Оглавление

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 противодавление. 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 бессчетное 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 двум 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 слыхали 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`, противодавления 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 противодавления (*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-

ление в Твиттере (*https://twitter.com/benjchristensen/status/632287727749230592*) о поиске автора, в результате чего книга стала реальностью!

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!

И еще я благодарен Эрику, который создал Rx, столь многому научил меня и согласился написать предисловие к книге.

От Томаша

Прежде всего, я хочу сказать спасибо родителям, которые купили мне мой первый компьютер почти 20 лет назад (это был 486DX2 с 8 МБ памяти, такое не забудешь). Так началось мое путешествие в мир программирования. Несколько людей внесли свой вклад в создание этой книги. И первый среди них – Бен, который согласился написать первую и последнюю главу, а также рецензировать мой текст.

И раз уж речь зашла о рецензентах, то 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 Войцику, 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 охваты 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 противодавления 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 `unsubcribe` 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 природа 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 потоков[1] (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 дождаться 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 (Iterable)	Проталкивание (Observable)
`T next()`	`onNext(T`
`throws Exception`	`onError(Throwable)`
`returns`	`onCompleted()`

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:

```
// Объект 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	`T getData()`	`Iterable<T> getData()`
Асинхронно	`Future<T> getData()`	`Observable<T> getData()`

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://google.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 акцессоры:

```
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:

	Нуль	Один	Несколько
Синхронно	`void doSomething()`	`T getData()`	`Iterable<T> getData()`
Асинхронно	`Completable doSomething()`	`Future<T> getData()`	`Observable<T> getData()`

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

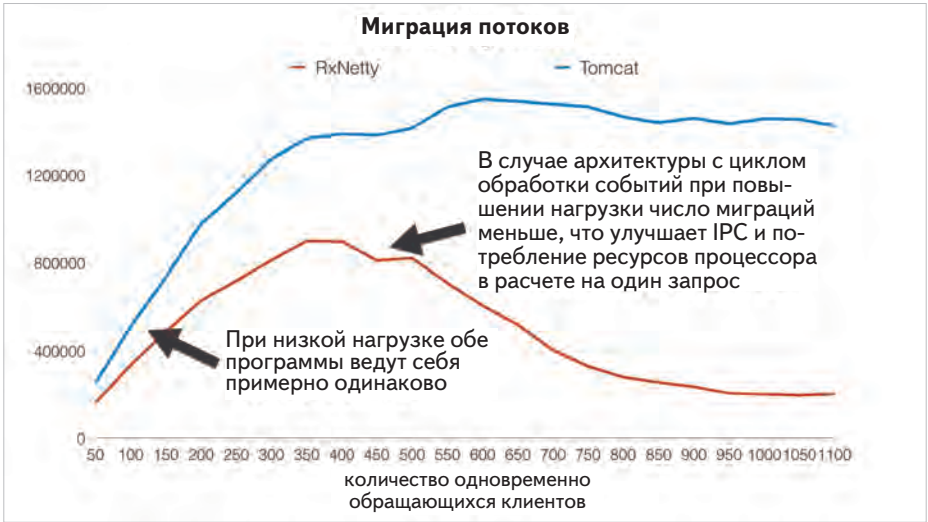
Поработав с RxJava некоторое время, я захотел получить ответ на вопрос о сравнительной эффективности блокирующего и неблокирующего ввода-вывода (обработка каждого запроса в отдельном потоке по сравнению с циклами обработки событий), но обнаружил, что получить определенные ответы очень сложно. Более того, 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 посчастливилось 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.

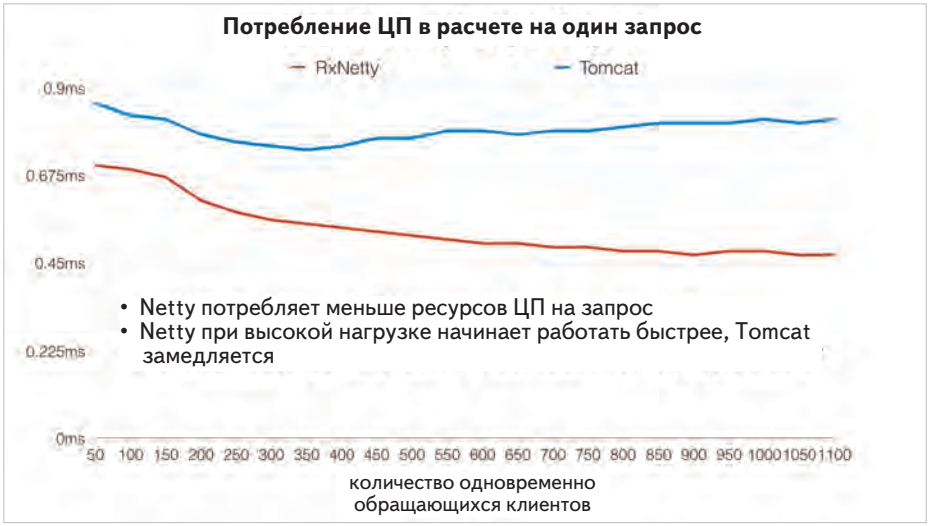
Мы специально выбрали «реальный» код – Tomcat и Netty – потому что эти программы имели непосредственное отношение к нашим производственным системам (мы уже работали с Tomcat и примеривались к Netty). Архитектура этих программ существенно различна: в одной каждый запрос обрабатывается в отдельном потоке,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

