет соответствия в наших bean-классах. API на основе JSON славятся тенденцией добавлять новые поля, которые клиенты раньше не поддерживали. Разумное умолчание – игнорировать такие поля и использовать только те, которые мы понимаем. Таким образом, сервер может включать в ответ какие-угодно новые поля, не опасаясь «поломать» существующие клиенты.

Имея экземпляр `Retrofit`, мы можем синтезировать реализацию `MeetupApi` для использования в клиентском коде:

```
MeetupApi meetup = retrofit.create(MeetupApi.class);
```

И наконец, с помощью класса `MeetupApi` мы можем отправлять HTTP-запросы, используя всю мощь RxJava. Построим более полный пример. Применяя `MeetupApi`, мы сначала получим список всех больших и малых городов рядом с указанным местом:

```

double warsawLat = 52.229841;
double warsawLon = 21.011736;
Observable<Cities> cities = meetup.listCities(warsawLat, warsawLon);
Observable<City> cityObs = cities
    .concatMapIterable(Cities::getResults);
Observable<String> map = cityObs
    .filter(city -> city.distanceTo(warsawLat, warsawLon) < 50)
    .map(City::getCity);

```

Сначала мы с помощью оператора `concatMapIterable()` преобразуем поток `Observable<Cities>`, содержащий всего один элемент, в поток `Observable<City>`,

в котором каждому городу соответствует отдельный элемент. Затем оставляем только города, расположенные не дальше 50 км от указанного места. И наконец, извлекаем название города. Наша следующая цель – узнать, сколько народу проживает в каждом городе в окрестностях Варшавы и подсчитать общее количество людей, живущих в радиусе 50 км. Для этого нам понадобится другой API – GeoNames (<http://www.geonames.org/export/geonames-search.html>). Один из его методов ищет населенный пункт по заданному названию и возвращает – среди прочих атрибутов – численность населения. Для подключения к этому API мы снова воспользуемся библиотекой Retrofit:

```
public interface GeoNames {

    @GET("/searchJSON")
    Observable<SearchResult> search(
        @Query("q") String query,
        @Query("maxRows") int maxRows,
        @Query("style") String style,
        @Query("username") String username);
}
```

JSON-объект необходимо отобразить на объекты данных (методы чтения и установки опущены):

```
class SearchResult {
    private List<Geoname> geonames = new ArrayList<>();
}

public class Geoname {
    private String lat;
    private String lng;
    private Integer geonameId;
    private Integer population;
    private String countryCode;
    private String name;
}
```

Объект GeoNames создается так же, как в случае MeetupApi:

```
GeoNames geoNames = new Retrofit.Builder()
    .baseUrl("http://api.geonames.org")
    .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
    .addConverterFactory(JacksonConverterFactory.create(objectMapper))
    .build()
    .create(GeoNames.class);
```

Таким образом, наше приложение потребляет и элегантно объединяет два разных API. Для каждого города мы обращаемся к GeoNames API и извлекаем из ответа численность населения:

```
Observable<Long> totalPopulation = meetup
    .listCities(warsawLat, warsawLon)
    .concatMapIterable(Cities::getResults)
```

```
.filter(city -> city.distanceTo(warsawLat, warsawLon) < 50)
.map(City::getCity)
.flatMap(gеоNames::populationOf)
.reduce(0L, (x, y) -> x + y);
```

Подумайте, сколько работы делает это коротенькая программа. Сначала она запрашивает у `MeetupApi` список городов, а затем для каждого города запрашивает численность населения. Ответы о численности населения (которые, вероятно, приходят асинхронно) затем суммируются с помощью `reduce()`. И в самом конце конвейера мы получаем объект `Observable<Long>`, порождающий единственное значение типа `long`, содержащее суммарную численность населения во всех городах. В таком органичном соединении потоков из разных источников и проявляется истинная мощь RxJava. Так, метод `populationOf()` на самом деле представляет собой сложную цепочку операторов, которые отправляют HTTP-запрос службе GeoNames по названию города и извлекают из ответа численность населения.

```
public interface GeoNames {

    default Observable<Integer> populationOf(String query) {
        return search(query)
            .concatMapIterable(SearchResult::getGeonames)
            .map(Geoname::getPopulation)
            .filter(p -> p != null)
            .singleOrDefault(0)
            .doOnError(th ->
                log.warn("По умолчанию 0 для {}", query, th))
            .onErrorReturn(th -> 0)
            .subscribeOn(Schedulers.io());
    }

    default Observable<SearchResult> search(String query) {
        return search(query, 1, "LONG", "some_user");
    }

    @GET("/searchJSON")
    Observable<SearchResult> search(
        @Query("q") String query,
        @Query("maxRows") int maxRows,
        @Query("style") String style,
        @Query("username") String username
    );
}
```

Универсальный метод `search()` в конце обернут методами по умолчанию, чтобы им было проще пользоваться. Получив объект `SearchResult` в формате JSON, мы извлекаем из него результаты поиска, проверяем, что в ответе присутствует численность населения, а в случае ошибки просто возвращаем 0. Наконец, для повышения степени конкурентности мы выполняем все запросы о численности населения в контексте диспетчера `io()`. Оператор `subscribeOn()` здесь играет центральную роль. Не будь его, запросы о численности населения в каждом городе выполнялись бы последовательно, что намного увеличило бы суммарную наблюдаемую

задержку. Но поскольку для каждого города оператор `flatMap()` вызывает метод `populationOf()` и подписывается на него в нужный момент, то данные о различных городах запрашиваются параллельно. Можно было бы добавить к каждому запросу о численности населения еще и оператор `timeout()`, чтобы уменьшить суммарное время ответа ценой не всегда полных данных. Без RxJava реализация этого сценария потребовала бы большого объема кода с привлечением пула потоков. Даже с использованием класса `CompletableFuture` (см. раздел «`CompletableFuture` и потоки» главы 5) задача все равно не тривиальна. А RxJava со своей ненавязчивой конкурентностью и мощными операторами позволяет писать быстрый, краткий и понятный код.

Сочетание двух разных API, опосредованных библиотекой Retrofit, работает, как по волшебству. Но ничто не мешает объединить совершенно разнородные потоки `Observable`; например, один может исходить от Retrofit, другой – быть результатом обращения к JDBC, а третий основан на получении сообщений JMS. Все эти случаи сравнительно легко реализовать, не допуская протекания абстракций и не раскрывая слишком много деталей о природе исходных потоков.

Диспетчеры в Android

Одна из первых ошибок, которую делает каждый начинающий разработчик для Android, – блокирование потока UI. В Android есть один выделенный главный поток для двустороннего взаимодействия с пользовательским интерфейсом. Наши обработчики обратных вызовов от платформенных виджетов вызываются в главном потоке, и там же должно выполняться обновление виджетов (изменение меток, рисование). Это ограничение намного упрощает внутреннюю архитектуру UI, но у него имеются и серьезные недостатки.

- Попытка выполнить длительную операцию (например, блокирующий сетевой вызов) из обработчика события UI не дает обрабатывать другие события, поэтому интерфейс зависает. В конечном итоге операционная система снимает такие некорректно написанные приложения.
- Обновление UI, например, по завершении блокирующего системного вызова, должно производиться в главном потоке. Нам надлежит как-то попросить операционную систему вызвать наш код обновления из главного потока.

Замечательно, что в RxJava есть два встроенных механизма для этой цели. Задачи с побочными эффектами можно выполнять в фоновом режиме с помощью оператора `subscribeOn()`, а для перехода в главный поток использовать `observeOn()`. Оба оператора были рассмотрены в разделе «Декларативная подписка с помощью `subscribeOn()`» главы 4 и отлично ложатся на архитектуру Android. Нужен лишь специальный диспетчер, знающий об окружении Android и его главном потоке. Такой диспетчер уже был частично реализован в разделе «Обзор деталей реализации диспетчера» главы 4, но, к счастью, нам не нужно писать его самостоятельно. Наше путешествие в мир RxJava для Android начнем с добавления такой зависимости:

```
compile 'io.reactivex:rxandroid:1.1.0'
```

Эта небольшая библиотека добавляет класс `AndroidSchedulers` в `CLASSPATH`, что необходимо для написания конкурентного кода для Android с применением RxJava. Работу с ней проще всего продемонстрировать на примере. Мы хотим вызвать Meetup API (см. раздел «Библиотека Retrofit со встроенной поддержкой RxJava» выше), получить список городов рядом с указанным местом и вывести его.

```
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        meetup
            .listCities(52.229841, 21.011736)
            .concatMapIterable(extractCities())
            .map(toCityName())
            .toList()
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(
                putOnListView(),
                displayError());
    }

    //...

});
```

Эта глава – единственное место в книге, где не используются лямбда-выражения из Java 8. На момент написания книги Android поддерживал только Java 7 и никаких замыканий¹. Поэтому мы вынесли анонимные внутренние классы в отдельные методы, чтобы сделать код понятнее. Если такой синтаксис кажется вам слишком громоздким (даже безотносительно к RxJava), попробуйте поэкспериментировать с библиотекой `retrolambda` (<https://github.com/orfjackal/retrolambda>), которая переносит лямбда-выражения на старые версии Java и работает на Android. В немодифицированном Android преобразования и обратные вызовы выглядят так:

```
//Cities::getResults
Func1<Cities, Iterable<City>> extractCities() {
    return new Func1<Cities, Iterable<City>>() {
        @Override
        public Iterable<City> call(Cities cities) {
            return cities.getResults();
        }
    };
}

//City::getCity
```

¹ Это может измениться с выходом версии Android N; дополнительные сведения смотрите в руководстве по использованию возможностей, появившихся в Java 8 (<https://developer.android.com/guide/platform/j8-jack.html>).

```
Func1<City, String> toCityName() {
    return new Func1<City, String>() {
        @Override
        public String call(City city) {
            return city.getCity();
        }
    };
}

//cities -> listView.setAdapter(...)
Action1<List<String>> putOnListView() {
    return new Action1<List<String>>() {
        @Override
        public void call(List<String> cities) {
            listView.setAdapter(new ArrayAdapter(
                MainActivity.this, R.layout.list, cities));
        }
    };
}

//throwable -> {...}
Action1<Throwable> displayError() {
    return new Action1<Throwable>() {
        @Override
        public void call(Throwable throwable) {
            Log.e(TAG, "Error", throwable);
            Toast.makeText(MainActivity.this,
                "Unable to load cities",
                Toast.LENGTH_SHORT)
                .show();
        }
    };
}
```

Вот что здесь происходит. Нажимая кнопку (мы избавимся от обратных вызовов в разделе «События пользовательского интерфейса как потоки» ниже), мы отправляем HTTP-запрос с помощью Retrofit и получаем в ответ объект `Observable<Cities>`, который затем преобразуем извлекая только нужную информацию. И в конце оказывается список `List<String>` с близлежащими городами.

Наличие двух диспетчеров здесь критически важно. Без `subscribeOn()` Retrofit отправляла бы HTTP-запрос в потоке клиента, и, значит, `Observable` получился бы блокирующим. Следовательно, HTTP-запрос попытался бы заблокировать главный поток Android, что было бы немедленно замечено операционной системой, и запрос завершился бы исключением `NetworkOnMainThreadException`. Традиционно для выполнения кода в фоновом режиме либо создается новый поток `Thread`, либо используется класс `AsyncTask`. Преимущества оператора `subscribeOn()` очевидны: код намного чище, меньше изменяет структуру программы, а ошибки обрабатываются декларативно благодаря уведомлению `onError`.

Вызов `observeOn()` не менее важен. После всех преобразований мы производим обновление UI только из главного потока. Не будь оператора `observeOn()`,

который переключается на диспетчер `mainThread()`, наш `Observable` попытался бы обновить `listView` из фонового потока, и это незамедлительно закончилось бы исключением `CalledFromWrongThreadException`. При этом `observeOn()` намного удобнее метода `postDelayed()` из класса `android.os.Handler` (который диспетчер `AndroidSchedulers.mainThread()` вызывает внутри себя).

Гибкость диспетчеров в сочетании с простотой API очень нравится многим разработчикам для Android. RxJava предлагает более простой, более элегантный, но вместе с тем и более безопасный способ справиться со сложностью конкурентного программирования на мобильных устройствах.



Об утечках памяти

В предыдущем примере есть существенный дефект, который может привести к утечке памяти. `Observer` хранит ссылку на объемлющий объект `Activity` и может существовать дольше его. Суть проблемы и способ ее решения описаны в разделе «Предотвращение утечек памяти в компонентах `Activity`» выше.

События пользовательского интерфейса как потоки

С точки зрения синтаксиса, цель RxJava – уйти от *ада обратных вызовов*, заменив вложенные обратные вызовы декларативными преобразованиями. Поэтому метод `setOnClickListener()`, охватывающий `Observable`, немного раздражает. По счастью, существует библиотека, которая транслирует события пользовательского интерфейса Android в потоки². Просто добавьте в проект такую зависимость:

```
compile 'com.jakewharton.rxbinding:rxbinding:0.4.0'
```

Теперь можно заменить императивную регистрацию обратного вызова таким конвейером:

```
RxView
    .clicks(button)
    .flatMap(listCities(52.229841, 21.011736))
    .delay(2, TimeUnit.SECONDS)
    .concatMapIterable(extractCities())
    .map(toCityName())
    .toList()
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(
        putOnListView(),
```

² Похожее средство существует для Swing.

```

        displayError();
    }

    Func1<Void, Observable<Cities>> listCities(final double lat, final double lon) {
        return new Func1<Void, Observable<Cities>>() {
            @Override
            public Observable<Cities> call(Void aVoid) {
                return meetup.listCities(lat, lon);
            }
        };
    }
}

```

Вместо регистрации обратного вызова, который локально создает и преобразует Observable, мы начинаем с объекта Observable<Void>, представляющего нажатия кнопки. Само нажатие не несет никакой информации, поэтому указан параметрический тип void. Каждое событие нажатия отправляет асинхронный HTTP-запрос, который возвращает Observable<Cities>. Почти ничего не изменилось. Если вам кажется, что из-за такого незначительного улучшения удобочитаемости не стоило и огород городить, то попробуйте составить композицию нескольких потоков событий GUI.

Допустим, есть два текстовых поля: широта и долгота. Как только одно из них изменяется, мы отправляем HTTP-запрос для поиска близлежащих городов. Но чтобы не создавать ненужный сетевой трафик, пока пользователь еще печатает, мы хотим ввести небольшую задержку. Сетевой запрос отправляется, только если в течение секунды ни то, ни другое текстовое поле не изменялось. Это очень похоже на автозаполнение текстовых полей, когда небольшая задержка предотвращает избыточный трафик, но только в данном случае мы учитываем не одно, а два поля. Реализация с помощью RxJava и RxBinding выглядит очень элегантно:

```

import android.widget.EditText;
import com.jakewharton.rxbinding.widget.RxTextView;
import com.jakewharton.rxbinding.widget.TextViewAfterTextChangeEvent;

EditText latText = //...
EditText lonText = //...

Observable<Double> latChanges = RxTextView
    .afterTextChangeEvent(latText)
    .flatMap(toDouble());

Observable<Double> lonChanges = RxTextView
    .afterTextChangeEvent(lonText)
    .flatMap(toDouble());

Observable<Cities> cities = Observable
    .combineLatest(latChanges, lonChanges, toPair())
    .debounce(1, TimeUnit.SECONDS)
    .flatMap(listCitiesNear());

```

А вот все преобразования (обратите внимания, каким громоздким становится код, когда нельзя воспользоваться лямбда-выражениями):

```

Func1<TextViewAfterTextChangedEvent, Observable<Double>> toDouble() {
    return new Func1<TextViewAfterTextChangedEvent, Observable<Double>>() {
        @Override
        public Observable<Double> call(TextViewAfterTextChangedEvent e) {
            String s = e.editable().toString();
            try {
                return Observable.just(Double.parseDouble(s));
            } catch (NumberFormatException e) {
                return Observable.empty();
            }
        }
    };
}

//return Pair::new
Func2<Double, Double, Pair<Double, Double>> toPair() {
    return new Func2<Double, Double, Pair<Double, Double>>() {
        @Override
        public Pair<Double, Double> call(Double lat, Double lon) {
            return new Pair<>(lat, lon);
        }
    };
}

//return latLon -> meetup.listCities(latLon.first, latLon.second)
Func1<Pair<Double, Double>, Observable<Cities>> listCitiesNear() {
    return new Func1<Pair<Double, Double>, Observable<Cities>>() {
        @Override
        public Observable<Cities> call(Pair<Double, Double> latLon) {
            return meetup.listCities(latLon.first, latLon.second);
        }
    };
}

```

Сначала `RxTextView.afterTextChanged()` преобразует императивные обратные вызовы при изменении содержимого поля `EditText`. Мы создаем два разных потока: для широты и для долготы. Событие `TextViewAfterTextChanged` на лету преобразуется в `double`, причем неправильно введенные значения молча отбрасываются. Два потока `double` объединяются с помощью оператора `combineLatest()`, так что мы получаем поток пар при изменении любого поля. Оператор `debounce()` (см. раздел «Пропуск устаревших событий с помощью `debounce()`» главы 6) ждет одну секунду, прежде чем отправить пару дальше, — на случай, если редактирование (широты или долготы) еще не закончилось. Благодаря `debounce()` мы предотвращаем ненужные сетевые вызовы во время печати. Больше в приложении ничего не меняется.

Этот пример показывает, как реактивное программирование распространяется от библиотеки `Retrofit` на пользовательские компоненты, так что приложение приобретает вид композиции потоков. Не забудьте только отписаться от `afterTextChanged()`, иначе появится утечка памяти.

Управление отказами с помощью Hystrix

Распределенной называется такая система, в которой отказ компьютера, о существовании которого вы даже не подозревали, может сделать ваш собственный компьютер бесполезным.

– Лесли Лэмпорт, 1987

В RxJava есть много операторов для поддержки написания масштабируемых, реактивных и разносторонних приложений.

- Декларативная конкурентность с помощью диспетчеров (см. раздел «Многопоточность в RxJava» главы 4).
- Таймауты (раздел «Таймаут в случае отсутствия событий» главы 7) и различные механизмы обработки ошибок (разделы «Обработка ошибок» и «Повтор после ошибки» главы 7).
- Применение `flatMap()` для распараллеливания работы, с одной стороны, (раздел «`flatMap()` как оператор асинхронного сцепления» главы 4) и ограничения уровня конкурентности («Управление уровнем конкурентности `flatMap()`» главы 3), с другой.

И все же для создания надежных и многогранных приложений, особенно в облачной среде или при использовании микросервисной архитектуры, необходимы дополнительные средства, выходящие за рамки RxJava. В этом разделе мы кратко познакомимся с библиотекой Hystrix (<https://github.com/Netflix/Hystrix>) для управления отказами и их изоляции в распределенных системах. Hystrix позволяет обернуть действия, которые потенциально могут привести к отказу, окружив такой код подходящей к делу логикой. Сюда входят:

- паттерн Переборка (Bulkhead), позволяющий на некоторое время вообще отсечь некорректные действия;
- быстрое прекращение за счет применения таймаутов, ограничения уровня конкурентности и реализации так называемого *прерывателя*;
- пакетирование запросов путем объединения нескольких небольших заказов в один большой;
- сбор, публикация и визуализация статистических показателей.

Одно из величайших достоинств Hystrix – прерыватель (circuit breaker), механизм для временного выключения отказавших зависимостей, с тем чтобы предотвратить каскадное распространение отказов. Если отказ в распределенной системе не обработать должным образом, то он может распространиться на последующие зависимости – точно так же, как распространяется вверх по стеку исключение. В распределенной системе запрос одного конечного пользователя легко может стать причиной десятков, а то и сотен запросов к различным зависимостям. Одна

вышедшая из строя служба, пусть даже не очень существенная, может «положить» всю систему — все запросы до единого.

Интересно, что медленно работающая служба может оказаться даже хуже не работающей вовсе. Если все пользователи внезапно получают ясное сообщение об ошибке, это плохо. Но гораздо хуже, когда пользователи не получают вообще никакого сообщения, а просто ждут бесконечно. Типичная реакция (согласитесь!) — попытаться обновить веб-страницу. Помогает редко, а чаще отправляет еще один запрос, что только усугубляет ситуацию. Одна медленная служба приводит к каскаду задержек и остановке всей системы. Ни с того ни с сего все службы, пользующиеся медленной, сами становятся медленными, и беда распространяется рекурсивно. *Hystrix* пытается экранировать такие «битые» зависимости и остановить каскад отказов.

Hystrix: первые шаги

Для изучения *Hystrix* в этой книге есть несколько причин. Во-первых, она построена на основе RxJava и потому является отличным практическим примером применения реактивных расширений в реальной жизни. Во-вторых, мы можем выполнить команду *Hystrix* и получить в ответ объект *Observable*. Наконец, и это самое главное, *Hystrix* поддерживает неблокирующие команды (см. раздел «Неблокирующие команды и *HystrixObservableCommand*» ниже).

Прежде чем двигаться дальше, рассмотрим применение *Hystrix* в простейшем блокирующем сценарии. Вообще говоря, использовать *Hystrix* рекомендуется, когда выполнение не ограничивается одним процессом или машиной. Сетевой вызов (но также и ввод-вывод) существенно повышает риски отказа: к рискам следует отнести непредсказуемые задержки, утрату связности сети и потерю пакетов. Такой потенциально опасный участок кода можно обернуть объектом *HystrixCommand*:

```
import org.apache.commons.io.IOUtils;
import com.netflix.hystrix.HystrixCommand;
import com.netflix.hystrix.HystrixCommandGroupKey;

class BlockingCmd extends HystrixCommand<String> {
    public BlockingCmd() {
        super(HystrixCommandGroupKey.Factory.asKey("SomeGroup"));
    }

    @Override
    protected String run() throws IOException {
        final URL url = new URL("http://www.example.com");
        try (InputStream input = url.openStream()) {
            return IOUtils.toString(input, StandardCharsets.UTF_8);
        }
    }
}
```

Блокирующий код, который теоретически может отказать, помещается внутрь метода *run()*. Тип *T*, возвращаемый этим методом, определяется универсальным

типом `HystrixCommand<T>`. Если мы хотим параметризовать действие (например, задать другой URL), то параметр следует передавать через конструктор. Тип `HystrixCommand` — реализация паттерна проектирования *Команда*, описанного в книге Э. Гамма и др. «Приемы объектно-ориентированного проектирования. Паттерны проектирования» (издательство «Питер»).

Получив экземпляр `HystrixCommand`, мы должны его как-то выполнить. Есть два способа блокирующего выполнения: вызвать метод `execute()` или получить объект `Future` (до Java 8), но ни один из них нам не интересен:

```
String string = new BlockingCmd().execute();  
Future<String> future = new BlockingCmd().queue();
```

Метод `execute()` косвенно вызывает метод `run()`, поставив перед ним страховочную сетку из таймаутов, хитроумной обработки ошибок и т. д. Подробнее об этом можно прочитать в разделе «Паттерн Переборка и быстрое прекращение» ниже. Этот метод блокирует выполнение программы и возвращает управление только после того, как метод `run()` успешно завершится или возбудит исключение. С другой стороны, метод `queue()` не блокирует выполнение, а возвращает объект типа `Future<T>`. Старый интерфейс `Future` не особенно реактивен, так что в этой книге ни `execute()`, ни `queue()` нас интересовать не будут.



Отметим, что мы всегда создаем новый экземпляр `BlockingCmd` — не разрешается повторно выполнить старый экземпляр команды. Объекты `HystrixCommand` должны создаваться непосредственно перед выполнением и никогда не используются повторно. На практике команда обычно параметризуется при конструировании, поэтому возможность повторного использования в любом случае сомнительна.

Hystrix в полной мере поддерживает тип `Observable`³ и может возвращать результат команды в виде потока:

```
Observable<String> eager = new BlockingCmd().observe();  
Observable<String> lazy = new BlockingCmd().toObservable();
```

Семантическое различие между `observe()` и `toObservable()` весьма важно. Метод `toObservable()` преобразует команду в ленивый и холодный объект `Observable` — команда не выполняется, пока кто-то не подпишется на этот объект. К тому же, этот `Observable` не кэшируется, так что каждый вызов `subscribe()` приводит к новому выполнению команды. С другой стороны, `observe()` сразу вызывает команду асинхронно и возвращает горячий `observable` который еще и кэшируется. В разделе «О пользе лени» главы 4 мы узнали, как удобны ленивые объекты `Observable`; например, их можно создать в любой момент времени, но отложить подписку и избежать побочных эффектов типа сетевых вызовов. Открывается также возможность эффективно собирать запросы в пакеты. Однако при работе с холодным

³ На самом деле, метод `execute()` реализован с помощью `queue()`, а тот — с помощью `toObservable()`.

Observable есть риск случайно вызвать действие несколько раз, если подписчиков больше одного. В таких случаях может помочь оператор `cache()`. Вообще говоря, ленивость — путь к наиболее эффективной конкурентности, поэтому лучше использовать `toObservable()`, а не `observe()`, если только нет очень веских доводов в пользу энергичного выполнения команды.

Имея Observable, мы можем применить любые операторы, например, повторить сбойнувшую команду с помощью `retry()`:

```
Observable<String> retried = new BlockingCmd()
    .toObservable()
    .doOnError(ex -> log.warn("Error ", ex))
    .retryWhen(ex -> ex.delay(500, MILLISECONDS))
    .timeout(3, SECONDS);
```

Здесь конвейер вызывает команду и в случае ошибки повторяет ее спустя 500 мс. Однако повтор не должен занимать больше 3 секунд, в противном случае возбуждается исключение `TimeoutException` (см. раздел «Таймаут в случае отсутствия событий» главы 7). Ниже мы увидим, чем могут помочь встроенные в `Hystrix` таймауты.

Неблокирующие команды и `HystrixObservableCommand`

Если приложение с самого начала проектировалось с учетом `RxJava`, то велики шансы, что действия, в которых участвуют сторонние службы и неизвестные библиотеки, уже моделируются как `Observable`. Основной класс `HystrixCommand` поддерживает только блокирующий код. Если взаимодействия с внешним миром уже представлены в виде `Observable`, и требуется дополнительно защитить их с помощью `Hystrix`, то гораздо лучше воспользоваться классом `HystrixObservableCommand`:

```
public class CitiesCmd extends HystrixObservableCommand<Cities> {

    private final MeetupApi api;
    private final double lat;
    private final double lon;

    protected CitiesCmd(MeetupApi api, double lat, double lon) {
        super(HystrixCommandGroupKey.Factory.asKey("Meetup"));
        this.api = api;
        this.lat = lat;
        this.lon = lon;
    }

    @Override
    protected Observable<Cities> construct() {
        return api.listCities(lat, lon);
    }
}
```

С классом `MeetupApi` мы познакомились в разделе «Библиотека `Retrofit` со встроенной поддержкой `RxJava`» выше и знаем, что он может вернуть значение типа `Observable<Cities>`. `Hystrix` прозрачно обертывает этот `Observable`, добавляя средства отказоустойчивости, о которых мы поговорим чуть ниже. Класс `CitiesCmd`, ко всему прочему, гораздо более типичен, чем `BlockingCmd`, потому что его конструктор принимает несколько параметров. В автономных тестах мы можем также передать заглушку `MeetupApi` для проверки поведения команды.

Преимущество класса `HystrixObservableCommand` по сравнению с `HystrixCommand` в том, что первому для работы не требуется пул потоков. Команды `HystrixCommand` всегда выполняются в связанном пуле потоков, тогда как командам `Observable` дополнительные потоки не нужны. Конечно, объект `Observable`, возвращенный методом `construct()` (обратите внимание, что метода с именем `run()` больше нет), может использовать какие-то потоки в зависимости от его реализации. Зная, как создаются команды в `Hystrix` и как они сочетаются с экосистемой `RxJava`, мы можем перейти к тем дополнительным возможностям, которые предоставляет `Hystrix`.

Паттерн Переборка и быстрое прекращение

Переборками называются перегородки, разделяющие корпус судна на водонепроницаемые отсеки. В случае течи переборка удерживает воду в одном отсеке и не дает судну затонуть. Тот же инженерный принцип применим и к распределенным системам. Отказавший компонент необходимо изолировать. Система должна работать, даже если отдельные компоненты вышли из строя.

В проектировании программного обеспечения находит применение еще один инженерный принцип – *прерыватель*. Задача прерывателя цепи – остановить протекание электрического тока и защитить устройства от перегрузки и, возможно, возгорания. Прерыватель можно сбросить (вручную или автоматически), когда опасность миновала. Но не приведет ли это к выключению света, отопления или (в худшем случае) маршрутизатора? Не обязательно. Другие электрические устройства могут быть защищены независимыми прерывателями, поэтому будут продолжать работать. Но самое главное – ваш дом не загорится.

`Hystrix` реализует оба эти паттерна, относящиеся к системной интеграции. У каждой команды имеется таймаут (по умолчанию 1 секунда) и ограничение на уровень конкурентности (по умолчанию не более 10 команд из заданной группы). Эти довольно жесткие ограничения гарантируют, что команда не захватит слишком много ресурсов, например потоков и памяти. Кроме того, применение таймаутов защищает от чрезмерной задержки. Это поведение можно сравнить с переборками в судне, поскольку если какая-то зависимость начинает сбывать (напомним, чрезмерная задержка неотличима от отказа), проблема не затронет систему в целом. Таймауты и ограничения на уровень конкурентности значительно уменьшают количество потоков, блокированных в ожидании ответа от внешней системы.

А прерыватель ведет себя еще «умнее». Что если какая-то зависимость раньше отвечала за 100 мс, а теперь почти каждый раз дает таймаут по истечении 1 секун-

ды? Если эта зависимость вызывалась как часть процедуры обработки запроса, то почти все операции будут занимать на секунду дольше. Без Hystrix задержка могла бы быть гораздо дольше, но реализовать таймауты позволяет и сама RxJava. Hystrix же делает куда больше. Обнаружив, что какая-то команда продолжает сбывать (выдавать исключения или таймауты) слишком часто (по умолчанию в половине всех вызовов) в течение заданного временного окна (по умолчанию 10 секунд), Hystrix прерывает цепь (circuit). А дальше очень интересно. Hystrix вообще перестает вызывать сбойную команду, а сразу возбуждает исключение – пример быстрого прекращения.

Самое время посмотреть на Hystrix в действии. Сначала мы с помощью Mockito создадим вместо класса MeetupApi подставку, которая всегда будет давать ошибку после неприемлемо большой задержки:

```
import static org.mockito.BDDMockito.given;
import static org.mockito.Matchers.anyDouble;
import static org.mockito.Mockito.mock;

MeetupApi api = mock(MeetupApi.class);
given(api.listCities(anyDouble(), anyDouble())).willReturn(
    Observable
        .<Cities>error(new RuntimeException("Broken"))
        .doOnSubscribe(() -> log.debug("Вызывается"))
        .delay(2, SECONDS)
);
```

По умолчанию таймаут равен одной секунде, поэтому мы никогда не увидим исключения "Broken", т. к. таймаут случится раньше. Затем мы хотели бы несколько раз вызвать MeetupApi одновременно и посмотреть, как поведет себя Hystrix:

```
Observable
    .interval(50, MILLISECONDS)
    .doOnNext(x -> log.debug("Запрашивается"))
    .flatMap(x ->
        new CitiesCmd(api, 52.229841, 21.011736)
            .toObservable()
            .onErrorResumeNext(ex -> Observable.empty()),
        5)
```

С помощью оператора interval() мы порождаем события каждые 50 мс. Для каждого события мы вызываем CitiesCmd и проглатываем ошибки. Напомним, что в реальных проектах следует хотя бы протоколировать ошибки, пользуясь оператором doOnError(). Каждые 50 мс Hystrix вызывает нашу команду и замечает, что она дает таймаут после 1 секунды. На самом деле, команда работает еще медленнее, но Hystrix прерывает ее раньше. Подписавшись, мы увидим, что CitiesCmd несколько раз вызывалась, а потом внезапно перестала. И хотя сообщение "Запрашивается" по-прежнему появляется раз в 50 мс, команда больше не вызывается.

Hystrix, применяя определенные эвристики, поняла, что CitiesCmd неисправна, и перестала вызывать ее. Теперь при любой попытке вызвать эту команду возвращенный observable сразу возбуждает исключение. Это вмешался прерыватель.

Наша команда не вызывается, потому что Hystrix увидела, что она постоянно сбоят и вызывать ее не имеет смысла. Когда частота отказов превышает 50 %, прерыватель размыкает цепь, и все последующие попытки вызвать команду немедленно дают ошибку. Под отказом Hystrix понимает как исключение, так и таймаут.

У прерывателя двойные достоинства. С точки зрения приложения, вызывающего команду, она в любом случае отказала бы, но так мы получаем ответ быстрее, и для пользователя это лучше. Но еще интереснее обстоит дело с точки зрения сервера – ну, или любой программы, которой адресовался запрос. Если команда продолжает сбывать или работать слишком долго, то, возможно, причина в трудностях, испытываемых зависимостью (другой службой, базой данных). Быть может, это перезапуск, всплеск нагрузки или очень долгая пауза на сборку мусора. Отрубив команду с помощью прерывателя, мы даем системе возможность перевести дух. Не исключено, что после окончания всплеска нагрузки или опустошения внутренней очереди система снова станет отвечать своевременно. Тем самым мы предотвращаем DDoS-атаку (распределенную атаку на отказ от обслуживания) системы на саму себя.

А как Hystrix узнает, что зависимость пришла в норму, и можно замкнуть цепь? На наше счастье, этот процесс автоматический. В примере выше мы в какой-то момент перестали видеть сообщение «Вызывается», и считали это свидетельством того, что цепь разомкнута и команда больше не выполняется. Это не совсем так. Время от времени (по умолчанию раз в 5 секунд) Hystrix все же пропускает один запрос и вызывает команду, чтобы проверить, не заработала ли она. Все остальные клиенты по-прежнему видят быстрый отказ. Если этот запрос завершается успешно, Hystrix предполагает, что все наладилось, и замыкает цепь, в противном случае цепь остается разомкнутой.

Это свойство, называемое *самоисцелением* (self-healing), играет важную роль в вычислительных системах. Hystrix помогает в двух направлениях. Временно выключая сбойные команды, она дает зависимостям возможность восстановиться. После восстановления система возвращается к нормальной работе. Без такого рода механизмов даже незначительный сбой мог бы привести к каскаду отказов и необходимости ручного перезапуска для восстановления стабильной работы компонентов.

Пакетирование и объединение команд

Одна из самых продвинутых функций Hystrix – пакетирование запросов. Представьте, что вы отправляете вниз несколько небольших запросов в ходе обработки одного запроса, поступившего сверху. Например, требуется показать список книг, и для каждой книги нужно запросить у внешней системы ее оценку:

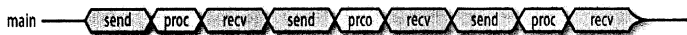
```
Observable<Book> allBooks() { /* ... */ }  
Observable<Rating> fetchRating(Book book) { /* ... */ }
```

Метод `allBooks()` возвращает поток книг `Book`, нуждающихся в обработке, а метод `fetchRating()` получает оценку `Rating` одной книги. Наивная реализация –

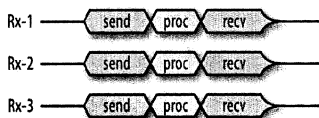
просто перебрать все книги и получать оценки по очереди. К счастью, RxJava прекрасно справляется с асинхронным запуском подзадач:

```
Observable<Rating> ratings = allBooks()
    .flatMap(this::fetchRating);
```

На следующих диаграммах последовательный вызов `fetchRatings()` сравнивается с использованием `flatMap()`. Выделены три фазы: `send` – отправка запроса, `proc` – обработка на стороне сервера, `recv` – передача ответа. На первой диаграмме показана последовательная выборка⁴:



Далее показана выборка с использованием `flatMap()`:



Все работает замечательно, и мы наблюдаем вполне удовлетворительную производительность. Все вызовы `fetchRating()` выполняются одновременно, что заметно уменьшает задержку. Но если учесть, что для каждого вызова `fetchRating()` имеется фиксированная сетевая задержка, то вызывать этот метод для нескольких десятков книг расточительно. Гораздо лучше было бы собрать все книги в один пакет и получить один ответ, содержащий все оценки:



Отметим, что все фазы – отправка, обработка и получение – занимают немного больше времени, что и не удивительно, поскольку приходится передавать и обрабатывать больше данных. Поэтому общая задержка фактически выше, если сравнивать с несколькими небольшими запросами. Улучшение под вопросом. Но надо взглянуть на картину в целом.

Хотя задержка для отдельного запроса увеличилась, пропускная способность системы, скорее всего, намного возросла. Количество одновременных соединений, полоса пропускания сети и потоки JVM – это ограниченные ресурсы. Если пропускная способность зависимости ограничена, то ее легко исчерпать сравнительно небольшим числом одновременно выполняемых транзакций. Эгоистично примененный оператор `flatMap()` снижает задержку для одного запроса, но может ухудшить производительность всех остальных из-за исчерпания ресурсов. Сле-

⁴ Для генерации диаграмм использовался проект по адресу <https://github.com/drom/wavedrom>.

довательно, имеет смысл пойти на жертву: немного увеличить задержку, чтобы добиться значительного улучшения суммарной пропускной способности за счет снижения нагрузки на зависимости. А в итоге и задержка уменьшится: ресурсы распределяются между запросами более справедливо, поэтому задержка становится более предсказуемой.

Но как реализовать пакетирование? Hystrix знает обо всех выполняемых командах. Увидев две похожие команды, запущенные примерно в одно время (например, для получения двух оценок), она может свернуть их в одну пакетную команду. Из пришедшего на нее ответа выделяются ответы на отдельные запросы. Прежде всего, нам понадобится реализация пакетной команды, которая может выбирать сразу несколько оценок:

```
class FetchManyRatings extends HystrixObservableCommand<Rating> {  
  
    private final Collection<Book> books;  
  
    protected FetchManyRatings(Collection<Book> books) {  
        super(HystrixCommandGroupKey.Factory.asKey("Books"));  
        this.books = books;  
    }  
  
    @Override  
    protected Observable<Rating> construct() {  
        return fetchManyRatings(books);  
    }  
}
```

Метод `fetchManyRatings()` принимает в качестве аргумента несколько книг (`books`) и порождает несколько экземпляров класса `Rating`. Внутри он может отправить по HTTP один пакетный запрос нескольких оценок, в отличие от метода `fetchRating(book)`, который всегда выбирает только одну оценку. Запрашивать несколько оценок, конечно, медленнее, чем одну, но заведомо быстрее, чем запрашивать оценки последовательно. Однако мы хотим вручную собирать пакет из нескольких запросов, а затем разбирать на части ответ на него. Это было бы еще ничего, когда имеется одна-единственная транзакция, но как быть, если оценки одновременно запрашивают несколько клиентов? Если два независимых запроса из двух браузеров доходят до нашего сервера, мы все равно хотели бы объединить их в пакет и отправить его дальше. Однако это потребовало бы межпоточковой синхронизации и какого-то глобального реестра всех запросов. Представьте, что один поток пытается выполнить некоторую команду, а другой поток выполняет такую же команду (но с другими аргументами) спустя несколько миллисекунд. Мы хотим немного повременить с отправкой команды после первого запроса в расчете на то, что скоро придет еще одна такая же команда. В таком случае оба запроса необходимо запомнить, объединить, отправить один пакетный запрос и сопоставить полученные на него ответы исходным запросам. Именно это и сделает Hystrix, если мы ей немного поможем:

```

public class FetchRatingsCollapser
    extends HystrixObservableCollapser<Book, Rating, Rating, Book> {

    private final Book book;

    public FetchRatingsCollapser(Book book) {
        // См. объяснение ниже
    }

    public Book getRequestArgument() {
        return book;
    }

    protected HystrixObservableCommand<Rating> createCommand(
        Collection<HystrixCollapser.CollapsedRequest<Rating, Book>> requests) {
        // См. объяснение ниже
    }

    protected void onMissingResponse(
        HystrixCollapser.CollapsedRequest<Rating, Book> r)
    {
        r.setException(new RuntimeException("Не найдено для: " + r.getArgument()));
    }

    protected Func1<Book, Book> getRequestArgumentKeySelector() {
        return x -> x;
    }

    protected Func1<Rating, Rating> getBatchReturnTypesToResponseTypeMapper() {
        return x -> x;
    }

    protected Func1<Rating, Book> getBatchReturnTypesKeySelector() {
        return Rating::getBook;
    }
}

```

Кода много, поэтому разберем его по частям. Желая получить одну оценку `Rating` для заданной книги `Book`, мы создаем экземпляр класса `FetchRatingsCollapser`:

```

Observable<Rating> ratingObservable =
    new FetchRatingsCollapser(book).toObservable();

```

Благодаря классу `HystrixObservableCollapser` клиентский код может совершенно ничего не знать о пакетировании и свертывании. Снаружи все выглядит так, будто мы выбираем одну оценку для одной книги. Но внутри есть несколько интересных деталей, обеспечивающих возможность пакетирования. Сначала в конструкторе мы не только запоминаем книгу, соответствующую этому запросу, но и настраиваем свертывание запросов:

```

public FetchRatingsCollapser(Book book) {
    super(withCollapserKey(HystrixCollapserKey.Factory.asKey("Books"))
        .andCollapserPropertiesDefaults(HystrixCollapserProperties.Setter())

```

```

        .withTimerDelayInMilliseconds(20)
        .withMaxRequestsInBatch(50)
    )
    .andScope(Scope.GLOBAL));
    this.book = book;
}

```

В методе `withTimerDelayInMilliseconds()` задано 20 мс – продолжительность временного окна, в течение которого запросы сворачиваются (по умолчанию 10 мс). При появлении первого запроса его выполнение задерживается на 20 мс. В течение этого времени Hystrix ждет появления других запросов, возможно, из других потоков. Если время истекло или накопилось 50 запросов (параметр в вызове `withMaxRequestsInBatch(50)`), то шлюз открывается. Предполагается, что в этот момент библиотека выполнит все хранящиеся в очереди команды одним пакетом. Но Hystrix не умеет творить чудеса – вы должны подсказать, как объединить команды в пакет. Делается это так:

```

protected HystrixObservableCommand<Rating> createCommand(
    Collection<HystrixCollapser.CollapsedRequest<Rating, Book>> requests) {
    List<Book> books = requests.stream()
        .map(c -> c.getArgument())
        .collect(toList());
    return new FetchManyRatings(books);
}

```

Задача метода `createCommand()` – преобразовать отдельные запросы в одну пакетную команду. Он получает коллекцию `requests` запросов, поступивших в течение 20 миллисекунд, и собирает из них один пакетный запрос. В нашем случае конструируется объект класса `FetchManyRatings`, содержащий все книги, для которых запрошена оценка. Затем Hystrix выполняет эту пакетную команду и подписывается на получение нескольких ответов. Отметим, что классу `HystrixObservableCommand` разрешено возвращать несколько значений, а это как раз то, что нам нужно.

Когда от `FetchManyRatings` начинают поступать ответы, мы должны каким-то образом сопоставить экземпляры `Rating` независимым запросам. Напомню, что в этот момент имеется несколько потоков и транзакций, каждая из которых ожидает получить только одну оценку. Разбор пакетного ответа и маршрутизация к исходным запросам производится более-менее автоматически при поддержке со стороны следующих методов:

```
getRequestArgumentKeySelector()
```

Сопоставляет аргументу отдельного запроса (`book`) ключ, с помощью которого впоследствии можно будет сопоставить ответ и запрос. В нашем случае в качестве ключа используется сам экземпляр `Book`, поэтому мы задаем тождественное преобразование `x -> x`.

```
getBatchReturnTypesToResponseTypeMapper()
```

Сопоставляет один элемент пакетного ответа одному исходному запросу. И снова в нашем случае достаточно тождественного преобразования `x -> x`.

```
getBatchReturnTypeInfoSelector()
```

Сообщает Hystrix, какому ключу (`Book`) соответствует данный ответ (`Rating`). Для простоты в каждом объекте `Rating`, выделенном из пакетного ответа, имеется метод `getBook()`, который говорит, с какой книгой связана данная оценка.

Располагая всеми этими методами (и особенно последним: `getBatchReturnTypeInfoSelector()`), Hystrix подготавливает отображение между ключами (`Book`) и отдельными запросами. С его помощью она сможет автоматически сопоставить выделенные из пакета ответы и запросы.

Конечно, пришлось написать немало связующего кода для поддержки пакетирования, но это окупается сторицей. Если несколько клиентов обращаются к одной и той же зависимости, например кэширующему серверу, мы можем собрать много запросов в один пакет. Это заметно уменьшает трафик. Если зависимость является узким местом с ограниченной пропускной способностью, то свертывание запросов зачастую уменьшает нагрузку на нее. Однако пакетирование вносит дополнительную задержку на стороне клиента. Когда протяженность временного окна составляет 10 мс (`withTimerDelayInMilliseconds(10)`), каждый запрос при высокой нагрузке задерживается в среднем на 5 мс. Фактическая задержка зависит от того, был ли взведен для запроса новый таймер или запрос появился после начала формирования текущего пакета.

Отметим, что пакетирование не имеет смысла при низкой нагрузке. Если пакеты, содержащие более одного запроса, формируются очень редко, то мы просто добавляем лишнюю задержку в 10 мс к каждому запросу. Это время Hystrix проводит в бесполезном ожидании последующих запросов. Следовательно, важно правильно настраивать пакетирование запросов. Так, если задержка составляет 10 мс, то пакетирование имеет смысл лишь в случае, когда отправляется не менее 100 запросов в секунду. Иначе очень редко будут появляться пакеты, содержащие более одного запроса.



Настройка величины `withTimerDelayInMilliseconds`

Соблазнительно установить очень большую задержку, чтобы в один пакет попадало как можно больше запросов. Задавать величину 100 мс или даже 1 с вполне допустимо, но только в офлайновых системах, где генерируется большой объем трафика, а задержка не является проблемой.

Пакетирование оптимально работает при высокой нагрузке. Поэтому Hystrix предоставляет весьма развитый механизм мониторинга, позволяющий оценить общую производительность системы.

Мониторинг и инструментальные панели

Для правильной работы Hystrix должна собирать большой объем статистических данных для каждой команды, например, подсчитывать количество успеш-

ных и неудачных выполнений и распределение времени ответа. Было бы чистым эгоизмом хранить такие ценные данные внутри библиотеки, но не расстраивайтесь: Hystrix предлагает несколько способов доступа к ним. Можно подписаться на различные потоки событий, порождаемые библиотекой. Например, следующая программа создает поток событий `HystrixCommandCompletion`, порождаемых при каждом завершении команды `FetchRating`:

```
import com.netflix.hystrix.metric.HystrixCommandCompletion;
import com.netflix.hystrix.metric.HystrixCommandCompletionStream;

Observable<HystrixCommandCompletion> stats =
    HystrixCommandCompletionStream
        .getInstance(HystrixCommandKey.Factory.asKey("FetchRating"))
        .observe();
```

`HystrixCommandCompletionStream` — это фабрика именно таких потоков, но есть и другие, например: `HystrixCommandStartStream` или `HystrixCollapserEventStream`. Располагая этими потоками, ваше приложение может построить развитую систему мониторинга. Например, чтобы узнать, сколько раз в секунду данная команда заканчивалась неудачно, можно написать такой код:

```
import static com.netflix.hystrix.HystrixEventType.FAILURE;

HystrixCommandCompletionStream
    .getInstance(HystrixCommandKey.Factory.asKey("FetchRating"))
    .observe()
    .filter(e -> e.getEventCounts().getCount(FAILURE) > 0)
    .window(1, TimeUnit.SECONDS)
    .flatMap(Observable::count)
    .subscribe(x -> log.info("{} failures/s", x));
```

Но для создания инфраструктуры мониторинга поверх этих потоков ее нужно сначала спроектировать, а потом реализовать. Кроме того, вы, возможно, захотите вывести результаты мониторинга наружу. Благодаря модулю `hystrix-metrics-event-stream` Hystrix поддерживает отправку агрегированных метрик по протоколу HTTP. Если приложение работает поверх контейнера сервлетов или в него вложен контейнер сервлетов, то достаточно включить `HystrixMetricsStreamServlet` в состав отображений. В противном случае можно запустить миниатюрный контейнер самостоятельно:

```
import
    com.netflix.hystrix.contrib.metrics.eventstream.HystrixMetricsStreamServlet;
import org.eclipse.jetty.server.Server;
import org.eclipse.jetty.servlet.ServletContextHandler;
import org.eclipse.jetty.servlet.ServletHolder;
import static org.eclipse.jetty.servlet.ServletContextHandler.NO_SESSIONS;

//...

ServletContextHandler context = new ServletContextHandler(NO_SESSIONS);
```

```
HystrixMetricsStreamServlet servlet = new HystrixMetricsStreamServlet();
context.addServlet(new ServletHolder(servlet), "/hystrix.stream");
Server server = new Server(8080);
server.setHandler(context);
server.start();
```

Неважно, отображали вы сервлет на существующий контейнер или запустили свой, теперь можно получать поток статистики от Hystrix в режиме реального времени. Отметим, что это не соединение типа запрос-ответ, а поток событий, управляемых сервером (SSE). Каждую секунду клиенту отправляется новый пакет статистических данных в формате JSON:

```
$ curl -v localhost:8080/hystrix.stream
> GET /hystrix.stream HTTP/1.1
...
< HTTP/1.1 200 OK
< Content-Type: text/event-stream;charset=UTF-8

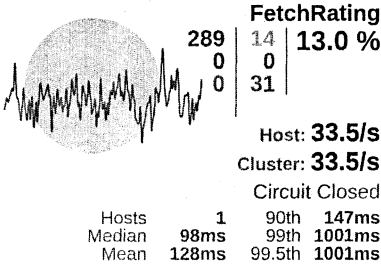
ping:

data: {
  "currentConcurrentExecutionCount": 2,
  "errorCount": 0,
  "errorPercentage": 0,
  "group": "Books",
  "isCircuitBreakerOpen": false,
  "latencyExecute": { /* ... */ },
  "latencyExecute_mean": 0,
  "latencyTotal": {"0":18, "25":80, "50":98, "75":120, "90":138,
                  "95":146, "99":159, "99.5":159, "100":167},
  "latencyTotal_mean": 0,
  "name": "FetchRating",
  "propertyValue_circuitBreakerErrorThresholdPercentage": 50,
  "propertyValue_circuitBreakerSleepWindowInMilliseconds": 5000,
  "propertyValue_executionIsolationSemaphoreMaxConcurrentRequests": 10,
  "propertyValue_executionTimeoutInMilliseconds": 1000,
  "requestCount": 334
  ...
}

data: { ...
```

Даже из этого урезанного примера видно, что отправляет сервер: измеряемая команда, распределение времени задержки (от нулевого до 100-го перцентиля), разомкнут ли прерыватель, значения параметров (порог ошибки, таймауты и т. д.). Этот непрерывный поток данных могут потреблять специализированные средства мониторинга и инструментальные панели. Но и здесь Hystrix приходит на помощь и предлагает очень надежную инструментальную панель, которая почти полностью написана на JavaScript и работает в браузере. Этому автономному приложению (hystrix-dashboard) нужен только URL-адрес вашего потока `hystrix.stream`. Пример инструментальной панели показан на рисунке ниже.

Success | Short-Circuited | Bad Request | Timeout
Rejected | Failure | Error %



Каждой команде соответствует своя плитка, на которой представлено несколько важных телеметрических показателей, в том числе:

- количество выполненных команд, собранных в несколько групп: завершившиеся успешно (289), завершившиеся по таймауту (14), завершившиеся с ошибкой (31), закороченные (0) и т. д.;
- проценты задержки (например, мы видим, что для выполнения 90 % запросов требовалось не более 147 мс) и график недавней истории;
- состояние прерывателя и общая пропускная способность;
- статистика пула потоков, если используется блокирующий класс `HystrixCommand`.

С помощью проекта `Turbine` (<https://github.com/Netflix/Turbine>) на инструментальной панели можно показать также агрегированные потоки от нескольких серверов. Потому-то мы и видим количество хостов и пропускную способность кластера, хотя приходит поток только с одной машины. Инструментальная панель `Hystrix` чрезвычайно полезна, потому что может отображать состояние нескольких команд в режиме, близком к реальному времени. Ко всему прочему, в ней применяется цветовое кодирование, так что если какие-то команды начинают сбоить, соответствующие им плитки краснеют.

`Hystrix` – полезный инструмент в распределенных системах, где отказы неизбежны. Паттерн Команда позволяет инкапсулировать и изолировать места возникновения ошибок. Благодаря великолепной интеграции с `RxJava` эта библиотека – отличный выбор для приложений, которым нужна улучшенная обработка ошибок.

Опрос баз данных NoSQL

В типичном современном приложении есть два источника данных с большим временем задержки: обращения по сети (как правило, по протоколу `HTTP`) и запросы к базе данных. Библиотека `Retrofit` (см. раздел «Библиотека `Retrofit` со встроен-

ной поддержкой RxJava» выше) – изумительный источник объектов Observable, связанных с асинхронным обращением по HTTP. Что касается доступа к базам данных, то мы потратили довольно много времени на изучение баз данных SQL (см. раздел «Доступ к реляционным базам данных» главы 5), которые исторически всегда были блокирующими вследствие устройства JDBC API. В этом отношении базы данных NoSQL более современны и часто идут в комплекте с асинхронными неблокирующими клиентскими драйверами. В этом разделе мы кратко рассмотрим драйверы Couchbase и MongoDB, которые изначально поддерживают RxJava и могут возвращать объект Observable в ответ на внешний вызов.

Клиентский API Couchbase

Couchbase Server (<https://www.couchbase.com/>) – современная документная база данных семейства NoSQL. Интересно, что Couchbase полноценно поддерживает RxJava в своем клиентском API. Реактивные расширения – это не просто обертка, а официально поддерживаемый идиоматический способ взаимодействия с базой данных. У многих других движков хранения имеется неблокирующий асинхронный API, но создатели Couchbase выбрали RxJava как наилучшую основу для клиентского уровня.

В качестве примера запросим у набора данных `travel-sample` документ с идентификатором `route_14197`. В этом наборе данных документ о маршруте имеет такой вид:

```
{
  "id": 14197,
  "type": "route",
  "airline": "B6",
  "airlineid": "airline_3029",
  "sourceairport": "PHX",
  "destinationairport": "BOS",
  "stops": 0,
  "equipment": "320",
  "schedule": [
    {
      "day": 0,
      "utc": "22:12:00",
      "flight": "B6928"
    },
    {
      "day": 0,
      "utc": "06:40:00",
      "flight": "B6387"
    },
    ...
    {
      "day": 1,
      "utc": "08:16:00",
      "flight": "B6922"
    }
  ]
}
```


Каждый запрос возвращает объект `Observable`, и, начиная с этого момента, мы можем безопасно преобразовывать полученные записи, как сочтем нужным:

```
CouchbaseCluster cluster = CouchbaseCluster.create();
cluster
    .openBucket("travel-sample")
    .get("route_14197")
    .map(AbstractDocument::content)
    .map(json -> json.getJSONArray("schedule"))
    .concatMapIterable(JsonArray::toList)
    .cast(Map.class)
    .filter(m -> ((Number)m.get("day")).intValue() == 0)
    .map(m -> m.get("flight").toString())
    .subscribe(flight -> System.out.println(flight));
```

Метод `AsyncBucket.get()` возвращает значение типа `Observable<JsonDocument>`. JSON-документ, по определению, слабо типизирован, поэтому для извлечения полезной информации мы должны обойти его, ничего не зная о структуре.

Зная же, как документ выглядит, легко понять, какие преобразования применяются к `JsonDocument`. Сначала выделяется элемент `"schedule"`, затем в нем ищутся все элементы `"flight"` для которых подэлемент `"day"` равен 0. В итоге `Observer` получает строки `"B6928"`, `"B6387"` и т. д. RxJava одинаково хорошо справляется со следующими задачами:

- выборка данных с таймаутами, кэшированием и обработкой ошибок;
- преобразование данных: выделение, фильтрация, анализ внутренних полей и агрегирование.

Это убедительная демонстрация выразительности абстракции `Observable`, лаконичный API которой применим в различных ситуациях⁵.

Клиентский API MongoDB

Как и Couchbase, MongoDB (<https://www.mongodb.com/>) допускает хранение произвольных документов в формате JSON без предварительно определенной схемы. В клиентской библиотеке имеется полноценная поддержка RxJava для асинхронного сохранения и запроса данных. В примере ниже демонстрируется то и другое. Сначала в базу данных вставляется 12 документов, а по завершении папечной вставки производится их выборка.

```
import com.mongodb.rx.client.*;
import org.bson.Document;
import java.time.Month;

MongoCollection<Document> monthsColl = MongoClient
    .create()
    .getDatabase("rx")
    .getCollection("months");
```

⁵ Это похоже на механизм интегрированных в язык запросов (Language Integrated Query – LINQ) на платформе .NET.

```
Observable
    .from(Month.values())
    .map(month -> new Document()
        .append("name", month.name())
        .append("days_not_leap", month.length(false))
        .append("days_leap", month.length(true))
    )
    .toList()
    .flatMap(monthsColl::insertMany)
    .flatMap(s -> monthsColl.find().toObservable())
    .toBlocking()
    .subscribe(System.out::println);
```

Класс `Month` – это перечисление `enum` значений от `January` до `December`. Мы легко можем получить число дней в любом месяце как в обычном, так и в високосном году. Сначала создаем 12 документов в формате BSON (двоичный JSON), каждый из которых представляет один месяц и число дней в нем. Затем пакетно вставляем весь список `List<Document>`, применяя метод `insertMany()` из класса `MongoCollection`. В ответ получаем значение типа `Observable<Success>` (оно не содержит полезной информации, это синглтон). Увидев событие `Success`, мы можем приступить к опросу базы данных, вызвав `find().toObservable()`. В случае успеха мы найдем все только что вставленные 12 документов. Если для краткости исключить автоматически присвоенное свойство `_id`, то вот что будет напечатано в самом конце:

```
Document{name=JANUARY, days_not_leap=31, days_leap=31}
Document{name=FEBRUARY, days_not_leap=28, days_leap=29}
Document{name=MARCH, days_not_leap=31, days_leap=31}
...
```

Как и раньше, истинная мощь проявляется в композиции. Драйвер `MongoDB` на основе `RxJava` позволяет опрашивать одновременно несколько коллекций и добиваться конкурентности, особо не думая о ней. В следующем фрагменте мы отправляем два одновременных запроса к `MongoDB` и еще один к какой-то службе ценообразования. Отметим, что `first()` – не оператор из типа `Observable`, а оператор `MongoDB`, который возвращает `Observable` после конструирования запроса. Метод `find()` эквивалентен фразе `WHERE` в `SQL`, `projection()` – аналог команды `SELECT`, а `first()` – то же, что `LIMIT 1`:

```
Observable<Integer> days = db.getCollection("months")
    .find(Filters.eq("name", APRIL.name()))
    .projection(Projections.include("days_not_leap"))
    .first()
    .map(doc -> doc.getInteger("days_not_leap"));
Observable<Instant> carManufactured = db.getCollection("cars")
    .find(Filters.eq("owner.name", "Smith"))
    .first()
    .map(doc -> doc.getDate("manufactured"))
    .map(Date::toInstant);

Observable<BigDecimal> pricePerDay = dailyPrice(LocalDateTime.now());
```

```
Observable<Insurance> insurance = Observable
    .zip(days, carManufactured, pricePerDay,
        (d, man, price) -> {
            // Создать страховку
        })
    .subscribe();
```

Технически можно комбинировать любые объекты `Observable` независимо от их природы и источника. В этом примере мы запрашиваем две разные коллекции MongoDB, а также посылаем запрос `dailyPrice()`, который может, например, вернуть `Observable`, созданный библиотекой `Retrofit` в результате обращения по HTTP. Резюме: источник `Observable` не имеет значения, композицию асинхронных вычислений и запросов можно составлять любым способом. Собираетесь опросить несколько баз данных, веб-служб и попутно выполнить какую-то операцию в локальной файловой системе? Все можно сделать конкурентно и объединить без всяких усилий. Нужно только усвоить общие принципы работы `RxJava`, и тогда любой источник `Observable` будет выглядеть одинаково.

Интеграция с Camel

В разделе «Библиотека `Retrofit` со встроенной поддержкой `RxJava`» выше мы узнали, как отправлять HTTP-запросы. Но есть немало других способов интеграции систем, многие из них включены в каркас `Apache Camel`. Он содержит на удивление богатый набор компонентов интеграции, позволяющих подключаться более чем к двум сотням платформ и обмениваться абстрактными сообщениями. Сюда входят такие технологии, как `AMQP`, `Amazon Web Services`, `Cassandra`, `ElasticSearch`, файловая система, `FTP`, `Google API`, `JDBC`, `Kafka`, `MongoDB`, `SMTP`, `XMP` и многие, многие другие. Большинство компонентов способны проталкивать абстрактные сообщения клиенту, например, при появлении нового почтового сообщения или файла в файловой системе. `Camel` предоставляет также адаптер к `RxJava`, чтобы с входящими сообщениями можно было работать декларативным и реактивным способом.

Потребление файлов с помощью Camel

Благодаря объектам `Observable` и операторам `RxJava` мы можем единообразно интегрироваться с сотнями систем. Пусть, например, требуется отслеживать появление новых файлов в файловой системе (сравните с разделом «Периодический опрос изменений» главы 4). Встроенная в `Camel` поддержка `RxJava` позволяет решить эту задачу очень просто:

```
CamelContext camel = new DefaultCamelContext();
ReactiveCamel reactiveCamel = new ReactiveCamel(camel);

reactiveCamel
    .toObservable("file:/home/user/tmp")
    .subscribe(e -> log.info("Новый файл: {}", e));
```

Вот и всё. После создания объектов `DefaultCamelContext` и `ReactiveCamel` мы готовы к потреблению сообщений. Любой платформе, поддерживаемой Camel, соответствует определенный URI-адрес, в нашем случае он имеет вид `file:/home/user`. Вызывая `toObservable()` с таким URI, мы создаем объект универсального типа `Observable<Message>`, который будет порождать событие при каждом появлении нового файла в указанном каталоге. В самом URI-адресе для каждого типа интеграции можно задавать десятки конфигурационных параметров. Например, добавив строку `?recursive=true&noop=true` в URI типа `file`, мы сообщаем Camel, что нужно искать файлы рекурсивно и не удалять их после обнаружения.

Получение сообщений от Kafka

Опрос изменений в файловой системе – на удивление популярная техника интеграции, как и опрос FTP-каталога. Но если требуется более устойчивый к ошибкам, надежный и быстрый коммуникационный протокол, то следует обратиться к брокерам сообщений, основанным на спецификации JMS или Kafka. Kafka (<http://kafka.apache.org/>) – это брокер сообщений типа издатель-подписчик с открытым исходным кодом. При его проектировании закладывалась отказоустойчивость и способность обрабатывать сотни тысяч сообщений в секунду. У Kafka имеется собственный Java API, но очень соблазнительно подойти к нему с точки зрения `Observable`. Интеграция средствами Camel выглядит примерно так же, как и выше, только URI другой:

```
reactiveCamel
    .toObservable("kafka:localhost:9092?topic=demo&groupId=rx")
    .map(Message::getBody)
    .subscribe(e -> log.info("Сообщение: {}", e));
```

Идея о том, что можно потреблять сообщения практически с любой платформой, используя один и тот же API на основе `Observable`, удивительно продуктивна. Camel предоставляет необходимое физическое подключение, скрытое за единообразным интерфейсом, а RxJava обогащает этот API многочисленным операторами. Camel и Retrofit (см. раздел «Библиотека Retrofit со встроенной поддержкой RxJava» выше) – отличные отправные точки для включения реактивных расширений в свое приложение. Имея стабильный источник `Observable`, будет гораздо проще распространить реактивное поведение на все большие и большие части приложения.

Потоки Java 8 и CompletableFuture

Иногда возникает путаница с тем, какой абстракцией пользоваться для конкурентного программирования, особенно после выхода Java 8. Существует несколько API, позволяющих элегантно выразить асинхронные вычисления. В этом разделе мы сравним их, с тем чтобы помочь в выборе подходящего инструмента. Ниже перечислены имеющиеся абстракции.

CompletableFuture

Класс `CompletableFuture`, появившийся в Java 8, – расширение давно известного класса `Future` из пакета `java.util.concurrent`, обладающее куда большими возможностями. `CompletableFuture` позволяет зарегистрировать асинхронную функцию обратного вызова, которая вызывается, когда объект `Future` завершается – успешно или неудачно; при этом нет необходимости блокировать выполнение в ожидании результата. Но истинная мощь этого класса проистекает из возможностей композиции и преобразования, аналогичных тем, что предлагают операторы `Observable.map()` и `flatMap()`. Хотя `CompletableFuture` включен в стандартный JDK, ни один класс из стандартной библиотеки Java им не пользуется. Он прекрасно работает, но не слишком хорошо интегрирован в экосистему Java. О проблемах интероперабельности с RxJava см. раздел «Краткое введение в `CompletableFuture`» главы 5.

Параллельные потоки

Как и класс `CompletableFuture`, пакет `java.util.stream` появился в JDK 8. Потоки – это способ объявить последовательность операций – отображения, фильтрации и т. п. – до выполнения. Все операции в потоке ленивые, они не начинают выполняться, пока не появится одна из терминальных операций, например `collect()` или `reduce()`. Кроме того, JDK умеет автоматически распараллеливать некоторые операции на все доступные процессорные ядра – звучит очень соблазнительно. Параллельные потоки обещают прозрачное отображение, фильтрацию и даже сортировку больших наборов данных на нескольких ядрах. Обычно потоки порождаются из коллекций, но могут быть созданы на лету и бывают бесконечными.

rx.Observable

Тип `Observable` представляет поток событий, возникающих в непредсказуемые моменты времени. Это может быть 0, 1, фиксированное или бесконечное число событий, доступных немедленно или по истечении времени. Поток `Observable` может оканчиваться событием завершения или ошибки. Дочитав до этого места, вы уже должны хорошо понимать, что такое `Observable`.

rx.Single

По мере развития RxJava стало ясно, что у специализированного типа, представляющего ровно один результат, есть неоспоримые преимущества. Тип `Single` – это поток, который либо завершается после порождения в точности одного значения, либо дает ошибку. В этом смысле он очень напоминает `CompletableFuture`, но объекты типа `Single` ленивые, т. е. не начинают вычисления, пока на них кто-то не подпишется. Тип `Single` описан в разделе «Сравнение типов `Observable` и `Single`» главы 5.

rx.Completable

Иногда вычисление выполняется исключительно ради побочных эффектов и не возвращает никакого результата. Отправка почтового сообщения или

сохранение записи в базе данных – примеры операций, которые подразумевают ввод-вывод (и могут выиграть от асинхронной обработки), но не возвращают никакого осмысленного значения. Традиционно в таких случаях использовался тип `CompletableFuture<Void>` или `Observable<Void>`. Но еще более специфичный тип `Completable` лучше выражает намерение выполнить асинхронное вычисление без результата. Объект `Completable` может уведомить о завершении или ошибке в конкурентном вычислении и, как и все остальные типы Rx, является ленивым.

Очевидно, что есть и другие способы выразить асинхронное вычисление, например:

- Flux и Mono из проекта Reactor (<https://projectreactor.io/>). Эти типы чем-то похожи на Observable и Single соответственно;
- ListenableFuture из Guava (<https://github.com/google/guava/wiki/ListenableFutureExplained>).

Однако мы не будем увлекаться и ограничимся JDK и RxJava. Но прежде чем продолжить, я хочу сказать, что если в вашем приложении уже всюду используется класс `CompletableFuture`, то, наверное, менять ничего не стоит. Некоторые API этого класса выглядят довольно громоздко, но в целом он предоставляет хорошую поддержку реактивного программирования. И можно ожидать, что в будущем все больше каркасов будут брать его на вооружение и предлагать идиоматическую поддержку. Поддерживать RxJava в сторонних библиотеках сложнее, потому что требуется дополнительная зависимость, а `CompletableFuture` уже входит в JDK.

Полезность параллельных потоков

Ненадолго отвлечемся и обсудим параллельные потоки из стандартного JDK. В Java для преобразования умеренно большой коллекции объектов можно декларативно заказать распараллеливание:

```
List<Person> people = //...

List<String> sorted = people
    .parallelStream()
    .filter(p -> p.getAge() >= 18)
    .map(Person::getFirstName)
    .sorted(Comparator.comparing(String::toLowerCase))
    .collect(toList());
```

Обратите внимание на вызов `parallelStream()` вместо обычного `stream()`. Тем самым мы говорим, что терминальную операцию, в данном случае `collect()`, следует выполнять параллельно, а не последовательно. Предполагается, что на конечном результате это не отразится, но должно быть гораздо быстрее. Под капотом `parallelStream()` разбивает входную коллекцию на несколько частей, параллельно выполняет над ними операции, а затем объединяет результаты – в духе принципа «разделяй и властвуй».

Многие операторы распараллелить очень просто – например, `map()` и `filter()` – другие потруднее (например, `sorted()`), поскольку обработанные порции нужно еще объединить, что в случае сортировки означает слияние отсортированных последовательностей. Некоторые операции очень трудно или вообще невозможно распараллелить без дополнительных предположений. Например, операцию `reduce()` можно выполнить параллельно, только если функция-аккумулятор ассоциативна.



Одинаковые результаты?

Некоторые операторы могут давать различные результаты в контексте `stream()` и `parallelStream()`. Например, оператор `findFirst()` возвращает первый встретившийся в потоке элемент. Но есть еще оператор `findAny()`, который, на первый взгляд, делает то же самое. Однако `findFirst()` всегда возвращает самый первый элемент, а `findAny()` вправе вернуть любой элемент, если выполняется в параллельном потоке.

Вот что может случиться, когда оператор `filter()` встречается раньше `findFirst()` или `findAny()`. Оператор `parallelStream()` вправе разбить входной поток, скажем, на две половины и отфильтровать каждую часть независимо. Если во время фильтрации второй половины подходящее значение будет найдено раньше, то `findAny()` вернет его, пусть даже в первой половине тоже существуют подходящие значения. В то же время `findFirst()` гарантированно возвращает самое первое подходящее значение, поэтому должен дожидаться результата фильтрации обеих половин. У обоих методов есть свои достоинства, но применять их нужно с открытыми глазами.

В идеале, приняв во внимание закон Амдала (https://ru.wikipedia.org/wiki/Закон_Амдала) для машины с четырьмя процессорами, можно ожидать четырехкратного ускорения. Но у параллельных потоков есть недостатки. Прежде всего, если поток мал, а конвейер преобразований короткий, то затраты на контекстные переключения могут оказаться настолько высокими, что параллельный поток окажется медленнее последовательного. Проблема слишком мелкой конкурентности не чужда и RxJava, поэтому она поддерживает декларативную конкурентность с помощью диспетчеров (см. раздел «Что такое диспетчер?» главы 4). Ситуация с параллельными потоками иная.

Вы никогда не задавались вопросом, почему потоки называются *параллельными*, а не *конкурентными*? Параллельные потоки предназначены только для счетных задач, в них зашит пул потоков (точнее, `ForkJoinPool`), в котором число потоков точно равно числу имеющихся процессорных ядер. Этот пул доступен статически и глобально с помощью метода `ForkJoinPool.commonPool()`. Все параллельные потоки, а также некоторые обратные вызовы из класса `CompletableFuture` внутри одной и той же JVM разделяют этот пул. Еще раз – все параллельные потоки во всей JVM (т. е. в нескольких приложениях, если вы развертываете WAR-файлы на

серверах приложений) сообща пользуются одним и тем же небольшим пулом. Вообще говоря, это неплохо, потому что параллельные потоки проектировались для распараллеливания задач, потребляющих 100 % процессорного времени. Таким образом, если несколько параллельных потоков работают одновременно, то они конкурируют за процессоры, несмотря ни на что.

Но допустим, что какое-то эгоистичное приложение запустило в параллельном потоке операцию ввода-вывода:

```
// НЕ ДЕЛАЙТЕ ТАК
people
    .parallelStream()
    .forEach(this::publishOverJms);
```

Метод `publishOverJms()` посылает сообщение JMS для каждого человека в потоке `people`. Мы сознательно выбрали отправку через JMS. Эта система кажется быстрой, но из-за гарантий доставки отправка сообщения, скорее всего, затронет всю сеть (чтобы уведомить брокер сообщений) или диск (чтобы сохранить сообщение локально). Этой крохотной задержки ввода-вывода достаточно, чтобы надолго задержать драгоценные потоки из пула `ForkJoinPool.commonPool()`. И хотя эта программа вообще не использует процессор, никакой другой код во всей JVM не может использовать параллельный поток. А теперь представьте, что речь идет не об отправке сообщения JMS, а о запросе данных у веб-сервиса или о накладном запросе к базе данных. Оператор `parallelStream()` следует использовать только для чисто счетных задач, иначе производительность JVM серьезно пострадает.

Это вовсе не значит, что параллельные потоки – зло. Но из-за фиксированного пула потоков их применение сильно ограничено. Разумеется, параллельные потоки из JDK – не замена `Observable.flatMap()` или другим механизмам конкурентности. Параллельные потоки оптимальны, когда работают ... ну да, параллельно. Если же конкурентные задачи потребляют не все 100 % времени процессоров, а, например, блокируются в ожидании завершения сетевой или дисковой операции, то лучше воспользоваться другими механизмами.

Зная об ограничениях потоков, давайте сравним будущие объекты и RxJava и решим, когда лучше применять одно, а когда – другое.

Выбор подходящей абстракции конкурентности

Ближайшим аналогом `CompletableFuture` в RxJava является тип `Single`. Можно использовать также `Observable`, памятуя о том, что он может порождать произвольное число событий. Важное различие между будущими объектами и типами RxJava состоит в том, что последние ленивые. Имея ссылку на `CompletableFuture`, можно быть уверенным, что фоновое вычисление уже началось, тогда как объекты `Single` и `Observable`, как правило, начинают работать только после того, как кто-то на них подпишется. Зная об этой семантической тонкости, вы легко сможете подменить `CompletableFuture` на `Observable` (см. раздел «`CompletableFuture`

и потоки» главы 5) или `single` (см. раздел «Интероперабельность с `Observable` и `CompletableFuture`» главы 5).

В редких случаях, когда результат асинхронного вычисления недоступен или несуществен, используются типы `CompletableFuture<Void>` или `Observable<Void>`. С первым все понятно, а второй может означать и потенциально бесконечный поток событий. Тип `rx.Single<Void>` выглядит так же плохо, как будущее значение типа `Void`. Потому-то и был введен тип `rx.Completable`. Использование `Completable` оправдано, когда в системе много операций, которым нельзя приписать осмысленный результат (но которые могут завершаться исключением). Примером такой архитектуры может служить разделение команд и запросов (`command-query separation` – CQS)⁶, где команды асинхронны и по определению не имеют результата.

Когда выбирать *Observable*?

Если приложение имеет дело с потоком событий во времени (например, входы пользователей в систему, события GUI, push-уведомления), с типом `Observable` ничто не может сравниться. Мы ни разу не упомянули об этом, но, начиная с версии 1.0, в Java имеется тип `java.util.Observable`, который позволяет регистрировать и уведомлять объекты `Observer`. Однако ему недостает следующей функциональности:

- возможность композиции (нет операторов);
- поддержка универсальных типов (интерфейс `Observer` содержит один метод `update()`, который принимает значение типа `Object`, представляющее произвольную полезную нагрузку, передаваемую вместе с уведомлением);
- производительность (всюду используется ключевое слово `synchronized`, а на внутреннем уровне – класс `java.util.Vector`);
- разделение обязанностей (в некотором смысле в одном интерфейсе объединены `Observable` и `PublishSubject`);
- поддержка конкурентности (все наблюдатели получают уведомления последовательно);
- неизменяемость.

Тип `Observable` из JDK – лучшее, что может предложить стандартный язык Java в части декларативного моделирования событий, его место сразу за методами `addListener()` в пакетах для разработки GUI. Если в вашей предметной области явно упоминаются события или потоки данных, то у типа `rx.Observable<T>` найдется мало конкурентов. Выразительность декларативного описания в сочетании с широким спектром операторов позволяет решать многие задачи. В случае холодных `Observable` можно воспользоваться противодавлением для управления пропускной способностью, а для горячих `Observable` имеется много операторов управления потоком, например `buffer()`.

⁶ <https://ru.wikipedia.org/wiki/CQRS>.

Потребление памяти и утечки

Основная идея библиотеки RxJava – обработка потоков событий в памяти и на лету. Она предоставляет единообразный развитый API, абстрагирующий детали источника событий. В идеале в памяти должно храниться только очень ограниченное множество событий на пути от порождающего их производителя к потребителю, который сохраняет события или переправляет другому компоненту. В действительности некоторые компоненты, особенно при неправильном использовании, могут потреблять неограниченный объем памяти. Но понятно, что память все-таки ограничена, и рано или поздно дело кончится исключением `OutOfMemoryError` или бесконечным циклом сборки мусора. В этом разделе мы приведем несколько примеров неконтролируемого потребления и утечек памяти в RxJava и расскажем, как их предотвратить. Частный случай утечки памяти вследствие отсутствия отписки был рассмотрен выше в разделе «Предотвращение утечек памяти в компонентах Activity», посвященном Android.

Операторы, потребляющие неконтролируемый объем памяти

Это те операторы, которые могут потратить сколько угодно памяти в зависимости от характера потока. Мы рассмотрим лишь некоторые из них и попытаемся предложить защитные меры, предотвращающие утечку памяти.

distinct() с кэшированием всех встречавшихся событий

Оператор `distinct()`, по определению, должен хранить все ключи, встретившиеся с момента подписки. Перегруженный вариант без параметров сравнивает каждое входящее событие с содержимым внутреннего кэша. Если такого события (тождественность определяется с помощью метода `equals()`) еще не было, то оно передается дальше и добавляется в кэш. Из кэша никогда ничего не вытесняется, чтобы гарантировать отсутствие повторяющихся событий. Понятно, что если события достаточно большие или происходят часто, то внутренний кэш неограниченно растет, и это влечет за собой утечку памяти.

Для демонстрации возьмем такое событие, моделирующее большой блок данных:

```
class Picture {

    private final byte[] blob = new byte[128 * 1024];
    private final long tag;

    Picture(long tag) { this.tag = tag; }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Picture)) return false;
```

```

    Picture picture = (Picture) o;
    return tag == picture.tag;
}

@Override
public int hashCode() {
    return (int) (tag ^ (tag >>> 32));
}

@Override
public String toString() {
    return Long.toString(tag);
}
}

```

Следующая программа выполняется в окружении, где памяти очень мало (-mx32M – куча размером 32 МБ), и порождает события максимально быстро:

```

Observable
    .range(0, Integer.MAX_VALUE)
    .map(Picture::new)
    .distinct()
    .sample(1, TimeUnit.SECONDS)
    .subscribe(System.out::println);

```

Очень скоро программа завершится с исключением `OutOfMemoryError`, поскольку во внутреннем кэше оператора `distinct()` больше нет места для хранения экземпляров `Picture`. Непосредственно перед крахом процессор был занят чуть ли не на 100 %, поскольку сборщик мусора из всех сил пытался освободить хоть немного места. Даже если бы мы использовали в качестве ключа, различающего события, не весь объект `Picture`, а только поле `Picture.tag`, программа все равно «грохнулась» бы, только намного позже:

```
distinct(Picture::getTag)
```

Утечки такого типа намного опаснее. Проблема медленно нарастает, но мы этого не замечаем, пока программа, наконец, не упадет в самый неподходящий момент, часто при высокой нагрузке. Чтобы доказать, что именно `distinct()` виноват в утечке памяти, запустим похожую программу, в которой вместо `distinct()` подсчитывается число событий в секунду без какой-либо буферизации. Точные цифры зависят от конкретной среды, но можно ожидать, что число обработанных в секунду событий достигает сотен тысяч, и при этом не потребляется много памяти и не создается ощутимая нагрузка на сборщик мусора.

```

Observable
    .range(0, Integer.MAX_VALUE)
    .map(Picture::new)
    .window(1, TimeUnit.SECONDS)
    .flatMap(Observable::count)
    .subscribe(System.out::println);

```

И как все-таки избежать утечек памяти из-за оператора `distinct()`?

- Старайтесь вообще избегать `distinct()`. Несмотря на внешнюю простоту этот оператор опасен при неправильном использовании.
- Разумно подходите к выбору ключа. В идеале он должен быть коротким. Тип перечисления и `byte` годятся, `long` и `String` — не очень. Если невозможно доказать, что у данного типа крайне ограниченное число возможных значений (как у `enum`), то возникает риск утечки памяти.
- Рассмотрите возможность использования оператора `distinctUntilChanged()`, который отслеживает только повтор последнего события, а не всех.
- А нужна ли вообще уникальность или это требование можно ослабить? Быть может, заранее известно, что дубликаты могут отстоять друг от друга не более чем на 10 секунд? Тогда попробуйте выполнять `distinct()` в небольшом окне:

```
Observable
    .range(0, Integer.MAX_VALUE)
    .map(Picture::new)
    .window(10, TimeUnit.SECONDS)
    .flatMap(Observable::distinct)
```

Раз в 10 секунд мы создаем новое окно (см. раздел «Буферизация событий в списке» главы 6) и гарантируем, что в нем не будет ни одного дубликата. Оператор `window()` порождает поток событий, произошедших внутри временного окна. Уникальные (в смысле `distinct()`) значения в этом окне передаются далее сразу же. Когда 10-секундное окно закончится, будет создано новое, но самое главное — сборщик мусора освободит кэш, ассоциированный со старым окном. Конечно, и в течение 10 секунд может накопиться столько событий, что будет возбуждено исключение `OutOfMemoryError`, поэтому лучше использовать окно фиксированного размера (например, `window(1000)`), а не фиксированной длительности. Кроме того, если не повезет, и неразличимые события возникнут вблизи правого конца предыдущего окна и левого конца следующего, то мы не обнаружим дубликаты. Это компромисс, о котором следует помнить.

Буферизация событий с помощью `toList()` и `buffer()`

Совершенно очевидно, что оператор `toList()` может потреблять неограниченный объем памяти. К тому же, применять `toList()` к бесконечным потокам вообще бессмысленно. `toList()` порождает единственное событие по завершении входного потока, а если завершения не наступит, то `toList()` не породит ничего. Однако же в памяти события будут накапливаться. Использовать `toList()` для очень длинных потоков тоже вряд ли стоит. Найдите способ потреблять события на лету или, по крайней мере, ограничьте число входящих событий с помощью оператора `take()` или ему подобного.

Оператор `toList()` имеет смысл, когда нужно просмотреть сразу все события из конечного `Observable`. Так бывает редко, обычно можно применить предикаты (типа `allMatch()` или `anyMatch()`), подсчитать элементы (`count()`) или редуциро-

вать их в одно агрегированное значение (`reduce()`), не храня все события в памяти. Одна из возможных ситуаций – преобразование `Observable<Observable<T>>` в `Observable<List<T>>`, где длина внутреннего `Observable` известна и фиксирована:

```
.window(100)
.flatMap(Observable::toList)
```

Это эквивалентно вызову:

```
.buffer(100)
```

Что и подводит нас к оператору `buffer()`. Прежде чем использовать его, хорошенько подумайте, а так ли вам нужен список `List<T>` всех событий, произошедших в заданном промежутке времени. Быть может, хватит и `Observable<T>`? Допустим, к примеру, что имеется поток `Observable<Incident>` и требуется узнать, было ли зарегистрировано более пяти происшествий с высоким приоритетом в течение каждой секунды. Мы хотим сгенерировать поток `Observable<Boolean>`, который раз в секунду порождает `true`, если в течение этой секунды было много высокоприоритетных происшествий, и `false` в противном случае. Оператор `buffer()` позволяет сделать это легко и просто:

```
Observable<Incident> incidents = //...

Observable<Boolean> danger = incidents
    .buffer(1, TimeUnit.SECONDS)
    .map((List<Incident> oneSecond) -> oneSecond
        .stream()
        .filter(Incident::isHighPriority)
        .count() > 5);
```

Однако оператор `window()` не требует буферизации событий в промежуточном списке, а переправляет их на лету. Поэтому `window()` столь же удобен для решения этой задачи, но потребляет постоянный объем памяти:

```
Observable<Boolean> danger = incidents
    .window(1, TimeUnit.SECONDS)
    .flatMap((Observable<Incident> oneSecond) ->
        oneSecond
            .filter(Incident::isHighPriority)
            .count()
            .map(c -> (c > 5))
    );
```

Тип `Observable` предоставляет гораздо более развитый API по сравнению с типом `Stream` из JDK, поэтому вы, возможно, захотите преобразовать коллекцию Java в `Observable` просто ради более удобных операторов. Например, потоки JDK не поддерживают ни скользящие окна, ни скрепление.

Таким образом, если есть возможность, используйте `window()`, а не `buffer()`, особенно когда размер внутреннего списка, который строит `buffer()`, невозможно предсказать.

Кэширование с помощью `cache()` и `ReplaySubject`

Оператор `cache()` — еще один очевидный пожиратель памяти. Он даже хуже, чем `distinct()`, потому что хранит ссылки на все вообще входящие события. Использовать кэш имеет смысл только для `Observable`, которые заведомо имеют небольшую фиксированную длину. Например, если `Observable` служит для моделирования асинхронного ответа от какого-то компонента, использовать `cache()` безопасно и желательно. В противном случае появление каждого `Observer` инициирует повторную отправку запроса, что может иметь непредвиденные побочные эффекты. Наоборот, кэширование длинных, потенциально бесконечных `Observable`, особенно горячих, практически лишено смысла. В случае горячих `Observable` старые события, скорее всего, вообще не интересны.

Все то же самое относится к `ReplaySubject` (см. раздел «Класс `rx.subjects.Subject`» главы 2). Все, что помещено в такой объект `Subject`, должно сохраняться, так чтобы последующие подписчики получали все события, а не только случившиеся после подписки. Рекомендации по использованию `cache()` и `ReplaySubject` очень схожи. Если решите использовать их, то проследите за тем, чтобы кэшируемый источник был конечным и сравнительно коротким. И постарайтесь по возможности не хранить ссылку на кэшированный `Observable` слишком долго, чтобы ее можно было со временем убрать в мусор.

Противодавление сокращает потребление памяти

Вспомните, как мы скрепили два источника, порождавших события в разном темпе (раздел «Когда потоки не синхронизированы: `combineLatest()`, `withLatestFrom()` и `amb()`» главы 3). Если мы пытаемся скрепить два источника, один из которых чуть медленнее другого, то оператор `zip()` (или `zipWith()`) должен сохранять события из быстрого источника во временном буфере до тех пор, пока не поступят события из медленного:

```
Observable<Picture> fast = Observable
    .interval(10, MICROSECONDS)
    .map(Picture::new);
Observable<Picture> slow = Observable
    .interval(11, MICROSECONDS)
    .map(Picture::new);

Observable
    .zip(fast, slow, (f, s) -> f + " : " + s)
```

Думаете, этот код в конечном итоге завершится исключением `OutOfMemoryError`, поскольку `zip()` все добавляет и добавляет⁷ события от `fast` в буфер, поджидая `slow`? Но это не так; на самом деле, мы почти сразу получаем кошмарное исключение `MissingBackpressureException`. Оператор `zip()` (и `zipWith()`) не хочет слепо принимать события в темпе, который навязывает источник. Вместо этого он поль-

⁷ Пока в RxJava не появилось противодавление, `zip()` именно так и работал. Несинхронизированные потоки могли привести к медленно нарастающей утечке памяти. Но после включения противодавления в версию 0.20.0-RC2 оператор `zip()` был переработан.

зуется противодавлением и запрашивает как можно меньше данных. Поэтому, если входные объекты `Observable` холодные и реализованы правильно, то `zip()` просто замедлит более быстрый `Observable`, запрашивая меньше данных, чем тот теоретически способен выдать.

Но в случае `interval()` этот механизм не работает. Оператор `interval()` холодный, т. к. начинает отсчет только после подписки, и каждый подписчик получает свой независимый поток. Тем не менее, после того как мы подписались на `interval()`, замедлить его нет никакой возможности, потому он, по определению, обязан порождать события с определенной частотой. Поэтому он вынужден игнорировать запросы противодавления, что с большой вероятностью приведет к `MissingBackpressureException`. Единственное, что можно сделать, – отбрасывать лишние события (см. раздел «Производители и отсутствие противодавления» главы 6).

```
Observable
    .zip(
        fast.onBackpressureDrop(),
        slow.onBackpressureDrop(),
        (f, s) -> f + " : " + s)
```

Но чем `MissingBackpressureException` лучше `OutOfMemoryError`? Тем, что игнорирование противодавления приводит к быстрому прекращению, а память может течь медленно. Зато исключение `MissingBackpressureException` может произойти в самый неожиданный момент, например, во время сборки мусора. В разделе «Проверка порожденных событий» главы 7 объясняется, как проверять поведение противодавления в автономных тестах.

Резюме

Приступить к использованию RxJava гораздо проще, если в коде уже есть какой-то источник `Observable`. Реализация нового `Observable` с нуля чревата ошибками, поэтому если библиотека (например, Hystrix, Retrofit, клиентские драйверы базы данных) поддерживает RxJava, то начать много легче. В разделе «От коллекций к `Observable`» главы 4 мы постепенно и с трудом переделали существующее приложение, заменив императивный, ориентированный на коллекции подход другим – декларативным и основанным на потоках. Но добавление библиотек, являющихся источниками асинхронных объектов `Observable`, значительно упрощает такой рефакторинг. Чем больше потоков в приложении, тем дальше распространяется реактивный API. Всё начинается на уровне сбора данных (база данных, веб-сервис и т. п.) и поднимается на уровень служб и веба. И вот уже весь стек стал реактивным. В какой-то момент, когда использование RxJava достигает критической массы, нужда в операторе `toBlocking()` отпадает, потому что потоки используются снизу доверху.

Глава 9.

Направления будущего развития

Мы долго оставались на уровне версий 0.x, прежде чем зафиксировали API в RxJava 1.0, так что это достаточно зрелая и стабильная версия. Кроме того, мы приняли решение поддерживать «экспериментальный» и «бета» API, поэтому можем продолжать эксперименты, прежде чем перевести API на «финальный» уровень. Однако на этапе 0.x/1.x все же было принято несколько решений, требующих выпуска новой не полностью совместимой версии, в связи с чем ведутся работы над версией 2.0 (<https://github.com/ReactiveX/RxJava/blob/2.x/DESIGN.md>).

В принципе, она будет очень похожа на 1.x, так что не потребуются ни значительного изменения мыслительного процесса, ни кардинальных перемен в порядке использования. Даже с выходом версии 2.0 эта книга останется в большинстве случаев актуальной. Так зачем же нужна вторая версия?

Реактивные потоки

Главная причина – встроенная поддержка Reactive Streams API (<http://www.reactive-streams.org/>). Несмотря на то, что команда RxJava принимала участие в совместной работе, которая привела к реактивным потокам, все API в версии RxJava v1 были уже зафиксированы, и включить интерфейсы Reactive Streams не представлялось возможным. Поэтому для RxJava версии 1 необходим адаптер, хотя семантически она ведет себя в основном, как Reactive Streams. В версии 2 типы Reactive Streams будут реализованы непосредственно и в полном соответствии со спецификацией, что обеспечит более качественную поддержку интероперабельности в сообществе Java.

Типы Observable и Flowable

Вторая причина – разделить тип Observable на два: Observable и Flowable. Было ошибкой требовать, чтобы все операторы поддерживали противодействие, потому что оно необходимо не во всех случаях. Оно, конечно, влечет небольшое снижение производительности, но главное, почему это признано ошибкой, – дополнительное мысленное усилие, необходимое для использования Observable, и значительные трудности при создании пользовательских операторов.

В ситуациях чистого проталкивания должна быть возможность использовать `Observable`, как первоначально задумывал Эрик Мейер, – без учета семантики `request(n)` из `Reactive Streams`. Такие ситуации встречаются очень часто. По сути дела, любое использование в пользовательском интерфейсе, в частности на платформе `Android`, – пример чистого проталкивания; `request(n)` при этом в лучшем случае вносит путаницу и без необходимости усложняет задачу. Да, операторы вида `onBackpressureDrop` могут быть весьма полезны в таких ситуациях, но их включение должно быть осознанным выбором пользователя.

Поэтому в версии 2 тип `Observable` вернется к чистому проталкиванию без `request(n)` и не будет реализовывать ни типы, ни спецификацию `Reactive Streams`. Будет добавлен новый тип `Flowable`, который станет «`Observable` с противодавлением» и будет реализовать тип и спецификацию `Publisher` из `Reactive Streams`. Имя «`Flowable`» было выбрано под влиянием пакета `java.util.concurrent.Flow` из `Java 9`, который вобрал в себя интерфейсы `Reactive Streams`.

Заодно использование `Observable` и `Flowable` в публичных API точнее информирует о поведении источника данных. Если это `Observable`, то источник проталкивает данные, и потребитель должен быть готов к их приему. Если это `Flowable`, то производится вытягивание с проталкиванием и отправляется лишь столько элементов, сколько запросил потребитель. Наведение мостов между двумя типами будет возможно и устроено так, как сейчас в `RxJava v1`, только гораздо более явно, например, метод `observable.toFlowable(Strategy.DROP)` будет преобразовывать `Observable` в `Flowable` с указанием стратегии противодавления, применяемой, когда данные проталкиваются быстрее, чем потребитель может их обработать.

Производительность

И последняя важная причина разработки версии 2 – желание повысить общую производительность (сократить накладные расходы), поскольку мы больше не связаны архитектурными ограничениями версии 1. Частично это достигается путем уменьшения числа операций выделения памяти при построении цепочки операторов, подписки на нее и выполнения. По умолчанию объекты `Subscriber` больше не обернуты объектом `SafeSubscriber` (для этой цели служит метод `Flowable.safeSubscribe()`), и отпала необходимость вызывать метод `cancel` (`unsubscribe` в терминологии версии 2) для отмены цепочки при появлении терминального события.

Второй источник повышения производительности – техника внутренней оптимизации под названием *слияние операторов* (*operator-fusion*) (расширяющая протокол `Reactive Streams`). Она существенно уменьшает накладные расходы на противодавление и управление очередью во многих типичных конфигурациях синхронных потоков (а иногда и в асинхронных потоках). В некоторых тестах пропускная способность потоков с противодавлением оказывалась всего на 20–30 % хуже, чем у потоков `Java 8` (синхронное вытягивание), тогда как в версии 1 она была хуже на 100–200 %.

Миграция

Поскольку RxJava глубоко укоренилась в различных приложениях, перейти на несовместимую версию будет очень нелегко. Поэтому у версии 2 другое имя пакета и артефакт Maven, так что обе версии могут сосуществовать в одном приложении.

Пакет v1	Пакет v2	Maven v1	Maven v2
rx.*	io.reactivex.*	io.reactivex:rxjava	io.reactivex.rxjava2:rxjava

Миграция с RxJava версии 1 на версию 2 сводится в основном к следующим действиям:

1. Заменить пакет `rx.` на `io.reactivex.`
2. Если необходимо противодействие, заменить `Observable` на `Flowable`.

RxJava v2 находится в ветви 2.x на GitHub, а в документе `DESIGN.md` (<https://github.com/ReactiveX/RxJava/blob/2.x/DESIGN.md>) силами сообщества описаны проектные решения, принятые во второй версии. Дополнительные сведения о различиях между версиями 1 и 2 можно найти на странице по адресу <https://github.com/ReactiveX/RxJava/wiki/What's-different-in-2.0>.

Приложение А.

Дополнительные примеры HTTP-серверов

Это приложение дополняет раздел «Решение проблемы C10k» главы 5 и содержит дополнительные примеры HTTP-серверов. Для понимания главы 5 они не обязательны, но могут представлять интерес сами по себе. Кроме того, некоторые из них участвуют в тестах производительности.

Системный вызов `fork()` в программе на C

Мы попытаемся реализовать конкурентный HTTP-сервер на языке C. Если вы знакомы с этим языком, то понять приведенную ниже программу будет нетрудно. В противном случае не огорчайтесь, достаточно уловить общую идею, детали не так существенны. Вызов `fork()` создает копию текущего процесса, так что в течение короткого времени существуют оба процесса: исходный (родитель) и его потомок. Во втором процессе точно такие же переменные и состояние, а единственное различие — значение, возвращенное `fork()`:

```
#include <signal.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    signal(SIGCHLD, SIG_IGN);
    struct sockaddr_in serv_addr;
    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = INADDR_ANY;
    serv_addr.sin_port = htons(8080);
    int server_socket = socket(AF_INET, SOCK_STREAM, 0);
    if(server_socket < 0) {
        perror("socket");
```

```

    exit(1);
}
if(bind(server_socket,
        (struct sockaddr *) &serv_addr,
        sizeof(serv_addr)) < 0) {
    perror("bind");
    exit(1);
}
listen(server_socket, 100);
struct sockaddr_in cli_addr;
socklen_t clilen = sizeof(cli_addr);
while (1) {
    int client_socket = accept(
        server_socket, (struct sockaddr *) &cli_addr, &clilen);
    if(client_socket < 0) {
        perror("accept");
        exit(1);
    }
    int pid = fork();
    if (pid == 0) {
        close(server_socket);
        char buffer[1024];
        while(1) {
            if(read(client_socket,buffer,255) < 0) {
                perror("read");
                exit(1);
            }
            if(write(client_socket,
                "HTTP/1.1 200 OK\r\nContent-length: 2\r\n\r\nOK", 40) < 0) {
                perror("write");
                exit(1);
            }
        }
    } else {
        if(pid < 0) {
            perror("fork");
            exit(1);
        }
    }
    close(client_socket);
}
return 0;
}

```

Самое важное здесь – вызов `fork()`. В родительском процессе он возвращает идентификатор (PID) процесса-потомка, а в потомке (копии исходного процесса) – значение 0. В некотором смысле `fork()` выполняется (в родительском процессе) один раз, а возвращается дважды. Обнаружив, что мы находимся в процессе-потомке (`fork() == 0`), мы должны обрабатывать клиентские подключения. За серверный сокет `server_socket` отвечает родитель, поэтому в потомке он закрывается. Одновременно (конкурентно!) родитель закрывает клиентский сокет `client_socket` (но в потомке он остается открытым) и может принимать другое

клиентское подключение. Разумеется, родитель может породить много потомков, за счет чего достигается высокий уровень конкурентности.

Один поток – одно подключение

Зная, что одного потока недостаточно для надлежащего масштабирования сервера (см. раздел «Традиционные HTTP-серверы на основе потоков» главы 5), мы готовы переписать код с использованием приемов многопоточного программирования. Но прежде чем переходить к реализации, немного переработаем класс `SingleThread`, чтобы в дальнейшем избежать дублирования.

```
abstract class HttpServer {

    void run(int port) throws IOException {
        final ServerSocket serverSocket = new ServerSocket(port, 100);
        while (!Thread.currentThread().isInterrupted()) {
            final Socket client = serverSocket.accept();
            handle(new ClientConnection(client));
        }
    }

    abstract void handle(ClientConnection clientConnection);
}
```

Класс `ClientConnection`:

```
import org.apache.commons.io.IOUtils;

class ClientConnection implements Runnable {

    public static final byte[] RESPONSE = (
        "HTTP/1.1 200 OK\r\n" +
        "Content-length: 2\r\n" +
        "\r\n" +
        "OK").getBytes();

    public static final byte[] SERVICE_UNAVAILABLE = (
        "HTTP/1.1 503 Service unavailable\r\n").getBytes();

    private final Socket client;

    ClientConnection(Socket client) {
        this.client = client;
    }

    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                readFullRequest();
                client.getOutputStream().write(RESPONSE);
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

```

        IOUtils.closeQuietly(client);
    }
}

private void readFullRequest() throws IOException {
    BufferedReader reader = new BufferedReader(
        new InputStreamReader(client.getInputStream()));
    String line = reader.readLine();
    while (line != null && !line.isEmpty()) {
        line = reader.readLine();
    }
}

public void serviceUnavailable() {
    try {
        client.getOutputStream().write(SERVICE_UNAVAILABLE);
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
}
}

```

Это обычный рефакторинг: мы переместили в базовый класс общий код, в частности, цикл прослушивания клиентских подключений. Кроме того, обработка клиентского соединения вынесена в отдельный класс `ClientConnection`. Дополнительный метод `serviceUnavailable()` будет использован позже. Единственная обязанность реализации самого сервера `HttpServer` – каким-то образом вызвать метод `run()` объекта `ClientConnection`, например, напрямую в переработанном классе `SingleThread`:

```

public class SingleThread extends HttpServer {

    public static void main(String[] args) throws Exception {
        new SingleThread().run(8080);
    }

    @Override
    void handle(ClientConnection clientConnection) {
        clientConnection.run();
    }
}

```

Имея базовый *каркас*, мы можем без труда разработать лучше масштабируемое приложение, которое запускает новый поток `Thread` на каждое клиентское подключение `ClientConnection`:

```

public class ThreadPerConnection extends HttpServer {

    public static void main(String[] args) throws IOException {
        new ThreadPerConnection().run(8080);
    }

    @Override

```

```
void handle(ClientConnection clientConnection) {  
    new Thread(clientConnection).start();  
}  
}
```

Воспользовавшись тем, что `ClientConnection` реализует интерфейс `Runnable`, мы просто запускаем поток `Thread` для обработки нового подключения. Теперь проблема блокирования сервера медленным клиентом стоит не так остро: обработка подключения производится в фоновом режиме, так что пока данные читаются из клиентского сокета и записываются в него, главный поток продолжает принимать запросы на новые подключения. Если два клиента захотят подключиться одновременно, главный поток просто запустит два фоновых и продолжит работу.

У создания новых потоков без ограничений есть недостатки. В 64-разрядной JVM 1.8 каждый поток по умолчанию потребляет 1024 КБ памяти (флаг `-xss`). Тысяча одновременных подключений, пусть даже простаивающих, — и вот вам 1000 потоков и 1 ГБ памяти под стеки. Но не стройте иллюзий — память для стека и кучи выделяется независимо, так что приложение потребит куда больше одного гигабайта.

Пул потоков для обслуживания подключений

На этот раз мы создадим пул потоков, которые в начале простаивают в ожидании запросов на подключение. Когда появляется новый объект, обертывающий клиентский сокет, из пула берется первый простаивающий поток. У пула потоков много преимуществ по сравнению с простым созданием потоков по запросу.

- Поток `Thread` уже инициализирован и запущен, поэтому не нужно ничего ждать, это уменьшает задержку, наблюдаемую клиентом.
- Мы ограничили общее число потоков в приложении, поэтому можем безопасно отклонять запросы на подключение при пиковой нагрузке вместо аварийного завершения.
- С пулом потоков ассоциирована настраиваемая очередь, которая может амортизировать короткие всплески нагрузки.
- На случай, когда емкости пула и очереди исчерпаны, есть возможность задать политику отклонения (ошибка, выполнение в потоке клиента вместо потока из пула и т. д.).

Если необходим полный контроль над создаваемыми потоками, то пул потоков гораздо лучше создания нового потока при каждом запросе. Но еще важнее тот факт, что мы можем строго ограничить общее число клиентских потоков и управлять пиками нагрузки:

```
class ThreadPool extends HttpServer {  
    private final ThreadPoolExecutor executor;  
  
    public static void main(String[] args) throws IOException {  
        new ThreadPool().run(8080);  
    }  
}
```

```

    }

    public ThreadPool() {
        BlockingQueue<Runnable> workQueue = new ArrayBlockingQueue<>(1000);
        executor = new ThreadPoolExecutor(100, 100, 0L,
            MILLISECONDS, workQueue,
            (r, ex) -> {
                ((ClientConnection) r).serviceUnavailable();
            });
    }

    @Override
    void handle(ClientConnection clientConnection) {
        executor.execute(clientConnection);
    }
}

```

Когда возникает необходимость обработать `ClientConnection`, мы перепоручаем задачу выделенному исполнителю `ThreadPoolExecutor`, который управляет пулом из 100 потоков. Перед этим пулом стоит ограниченная очередь на 1000 задач, а если возникнет необычно мощный шквал запросов, то в дело вступит объект `RejectedExecutionHandler`. Наш сервер просто вызывает метод `serviceUnavailable()`, который немедленно возвращает клиенту код 503 (тактика быстрого прекращения, см. также раздел «Управление отказами с помощью Nustrix» главы 8), вместо того чтобы заставлять его ждать бесконечно.

В спецификации сервлетов версии 3.0 стало возможно писать масштабируемые приложения поверх асинхронных сервлетов. Идея в том, чтобы разорвать связь между обработкой запроса и контейнерным потоком. Когда приложение захочет отправить ответ, оно сможет сделать это в любой момент времени из любого потока. Тот контейнерный поток, который получил запрос, возможно, уже не существует или занят обработкой другого запроса. Это революционная идея, но и все приложение должно быть построено соответственно. Иначе приложение оказывается более отзывчивым (контейнерный пул потоков никогда не испытывает насыщения), но если запрос должен обрабатывать какой-то другой пользовательский поток, то мы просто переместили проблему неконтролируемого роста числа потоков в другое место. Когда число потоков достигает сотен или тысяч, приложение начинает вести себя плохо, например, медленно отвечает из-за частых циклов сбора мусора и контекстных переключений.

Приложение В.

Решающее дерево для выбора операторов Observable

Цель этого приложения – помочь вам в выборе подходящего оператора из мира RxJava. При наличии более сотни вариантов становится трудновато отыскать тот встроенный оператор, который лучше всего отвечает потребностям. Текст приложения дословно скопирован из официальной документации по RxJava – раздел «A Decision Tree of Observable Operators» (<http://reactivex.io/documentation/operators.html#tree>) и публикуется на условиях лицензии Apache License Version 2.0. Но ссылки ведут на соответствующие разделы книги, а не онлайн-документации. Чаще всего оператору посвящен целый раздел, но иногда встречается лишь краткое упоминание или пример.

◆ Я хочу создать новый объект Observable...

- который порождает конкретный элемент: `just()`, см. раздел «Создание объектов Observable» главы 2;
 - который был возвращен из функции, вызванной в момент подписки: `start()`, см. модуль `rxjava-async` (<https://github.com/ReactiveX/RxJava/wiki/Async-Operators>);
 - который был возвращен из `Action`, `Callable`, `Runnable` или еще чего-то в этом роде, вызванного в момент подписки: `from()`, `fromCallable()`, `fromRunnable()`, см. разделы «Создание объектов Observable» и «Бесконечные потоки» главы 2;
 - по истечении заданной задержки: `timer()`, см. раздел «Хронометраж: операторы `timer()` и `interval()`» главы 2;
- который выбирает порождаемые элементы из указанного объекта типа `Array`, `Iterable` или чего-то подобного: `from()`, см. раздел «Создание объектов Observable» главы 2;
- путем получения значения из объекта типа `Future`: `from()`, см. раздел «Создание объектов Observable» главы 2 и раздел «`CompletableFuture` и потоки» главы 5;
- который получает свою последовательность от `Future`: `from()`, см. раздел «Создание объектов Observable» главы 2;

- который циклически порождает последовательность элементов: `repeat()`, см. раздел «Повторное использование операторов с помощью `compose()`» главы 3;
- с чистого листа, с пользовательской логикой: `create()`, см. «Подробнее о методе `Observable.create()`» главы 2;
- для каждого подписавшегося наблюдателя: `defer()`, см. раздел «О пользе лени» главы 4;
- порождающий последовательность целых чисел: `range()`, см. раздел «Создание объектов `Observable`» главы 2;
 - с заданным временным интервалом: `interval()`, см. раздел «Хронометраж: операторы `timer()` и `interval()`» главы 2;
 - по истечении заданной задержки: `timer()`, см. раздел «Хронометраж: операторы `timer()` и `interval()`» главы 2;
- завершающийся, не породив ни одного элемента: `empty()`, см. раздел «Создание объектов `Observable`» главы 2;
- который вообще ничего не делает: `never()`, см. раздел «Создание объектов `Observable`» главы 2;

◆ Я хочу создать объект `Observable` путем комбинирования других `Observable`...

- который порождает все элементы всех `Observable` в порядке получения: `merge()`, см. раздел «Обращение с несколькими объектами `Observable`, как с одним, с помощью `merge()`» главы 3;
- который порождает все элементы всех `Observable`, переходя к следующему по завершении предыдущего: `concat()`, см. раздел «Способы комбинирования потоков: `concat()`, `merge()` и `switchOnNext()`» главы 3;
- который порождает элементы, являющиеся результатом последовательного комбинирования элементов из двух или более `Observable`...
 - всякий раз как *каждый* `Observable` породит новое значение: `zip()`, см. раздел «Попарная композиция с помощью `zip()` и `zipWith()`» главы 3;
 - всякий раз как *хотя бы один* `Observable` породит новое значение: `combineLatest()`, см. раздел «Когда потоки не синхронизированы: `combineLatest()`, `withLatestFrom()` и `amb()`» главы 3;
 - с помощью операторов `and()`, `then()`, `when()` из объектов-посредников `Pattern` и `Plan`, см. модуль `rxjava-joins`;
- который порождает элементы только из последнего `Observable`, порождающего события: `switchOnNext()`, см. раздел «Способы комбинирования потоков: `concat()`, `merge()` и `switchOnNext()`» главы 3.

◆ Я хочу порождать элементы из `Observable`, применяя к ним преобразования...

- по одному за раз, применяя функцию: `map()`, см. раздел «Базовые операторы: отображение и фильтрация» главы 3;

- путем порождения всех элементов, порожденных соответствующими Observable: `flatMap()`, см. раздел «Обертывание с помощью `flatMap()`» главы 3;
 - перебирая Observable последовательно в том порядке, в котором они порождаются: `concatMap()`, см. раздел «Сохранение порядка с помощью `concatMap()`» главы 3;
 - с учетом всех предшествующих элементов: `scan()`, см. раздел «Просмотр последовательности с помощью `Scan` и `Reduce`» главы 3;
 - присоединяя к ним временную метку: `timestamp()`, см. раздел «Когда потоки не синхронизированы: `combineLatest()`, `withLatestFrom()` и `amb()`» главы 3;
 - преобразуя их в промежуток времени, прошедший до момента порождения элемента: `timeInterval()`, см. раздел «Таймаут в случае отсутствия событий» главы 7.
- ◆ Я хочу сдвинуть элементы, порожденные Observable, вперед во времени перед отправкой дальше: `delay()`, см. раздел «Откладывание событий с помощью оператора `delay()`» главы 3.
- ◆ Я хочу преобразовывать элементы и уведомления от Observable в элементы перед отправкой дальше...
- обертывая их объектами Notification: `materialize()`, см. раздел «Проверка порожденных событий» главы 7;
 - с возможностью последующего разворачивания: `dematerialize()`.
- ◆ Я хочу игнорировать все элементы, порожденные Observable, и передавать дальше только уведомления о завершении или ошибке: `ignoreElements()`, см. раздел «`flatMap()` как оператор асинхронного сцепления» главы 4.
- ◆ Я хочу повторить Observable, но добавить какие-то элементы до начала порождаемой им последовательности: `startWith()`, см. раздел «Оператор `withLatestFrom()`» главы 3;
- только если порождаемая им последовательность пуста: `defaultIfEmpty()`.
- ◆ Я хочу собрать элементы из Observable и передать их дальше в виде буферов элементов: `buffer()`, см. раздел «Буферизация событий с помощью `toList()` и `buffer()`» главы 8...
- содержащего только последние порожденные элементы: `takeLastBuffer()`.
- ◆ Я хочу разбить один Observable на несколько: `window()`, см. раздел «Скользящее окно» главы 6...
- так чтобы похожие элементы оказались в одном Observable: `groupBy()`, см. раздел «Расщепление потока по условию с помощью `groupBy()`» главы 3.

- ◆ Я хочу выбрать конкретный элемент, порожденный Observable...
 - последний элемент, порожденный перед завершением: `last()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - единственный порожденный элемент: `single()`, см. раздел «Проверка того, что Observable содержит ровно один элемент, с помощью `single()`» главы 3;
 - первый порожденный элемент: `first()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3.
- ◆ Я хочу передавать дальше только некоторые элементы, полученные от Observable...
 - путем отфильтровывания тех, что не удовлетворяют некоторому предикату: `filter()`, см. раздел «Базовые операторы: отображение и фильтрация» главы 3;
 - только первый элемент: `first()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - только несколько первых элементов: `take()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - только последний элемент: `last()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - только элемент в позиции *n*: `elementAt()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - только элементы, следующие за первыми элементами...
 - точнее, за первыми *n* элементами: `skip()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - точнее, пока не встретится элемент, удовлетворяющий предикату: `skipWhile()`, см. раздел «Таймаут в случае отсутствия событий» главы 7;
 - точнее, по истечении начального периода времени: `skip()`;
 - точнее, до того момента, как второй Observable породит какой-то элемент: `skipUntil()`.
 - только элементы, кроме последних...
 - точнее, кроме последних *n* элементов: `skipLast()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - точнее, пока элементы удовлетворяют некоторому предикату: `takeWhile()`, см. раздел «Выборка с помощью операторов `skip()`, `takeWhile()` и прочих» главы 3;
 - точнее, кроме элементов, порожденных в течение заданного промежутка времени перед завершением источника: `skipLast()`;
 - точнее, кроме элементов, порожденных после того, как второй observable породил какой-то элемент: `takeUntil()`.

- путем периодической выборки из observable: `sample()`, см. раздел «Периодическая выборка и отбрасывание событий» главы 6;
- только элементы, за которыми в течение заданного промежутка времени не было порождено ни одного элемента: `debounce()`, см. раздел «Пропуск устаревших событий с помощью `debounce()`» главы 6;
- подавляя дубликаты уже встречавшихся элементов: `distinct()`, см. раздел «Устранение дубликатов с помощью `distinct()` и `distinctUntilChanged()`» главы 3;
- отбрасывая элементы, являющиеся дубликатами предыдущего: `distinctUntilChanged()`, см. раздел «Устранение дубликатов с помощью `distinct()` и `distinctUntilChanged()`» главы 3;
- задержав подписку на него до тех пор, пока он не начнет порождать элементы: `delaySubscription()`.



Об авторах

Томаш Нуркевич – инженер-программист из Аллегро. Последние десять лет занимается написанием кода на Java, предпочитая разработку серверной части. Обоим языкам на платформе JVM и технологии с открытым исходным кодом. Часто пишет в блог на сайте DZone и выступает на конференциях по Java в разных уголках планеты. Связаться с Томашем можно по адресу в Твиттере: @tnurkiewicz и через его блог.

Бен Кристенсен – инженер-программист, специализирующийся на гибких масштабируемых распределенных системах. К числу проектов с открытым исходным кодом, отвечающих этим требованиям, относятся Hystrix (<https://github.com/Netflix/Hystrix>) и RxJava (<https://github.com/ReactiveX/RxJava>).

Об изображении на обложке

На обложке этой книги изображен гризон, или южноамериканская куница (*Galictis cuja* и *Galictis vittata*).

Гризоны достигают 61 см в длину и весят от 900 до 2700 г. Основное различие между двумя сохранившимися видами: большим и малым гризоном, — размер. Окраской гризон напоминает скунса: по лбу и до затылка проходит белая полоска меха, но тело у него крепче, шея толще, ноги короче, а хвост меньше.

Гризоны часто обитают в полуткрытых кустарниках, равнинных лесах и прериях. Живут в норах, дуплах упавших деревьев или в трещинах скал. Питаются в основном фруктами и мелкими животными.

Многие животные, изображенные на обложках книг O'Reilly, находятся под угрозой вымирания; все они важны для нашего мира. Если хотите узнать, чем вы можете помочь, зайдите на сайт animals.oreilly.com.

Предметный указатель

А

автономное тестирование 277, 279
ад обратных вызовов 22, 63
аккумулятор 107
Амдала закон 329

Б

блокирующие программы 144
буферизация
 достоинства 231
 по времени 234
 событий в списке 231
быстрое прекращение 200, 311

В

ввод-вывод
 блокирующий и неблокирующий 39
 ограничение масштабируемости 183
выборка периодическая 228

Г

горячие и холодные потоки 61, 241

Д

двойственность 33
декартово произведение 99
декларативная конкурентность 157, 177
дубликаты, устранение 110, 229

И

издатель-подписчик, паттерн 46
изменяемые аккумуляторы 108
инструментальные панели, мониторинг 318
инструменты нагрузочного тестирования 195
исключения 261

К

камешковые диаграммы
amb() 104

combineLatest() 101
concatWith() 140
distinctUntilChanged() 156
flatMap() 85
map() 82
merge 96
toList() 137
zip() 97
zipWith() 97
структура 80

количество элементов
 в Observable и Future 34
 единственное значение 36
 и композиция 36
 многозначные ответы 35
 потоки событий 34
 тип Completable 38
команда, паттерн проектирования 309
компоновщик, паттерн проектирования 167
конкурентность
 в RxJava 28
 декларативная 157, 177
 императивная 142
 и поведение subscribeOn() 171
 определение термина 28
 эффективное использование 22

Л

ленивое выполнение
 ленивое разбиение на страницы и
 конкатенация 141
 сравнение с энергичным 31

О

обработка ошибок
 декларативная замена try-catch 264
 исключения 261
 повтор после ошибки 271
 рекомендации для реактивных систем 260
таймаут 268

операторы и преобразования
 более сложные 106
 для отображения и фильтрации 79
 для работы с несколькими Observable 95
 для управления потоком 228
 достоинства 79
 камешковые диаграммы 80
 назначение операторов 79
 написание пользовательских операторов 125
 противодействие 243
 решающее дерево для выбора 347
 сцепление операторов 148
 отбрасывание событий 228
 отладка
 виртуальное время 275
 диспетчеры и автономное тестирование 277
 измерение и мониторинг 289
 обратные вызовы doOn...() 287
 отображение и фильтрация
 базовые операторы 79
 взаимно однозначное преобразование 81
 запуск асинхронного вычисления 85
 откладывание событий 90
 порядок событий 91
 сохранение порядка 93

П

пакетирование запросов 175, 313
 параллелизм 28
 параллельные потоки 327, 328
 паттерны
 асинхронное сцепление операторов 148
 декларативная конкурентность 177
 декларативная подписка 167
 замена обратных вызовов потоками 153
 императивная конкурентность 142
 использование диспетчеров 158, 180
 композиция Observable 140
 конкурентность 171
 ленивое разбиение на страницы и конкатенация 141
 многопоточность в RxJava 157
 о пользе лени 138
 пакетирование запросов 175
 переход от коллекций к Observable 135
 периодический опрос изменений 156
 переборка, паттерн 311
 периодическая выборка 228
 подслушивание, паттерн 83
 попарная композиция 97

поток диспетчеризации событий (EDT) 165
 потоки в Java 8
 выбор Observable 331
 выбор абстракции конкурентности 330
 имеющиеся абстракции 326
 полезность параллельных потоков 328
 потребление памяти и утечки
 буферизация событий 334
 и противодействие 336
 кэширование 336
 оператор distinct() 332
 при разработке для Android 294, 332
 практические примеры
 интеграция с Camel 325
 опрос баз данных NoSQL 321
 потоки Java 8 и CompletableFuture 326
 потребление памяти и утечки 332
 применение RxJava в разработке для Android 293
 управление отказами с помощью Hystrix 307
 предметно-ориентированное проектирование 121
 прерыватель, паттерн 311
 проекция 122
 проталкивание и вытягивание 23
 противодействие
 в RxJava 244
 встроенное 247
 назначение 244
 определение термина 244
 производители и исключение MissingBack-pressureException 250
 учет запрошенного объема данных 253

Р

разделение команд и запросов (CQS) 331
 рандеву, алгоритм 245
 реактивные приложения
 HTTP-клиенты 201
 абстракция одиночного элемента 220
 достоинства 183
 доступ к реляционным базам данных 205
 неблокирующие методы 211
 решение проблемы C10k 183
 реактивный манифест 260
 реактивный, определение термина 21

С

самоисцеление 313
 синхронный и асинхронный режим 25
 скользящие окна 237

словари и множества 79
события, отправляемые сервером (SSE) 320
СУБД

базы данных NoSQL 321
доступ к реляционным базам данных 205

T

таймауты 268
тестирование
на основе свойств 275
тестирование и отладка
автономное тестирование 279
мониторинг и отладка 287
обработка ошибок 260

У

управление потоком
буферизация событие в списке 231
назначение 228
периодическая выборка и отбрасывание
событий 228
пропуск устаревших событий 238
скользящие окна 237

Ф

функции
декораторы и замыкания 260
функциональное реактивное
программирование (FRP) 21

A

ab, инструмент нагрузочного тестирования 195
Akka, комплект инструментов 59
Android, разработка для
библиотека Retrofit 296
использование диспетчеров 301
обзор 293
предотвращение утечек памяти 294
события пользовательского интерфейса как
потоки 304
Apache Camel 325
Apache Commons Lang, библиотека 126
ArrayBlockingQueue 243
AsyncSubject 70

B

BehaviorSubject 70
BlockingObservable, класс 136

C

C10k, проблема 183

блокирующие и реактивные серверы 195
вида масштабируемости 184
неблокирующие HTTP-серверы 187
традиционные HTTP-серверы на основе
потоков 185

Camel 325

CompletableFuture, класс
интероперабельность с Observable 216
и потоки Java 8 326
неблокирующий код 212
преимущества 211
пример с заказом билетов 214
сравнение с Observable 213
ConnectableObservable
горячие и холодные потоки 241
единственная подписка 72
жизненный цикл 74
координация нескольких подписчиков 71

D

Dropwizard Metrics 289

E

ExecutorCompletionService 151, 212

F

File Transfer Protocol (FTP) 157
fork(), системный вызов 341

G

Graphite 290

H

HTTP-клиенты 202, 296
HTTP-серверы
блокирующие и реактивные 195
масштабирование 200
неблокирующие 187
системный вызов fork() 341
с отдельным потоком на каждое
подключение 343
с пулом потоков 345
традиционные на основе потоков 185
Hystrix
достоинства 307
пакетирование и объединение команд 313
первые шаги 308
применение для мониторинга 318

I

Iterable<T> 45

J

Jackson, библиотека 296
 java.util.stream 327
 JDBC 205
 JMeter 195
 JMS (Java Message Service) 153

K

Kafka 244, 326

L

LinkedBlockingQueue 243
 LISTEN, команда SQL 207

M

Mechanical Sympathy, форум 22
 MissingBackpressureException 250
 MongoDB 323

N

Netty 188
 NoSQL базы данных
 Couchbase 322
 MongoDB 323
 NOTIFY, команда SQL 207
 null 132

O

Observable.all(predicate) 114
 Observable.amb() 104
 Observable.ambWith() 105
 Observable.buffer() 79, 231, 335
 Observable.cache() 56, 62, 336
 Observable.collect() 109
 Observable.combineLatest() 101
 Observable.compose() 125
 Observable.concat() 114
 Observable.concatMap() 93
 Observable.concatWith() 114
 Observable.contains() 114
 Observable.count() 113
 Observable.create() 53, 228, 259
 Observable.debounce() 238
 Observable.delay() 79, 90
 Observable.distinct() 110, 157, 332
 Observable.distinctUntilChanged() 111
 Observable.doOnNext() 74, 83
 Observable.doOn...() операторы обратного
 вызова 287
 Observable.elementAt(n) 113
 Observable.elementAtOrDefault() 113
 Observable.empty() 52
 Observable.error() 52
 Observable.exists() 114
 Observable.filter()
 многократное применение 81
 назначение 79, 84
 Observable.first() 112
 Observable.firstOrDefault() 113
 Observable.flatMap()
 асинхронное сцепление 148
 использование 85
 порядок событий после 91
 управление уровнем конкурентности 93
 Observable.flatMapIterable() 87, 210
 Observable.from() 52
 Observable.fromCallable() 60
 Observable.groupBy() 121, 175
 Observable.ignoreElements() 151
 Observable.interval() 61, 229
 Observable.just() 52
 Observable.last() 112
 Observable.lastOrDefault() 113
 Observable.lift() 127
 Observable.map() 79, 81, 129
 Observable.merge() 95, 114
 Observable.never() 52
 Observable.observeOn() 177
 Observable.onBackpressureBuffer() 252
 Observable.onErrorResumeNext() 266
 Observable.onErrorReturn() 265
 Observable....OrDefault() 113
 Observable.publish() 74
 Observable.publish().refCount() 72
 Observable.range() 52, 245
 Observable.reduce() 108
 Observable.refCount() 73
 Observable.sample() 229
 Observable.scan() 106
 Observable.serialize() 60, 71
 Observable.single() 109
 Observable.singleOrDefault() 113
 Observable.skip() 112
 Observable.skipLast() 112
 Observable.startWith() 103
 Observable.subscribe()
 единственная подписка 72
 координация нескольких подписчиков 71
 наблюдение за Observable 48
 отписка 51
 управление несколькими подписчиками 55

управление подписчиками вручную 68
 Observable.subscribeOn() 146, 167
 Observable.switchOnNext() 115
 Observable<T>
 более одного 95
 более сложные операторы 106
 практический пример 63
 представление потока событий 45
 примеры 46
 создание Observable
 Observable.create() 53
 и бесконечные потоки данных 56
 распространение ошибок 60
 фабричные методы 52
 холодные и горячие потоки 61
 хронотраж с помощью timer() и
 interval() 61
 сравнение с Iterable 45
 сравнение с другими абстракциями 326
 типы порождаемых событий 47
 управление прослушивателями 50
 Observable.take() 112
 Observable.takeFirst() 113
 Observable.takeLast() 112
 Observable.takeUntil() 51, 113
 Observable.takeWhile() 113
 Observable.throttleLast() 231
 Observable.throttleWithTimeout() 239
 Observable.timeout() 268
 Observable.timer() 61
 Observable.timestamp() 101
 Observable.toBlocking() 136
 Observable.toList() 136, 334
 Observable.unsubscribe() 51
 Observable<Void> 47
 Observable.window() 237
 Observable.withLatestFrom() 102
 Observable.zip() 97
 Observable.zipWith() 97
 Observer<T> 50
 OnNext*, события 47

P

PostgreSQL 207
 PublishSubject 70

R

Reactive Streams, инициатива 244
 ReplaySubject 71, 336
 ResultSet 254
 Retrofit, библиотека 296

retrolambda, библиотека 302
 RxAndroid, проект 164
 rx.Completable, класс 327
 RxNetty 187, 202

S

Scheduler класс
 Schedulers.computation() 160, 180
 Schedulers.from() 160
 Schedulers.immediate() 161
 Schedulers.io() 159
 Schedulers.newThread() 159
 Schedulers.test() 163
 Schedulers.trampoline() 161
 в разработке для Android 301
 другие применения диспетчеров 180
 и автономное тестирование 277
 обзор деталей реализации 164
 применения 158
 Single класс
 интероперабельность с Observable и
 CompletableFuture 225
 когда использовать 226
 объединение ответов 223
 оператор zip() 223
 применения 220
 создание и потребление 221
 сравнение с другими абстракциями 326
 Spock, каркас 276
 Spring, каркас 153
 Subject, класс 70
 Subscriber<T> 51
 SyncOnSubscribe, класс 255

T

try-catch блоки 264
 Turbine 321

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.**

Оптовые закупки: тел. **(499) 782-38-89**

Электронный адрес: **books@aliants-kniga.ru.**

Томаш Нуркевич, Бен Кристенсен

Реактивное программирование с применением RxJava

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Перевод с английского *Слинкин А. А.*

Корректор *Синяева Г. И.*

Верстка *Паранская Н. В.*

Дизайн обложки *Мовчан А. Г.*

Формат 70×100¹/₁₆. Гарнитура «Петербург».

Печать офсетная. Усл. печ. л. 33,56.

Тираж 200 экз.

Веб-сайт издательства: www.dmk.ru

«Эта книга содержит глубокое и подробное изложение концепций и принципов использования реактивного программирования вообще и RxJava в частности, написанное двумя авторами, которые потратили бесчисленное количество часов на реализацию RxJava и применение ее к реальным задачам.

Если вам нужна «реактивность», то лучшего способа, чем купить и прочесть книгу, не придумаешь»

— Эрик Мейер, основатель и президент компании Applied Duality, Inc.

В наши дни, когда программы асинхронны, а быстрая реакция — важнейшее свойство, реактивное программирование поможет писать более надежный, лучше масштабируемый и быстрее работающий код. Благодаря этой книге программист на Java узнает о реактивном подходе к задачам и научится создавать программы, вобравшие в себя лучшие черты этой новой и весьма перспективной парадигмы.

Авторы приводят конкретные примеры применения RxJava для решения реальных задач на платформе Android и на сервере. Вы узнаете, как в RxJava поддерживается конкурентность и параллелизм. В книгу включен также предварительный анонс будущей версии 2.0.

- Написание программ, способных реагировать на несколько асинхронных источников входных данных без погружения в «ад обратных вызовов».
- Достижение того состояния просветления, когда начинаешь ясно понимать, как решить задачу реактивно.
- Приручение объектов Observable, порождающих данные в темпе, превышающем возможности потребителя.
- Стратегии тестирования и отладки реактивных программ.
- Эффективное применение параллелизма и конкурентности.
- Обсуждение перехода на версию RxJava 2.

Томаш Нуркевич — инженер-программист из Аллегро. Последние десять лет занимается написанием кода на Java, предпочитая разработку серверной части. Обожает языки на платформе JVM и технологии с открытым исходным кодом. Часто пишет в блог на сайте DZone и выступает на конференциях по Java в разных уголках планеты.

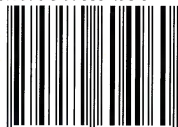
Бен Кристенсен — инженер-программист, работал в Apple, Netflix и Facebook. Начал писать на Java в 1990-х годах и внес большой вклад в проекты с открытым исходным кодом, включая Hystrix и RxJava. Бен стал активным приверженцем реактивного и функционального стиля программирования в ходе работы над Netflix API.

Интернет-магазин:
www.dmkpress.com
Книга — почтой:
orders@aliants-kniga.ru
Оптовая продажа:
“Альянс-книга”
тел. (499) 782-38-89
books@aliants-kniga.ru



www.dmk.pф

ISBN 978-5-97060-496-0



9 785970 604960 >