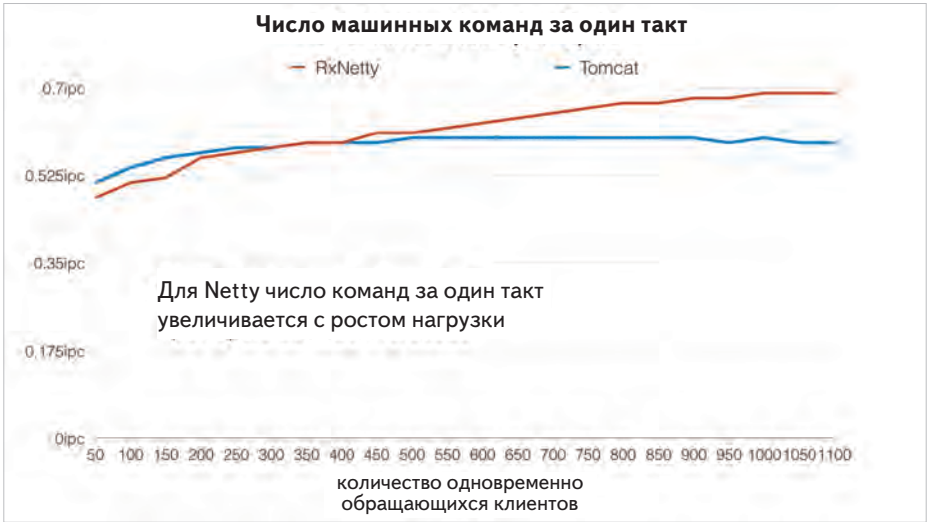
Детали нашего исследования опубликованы на GitHub по адресу *https://github.com/Netflix-Skunkworks/WSPerfLab/blob/master/test-results/RxNetty_vs_Tomcat_April2015.pdf* вместе с кодом тестов (*https://github.com/Netflix-Skunkworks/WSPerfLab/tree/master/ws-impls*). Презентацию результатов, озаглавленную «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:



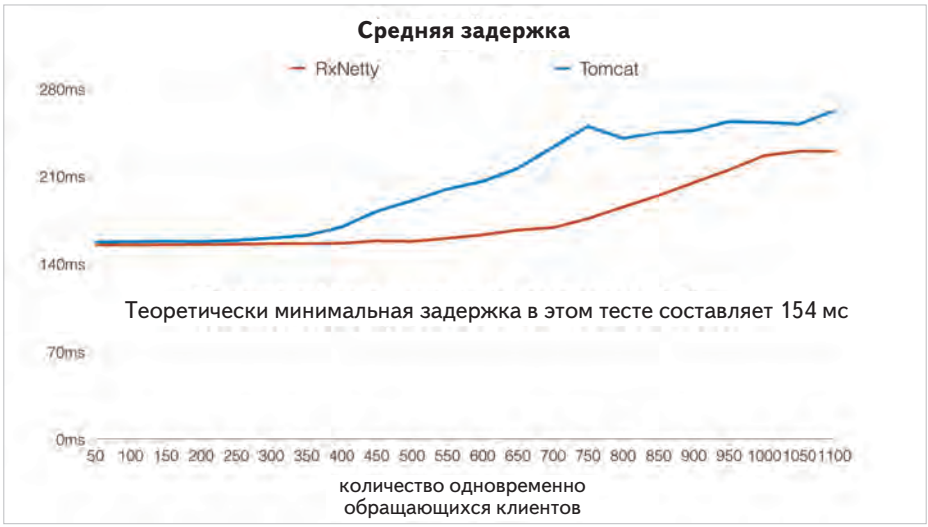
Обратите внимание, как расходятся кривые при повышении нагрузки. На этом рисунке показано количество миграций потоков. Для меня самым интересным оказался тот факт, что эффективность Netty с ростом нагрузки повышается, т. к. потоки становятся «горячими» и оказываются привязаны к одному и тому же процессорному 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.

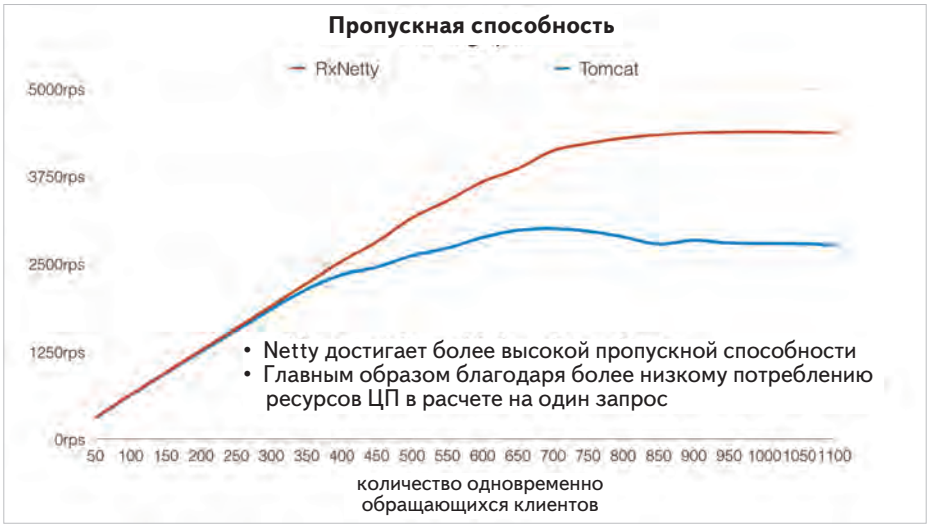
На следующих графиках показано влияние на задержку.



Хотя средние величины не очень убедительны (в отличие от процентилей), 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 требует[2] 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: 1x, 1.5x или 2 x число процессорных ядер. Я, впрочем, не обнаружил заметных различий и обычно оставляю значение по умолчанию 1x.

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: