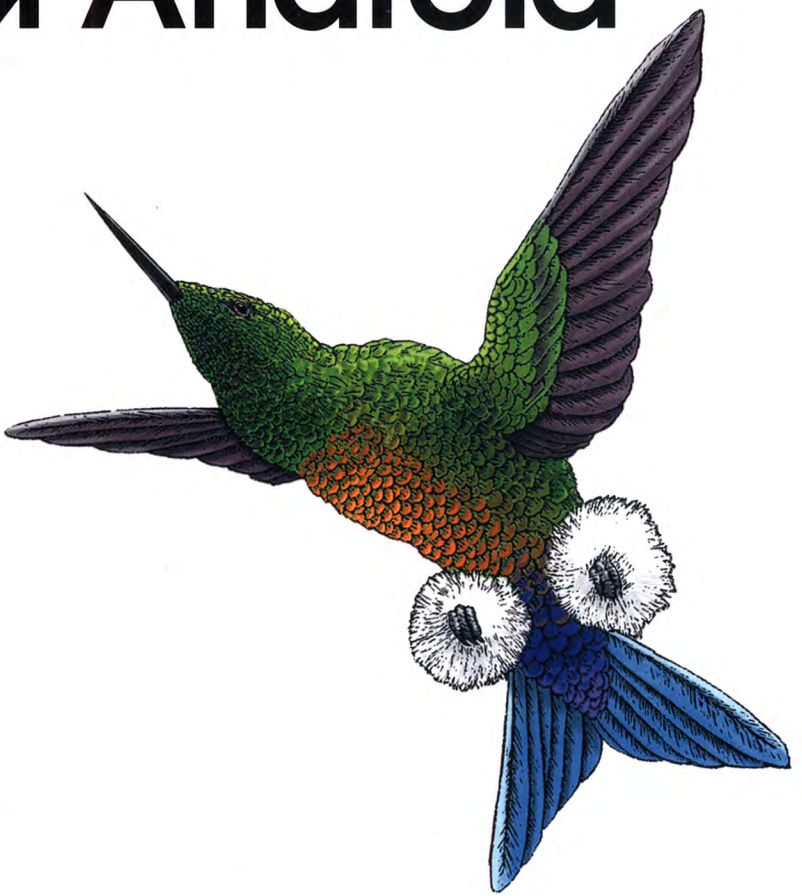
O'REILLY®

Программирование на Kotlin для Android



Пьер-Оливье Лоранс,
Аманда Хинчман-Домингес,
Дж. Блейк Мик, Майк Данн

Пьер-Оливье Лоранс,
Аманда Хинчман-Домингес,
Дж. Блейк Мик, Майк Данн

Программирование на Kotlin для Android

Содержание

Об авторах

Пьер-Оливье Лоранс (Pierre-Olivier Laurence) — ведущий инженер-программист из компании Safran Aircraft Engines, штаб-квартира которой находится недалеко от Парижа. Он начал изучать Java и системы Linux более десяти лет назад, после чего стал разрабатывать полноценные приложения для Android. Недавно, используя мощь сопрограмм Kotlin для поистине впечатляющего прироста производительности, Пьер, активно участвующий в сопровождении библиотеки с открытым исходным кодом TileView с момента ее создания, создал версию библиотеки, которая полностью написана на языке Kotlin. Он начал использовать этот язык еще на ранних этапах и участвовал в сопровождении нескольких проектов с открытым исходным кодом с момента появления этого языка в 2015 г. Пьер — заядлый поклонник издательства O'Reilly и в будущем надеется освоить еще больше технологий.

Аманда Хинчман-Домингес (Amanda Hinchman-Dominguez) — эксперт по Kotlin по программе Google Developer Expert. Она работает инженером Android в компании Groupon и активно участвует в глобальном сообществе Kotlin в качестве спикера и организатора сообщества. Начав свою карьеру в академических кругах, она получила степень бакалавра по информатике в Гриннеллском колледже. Она занималась разработкой приложений для мобильных и настольных устройств, а также веб-разработкой. Исследования в области метапрограммирования и знания, полученные в сообществе, сильно повлияли на ее увлечения и карьеру. После нескольких лет работы в отрасли страсть к Kotlin естественным образом привела Аманду к разработке приложений для Android, и с тех пор она идет только вперед.

Дж. Блейк Мик (G. Blake Meike) — старший инженер по разработке программного обеспечения в компании Amazon с более чем десятилетним опытом работы с Java. Он разрабатывал приложения, используя большинство наборов инструментов GUI и несколько платформ мобильных устройств на Java.

Майк Данн (Mike Dunn) — один из авторов книги "Нативная разработка мобильных приложений" ("Native Mobile Development"). Он работал ведущим инженером по мобильным технологиям в издательстве O'Reilly Media и внес свой вклад в разработку библиотеки Google Closure. Майк Данн — автор расширений для ExoPlayer, мультимедийного проигрывателя следующего поколения от компании Google.

Предисловие

Команда JetBrains создала Kotlin по двум причинам: отсутствие языка, который заполнил бы все пробелы в разработке для Android с использованием (устаревших) библиотек Java, и необходимость в новом языке, который позволял бы задавать тенденции, а не просто следовать им.

В феврале 2015 г. состоялся официальный анонс Kotlin версии 1.0. Kotlin лаконичен, безопасен, прагматичен и ориентирован на взаимодействие с кодом Java. Его можно использовать везде, где сегодня применяется Java: для разработки серверных приложений, приложений для Android, настольных или портативных клиентов, программирования IoT-устройств и многого другого. Kotlin довольно быстро завоевал популярность среди разработчиков, а решение компании Google использовать Kotlin в качестве официального языка разработки приложений под управлением Android вызвало резкий рост интереса к нему.

Согласно сайту для разработчиков приложений под Android (**https://developer.android.com/kotlin**) более 60% профессионалов, пишущих приложения для этой операционной системы, в настоящее время используют Kotlin.

Кривая обучения в Android довольно крутая: по общему признанию, овладеть этой операционной системой непросто. Со временем разработчикам приходится сталкиваться с непреднамеренным взаимодействием ОС Android и приложения. Данная книга предназначена для глубокого и подробного изучения подобных проблем. Мы поговорим не только о Kotlin и Java, но и о проблемах параллелизма, возникающих при использовании Android, и о том, как Kotlin умеет их решать.

Иногда мы будем сравнивать Kotlin с Java, когда считаем, что это дает более полное представление вещей (тем более, мы ожидаем, что у большинства читателей есть опыт работы с Java). На рабочих примерах мы продемонстрируем, как преодолеть этот разрыв и что основные концепции большинства операций в Kotlin больше похожи на эквиваленты в Java. Задачи будут организованы таким образом, чтобы предоставить инженерам-программистам структурированный анализ такого большого количества информации, и покажут, как сделать приложение надежным и удобным в сопровождении.

Кроме того, пользователи, знакомые с Java, в том числе и разработчики приложений для Android, обнаружат, что процесс обучения пойдет гораздо быстрее, когда мы представляем каждую распространенную задачу, используя Java и Kotlin. Там,

где это уместно, будут обсуждены различия и подводные камни одного из этих языков или их обоих, но мы надеемся предоставить небольшие и легко усваиваемые примеры, которые "просто работают" и позволяют читателю воспринимать современную парадигму и адаптироваться к ней, а также незамедлительно и инстинктивно осознать важность обновленного кода.

В то время как Kotlin полностью совместим с Java, другие варианты разработки приложений на Java (серверные приложения, настольные клиенты, промежуточное программное обеспечение и т. д.) не получили такого распространения, как в случае с Android. Во многом это связано с тем, что компания Google, занимающаяся поддержкой этой операционной системы, настоятельно "побуждает" своих пользователей вносить изменения. Пользователи регулярно переходят на Kotlin, но еще больше людей возвращаются к Java для выполнения критически важной работы. Мы надеемся, что эта книга послужит спасательным кругом, который необходим разработчику приложений для Android, чтобы чувствовать себя в безопасности, используя преимущества и простоту, представляемые Kotlin.

Кому адресована эта книга

Любому из более чем шести миллионов инженеров Android. Мы считаем, что эта книга будет полезна практически каждому разработчику, работающему с данной операционной системой. Хотя свободно владеть Kotlin будет небольшой процент из них, даже они, скорее всего, что-то почерпнут из той информации, которую мы представим. Но на самом деле эта книга рассчитана на подавляющее большинство разработчиков, которые еще не перешли на Kotlin, а также на тех, кто соприкоснулся с Kotlin, но не достиг того уровня знакомства с этим языком, который они, возможно, приобрели при разработке приложений для системы Android, ориентированной на Java.

Сценарий 1.

Читатели владеют Java, слышали о новом языке Kotlin и хотят опробовать его. Они читают какой-то онлайн-учебник и начинают его использовать, и все отлично работает. Вскоре они понимают, что это не просто новый синтаксис. Идиомы не совпадают (например, функциональное программирование, сопрограммы), и теперь им доступен совершенно новый способ разработки. Но им не хватает ориентиров, структуры. Для них эта книга — идеальный вариант.

Сценарий 2.

Читатель является частью небольшой группы разработчиков Java. Они обсуждают, стоит ли включать Kotlin в свой проект. Даже если говорят, что Kotlin на 100% совместим с Java, некоторые коллеги утверждают, что введение другого языка усложнит проект. Другие предполагают, что это может ограничить количество участников, которые смогут работать над проектом, из-за необходимости

владеть двумя языками. Читатель может использовать эту книгу, чтобы убедить своих коллег, показав, что выгоды перевешивают затраты.

Сценарий 3.

Опытный разработчик приложений для Android, возможно, экспериментировал с Kotlin или написал на нем что-то, но все равно возвращается к исходной базе на Java, когда что-то нужно сделать. Это тот сценарий, в котором оказались мы, когда поняли, что книга, которую мы сейчас представляем, сделала бы нашу жизнь намного проще. Кроме того, мы сами бываем свидетелями подобной ситуации — многие разработчики приложений для Android соприкасались с Kotlin, и многие чувствуют, что уже достаточно разбираются, чтобы писать на нем, когда это необходимо, но они либо не знают о важности классов данных, неизменяемых свойств и структурированного параллелизма, либо не убеждены в необходимости их использования. Мы думаем, что эта книга превратит любознательного программиста в преданного последователя.

Почему мы написали эту книгу

Есть много книг, в которых продемонстрировано, как работает Android, Kotlin или параллелизм. Kotlin становится все более популярным среди разработчиков благодаря простоте внедрения и более чистому синтаксису, но этот язык предлагает гораздо больше, а именно: новые способы решения проблем параллелизма в Android. Мы написали эту книгу, чтобы подробно представить уникальную и особую взаимосвязь этих тем. И Android, и Kotlin быстро меняются. Уследить за всеми изменениями может быть непросто.

Мы рассматриваем эту книгу как важную контрольную точку в истории: в ней показано, откуда появилась система Android, на каком этапе эта операционная система находится сейчас, и как она продолжит эволюционировать по мере развития Kotlin.

Как организована эта книга

Иногда мы включаем фрагменты кода в виде скриншотов вместо обычного форматированного кода в системе Atlas. Это особенно полезно в случае с сопрограммами и потоками, поскольку можно легко идентифицировать точки приостановки. Кроме того, интегрированная среда разработки предлагает подсказки при вводе кода.

Глава 1 "Основы языка Kotlin" и *глава 2 "Фреймворк коллекций Kotlin"* посвящены основным переходам, осуществляемым с помощью Android в Kotlin. Хотя информации в этих главах достаточно, чтобы у вас сложилось четкое представление о Kotlin, в последующих главах мы подробнее рассмотрим более сложные и расширенные возможности. Пользователи, знакомые с Java или подобными синтаксическими структурами, найдут этот переход удивительно естественным.

Глава 3 "Основы Android" и *глава 4 "Параллельное программирование в Android"* познакомят вас с основами ОС Android, а также в них будут рассмотрены вопросы, касающиеся памяти и многопоточности. Как и в любой другой операционной системе, достичь параллелизма непросто.

С *главы 5 "Потокобезопасность"* по *главу 11 "Вопросы производительности и инструменты профилирования Android"* рассматриваются распространенные проблемы, связанные с памятью и многопоточностью, а также показано, как развивался фреймворк Android, чтобы предоставить разработчикам больше контроля. Все эти главы демонстрируют, как расширения Kotlin и возможности языка способны помочь разработчикам быстрее создавать более качественные приложения.

В *главе 12 "Снижение потребления ресурсов за счет оптимизации производительности"* исследуется использование мощных инструментов разработчика приложений для Android для изучения производительности и аналитики, связанной с памятью, чтобы иметь возможность увидеть то, о чем вы не знали. Эта книга предоставит инженерам профессионально разработанные и проверенные реализации наиболее распространенных задач, возникающих при нативной разработке приложений для Android. Многие задачи будут состоять из реальной проблемы, за которой следует соответствующее решение на Java и Kotlin. Когда требуется дополнительное объяснение, решения будут следовать модели быстрого сравнения и сопоставления с упором на краткость и естественный язык.

Условные обозначения и соглашения

В книге используются следующие типографские условные обозначения:

- *курсивный шрифт* обозначает новые термины, имена и расширения файлов, имена каталогов;
- **жирный шрифт** обозначает URL-адреса, адреса электронной почты, элементы интерфейса программ;
- `моноширинный шрифт` используется для листингов программ, а также внутри абзацев для ссылки на элементы программ, такие как переменные или имена функций, базы данных, типы данных, переменные среды, инструкции и ключевые слова;
- **`жирный моноширинный шрифт`** показывает команды либо другой текст, который должен быть набран пользователем, а также зарезервированные в языке программирования слова;
- *`моноширинный шрифт курсивом`* показывает текст, который должен быть заменен значениями, передаваемыми пользователем, либо значениями, определяемыми по контексту, а также комментарии в листингах программ.

Данный элемент обозначает подсказку или совет.

Данный элемент обозначает общее замечание.

Данный элемент обозначает предупреждение или предостережение.

Примеры кода

Дополнительные материалы (примеры кода, упражнения и т. д.) доступны для скачивания на странице **https://github.com/ProgrammingAndroidWithKotlin**.

Если у вас есть вопрос технического характера или возникла проблема с использованием примеров кода, отправьте электронное письмо по адресу **bookquestions@oreilly.com**.

Эта книга предназначена для того, чтобы помочь вам в вашей работе. Вообще говоря, если к этой книге прилагается пример кода, вы можете использовать его в своих программах и документации. Не нужно обращаться к нам за разрешением, если вы не воспроизводите значительную часть кода. Например, написание программы, использующей несколько фрагментов кода из этой книги, не требует разрешения, но для продажи или распространения примеров из книг O'Reilly оно понадобится. Чтобы ответить на вопрос, цитируя эту книгу и код из примера, разрешения не нужно, но в случае включения значительного количества примеров кода из этой книги в документацию по вашему продукту потребуется разрешение.

Мы приветствуем, но обычно не требуем указания авторства. Ссылка на источник цитирования обычно включает в себя название, автора, издателя и ISBN.

Если вы считаете, что использование вами примеров кода выходит за рамки добросовестного использования или специального разрешения при условиях, описанных выше, то без колебаний обращайтесь к нам по адресу электронной почты **permissions@oreilly.com**.

Благодарности

Эта книга была значительно дополнена и улучшена благодаря нашим техническим обозревателям Аднану Созуану (Adnan Sozuan) и Эндрю Гибелю (Andrew Gibel). Мы благодарим сотрудников O'Reilly за то, что они помогли нам собраться, и за всю необходимую нам поддержку, чтобы воплотить эту книгу в жизнь, в особенности Джеффа Блайела (Jeff Bleiel) и Зана МакКуэйда (Zan McQuade).

Мы благодарим Романа Елизарова и Джейка Уортона (Jake Wharton) за то, что они нашли время побеседовать с нами о направлении развития параллелизма в Kotlin и низкоуровневой оптике в Android.

Спасибо нашим друзьям, семьям и коллегам за поддержку. Мы благодарны сообществу Kotlin и людям, которые нашли время, чтобы прочитать черновики на раннем этапе и оставить свой отзыв.

Наконец, мы посвящаем эту книгу Майку Данну (Mike Dunn) — соавтору, коллеге, другу и отцу.

Мы очень по нему скучаем и надеемся, что этой книгой он будет гордиться.

ГЛАВА 1
Основы языка Kotlin

Язык Kotlin был создан компанией JetBrains из Санкт-Петербурга (Россия). Пожалуй, больше всего JetBrains известна своей интегрированной средой разработки IntelliJ Idea, являющейся основой Android Studio. Теперь Kotlin используется в самых разных окружениях в различных операционных системах. Прошло почти пять лет с тех пор, как компания Google объявила о поддержке Kotlin в Android. Согласно блогу разработчиков приложений для этой операционной системы[1] по состоянию на 2021 г. более 1,2 млн приложений из магазина Google Play написаны на Kotlin, включая 80% лучших приложений.

Если вы взялись за эту книгу, то, вероятно, уже являетесь разработчиком приложений для Android и знакомы с Java.

Kotlin был разработан для совместимости с Java. Даже его название в честь острова, расположенного недалеко от Санкт-Петербурга, — это тонкий намек на остров Ява в Индонезии. Хотя Kotlin поддерживает и другие платформы (iOS, WebAssembly, Kotlin/JS и т. д.), ключ к обширному использованию Kotlin — это поддержка виртуальной машины Java (Java virtual machine, JVM). Поскольку код на Kotlin можно компилировать в байт-код Java, он может работать везде, где работает JVM.

В большей части этой главы мы будем сравнивать Kotlin с Java. Однако важно понимать, что Kotlin — это не просто "утепленный" Java с наворотами. Kotlin — это новый язык, который связан со Scala, Swift и C# почти так же сильно, как и с Java. У него свои стили и идиомы. Хотя можно думать на Java и писать на Kotlin, мышление идиоматическими категориями раскрывает всю мощь этого языка.

Мы понимаем, что, возможно, есть разработчики, которые какое-то время работали с Kotlin, но никогда не писали на Java. Если вы один из них, то, вероятно, просто пробежитесь по этой главе и представленному в ней обзору. Однако, даже если вы неплохо разбираетесь в Kotlin, возможно, это неплохой шанс вспомнить некоторые детали.

Данная глава не предназначена для полноценного знакомства с Kotlin, поэтому, если вы совсем не знакомы с этим языком, то рекомендуем отличную книгу "Kotlin в действии"[2].

---

[1] См. **https://oreil.ly/PrfQm**.

[2] Жемеров Д., Исакова С. Kotlin в действии. — М.: ДМК-Пресс, 2018.

Данная глава представляет собой обзор основ Kotlin: система типов, переменные, функции и классы. Даже если вы не эксперт по Kotlin, этого должно быть достаточно для того, чтобы понять остальную часть книги.

Как и во всех статически типизированных языках, здесь система типов — это метаязык, используемый Kotlin с описательной целью. Поскольку это важный аспект, мы начнем с обзора данной системы.

Система типов Kotlin

Как и Java, Kotlin — это статически типизированный язык. Проверка типов выполняется во время компиляции, и в Kotlin существует механизм вывода типов[3].

С его помощью Kotlin может выявлять ошибки, которые возникают, если в коде имеются противоречия. Проверка типов позволяет компилятору перехватывать и помечать большой класс ошибок программирования. В этом разделе освещаются некоторые наиболее интересные особенности системы типов Kotlin, в том числе тип `Unit`, функциональные типы, null-безопасность и обобщенные типы.

Примитивные типы

Наиболее очевидная разница между системами типов Java и Kotlin заключается в том, что в Kotlin не существует понятия *"примитивный тип"*.

В Java есть типы `int`, `float`, `boolean` и т. д. Они отличаются тем, что не наследуют от типа `Object`. Например, утверждение `int n = null;` не является допустимым в Java. Равно как и `List<int> integers;`. Чтобы сгладить это несоответствие, у каждого примитивного типа Java есть эквивалент — *класс-обертка* (boxed type). `Integer`, например, является аналогом `int`, `Boolean` — это аналог `boolean` и т. д. Различие между примитивными типами и классами-обертками почти исчезло, потому что, начиная с Java 5, компилятор Java автоматически выполняет преобразование между ними. Теперь утверждение `Integer i = 1` является допустимым.

В Kotlin нет примитивных типов, которые бы загромождали его систему типов. Суперкласс `Any`, аналог суперкласса `Object` в Java, лежит в основе иерархии типов Kotlin.

Внутреннее представление простых типов в Kotlin не связано с системой типов этого языка. У компилятора Kotlin имеется достаточно информации для представления, например, 32-битного целого числа с такой же эффективностью, как и в любом другом языке. Таким образом, если написать `val i: Int = 1`, это может привести к использованию примитивного типа или класса-обертки, в зависимости от того, как переменная i применяется в коде. По возможности компилятор Kotlin будет использовать примитивные типы.

---

[3] Официально в Kotlin это называется *автоматическим определением типов*, при котором используется частичная фаза компилятора (компонент внешнего интерфейса) для проверки типов в написанном вами коде, пока вы работаете в интегрированной среде разработки. Это плагин для IntelliJ! Забавный факт: IntelliJ и Kotlin полностью состоят из плагинов компилятора.

Null-безопасность

Второе существенное различие между Java и Kotlin заключается в том, что *допустимость значений null* является частью системы типов Kotlin. Тип с поддержкой значения null отличается от своего аналога, не поддерживающего это значение, знаком вопроса в конце имени: например, String и String?, Person и Person?. Компилятор Kotlin позволяет присвоить значение null типу с поддержкой значения Null: var name: String? = null. Однако в случае аналога это недопустимо: var name: String = null (поскольку String не является типом, допускающим значение null).

Все классы в Kotlin наследуют общий суперкласс, известный как Any. В Java этот суперкласс называется Object. Однако есть существенная разница: Any — это родительский класс для всех классов, не поддерживающих значения null, а Any? — это родительский класс для всех классов с поддержкой этого значения. В этом и состоит основа *null-безопасности*. Другими словами, может быть полезно рассматривать систему типов Kotlin как два идентичных дерева типов: все типы, не поддерживающие значения null, — это подтипы Any, а все нулевые типы с поддержкой null — это подтипы Any?.

Переменные должны быть инициализированы. Для переменной нет значения по умолчанию. Например, следующий код вызовет ошибку компилятора:

```
val name: String  // Ошибка! Типы, не поддерживающие значения null,
                  // должны быть инициализированы!
```

Как было сказано ранее, в Kotlin существует механизм вывода типов. Часто компилятор может определить тип идентификатора, располагая уже имеющейся у него информацией. Данный процесс называется *автоматическим определением типа*. Если компилятор сам может определить тип, то разработчику не нужно его указывать. Например, если присвоить переменной name значение "Jerry", это вполне допустимо, несмотря на то что тип переменной name не указан. Компилятор сам может определить, что переменная name должна быть строкой, поскольку ей присвоено значение "Jerry" (типа String).

Однако типы, определяемые автоматически, могут и удивить. Следующий код вызовет ошибку компилятора:

```
var name = "Jerry"
name = null
```

Компилятор автоматически определил тип String для переменной name, а не тип String?. Поскольку String не является типом, допускающим значение null, попытка присвоить ему это значение недопустима.

Важно отметить, что тип, *допускающий значение null*, и его аналог, *который не поддерживает это значение*, — это не одно и то же. Вполне логично, что тип, допускающий значение null, ведет себя как супертип связанного с ним типа, не допускающего значение null. Следующий код, например, компилируется без проблем, потому что String — это String?:

```
val name = Jerry
fun showNameLength(name: String?) { // Функция принимает параметр,
                                    // допускающий значение null
```

```
    // ...
}

showNameLength(name)
```

С другой стороны, этот код вообще не будет компилироваться, потому что String? — это *не* тип String:

```
val name: String? = null
fun showNameLength(name: String) { // Эта функция принимает только значения,
                                   // не являющиеся null
    println(name.length)
}

showNameLength(name) // Ошибка! Не компилируется, потому что "name"
                     // может быть значением null
```

Простое изменение типа параметра не решит проблему полностью:

```
val name: String? = null
fun showNameLength(name: String?) { // Эта функция принимает значения null
    println(name.length) // Ошибка!
}

showNameLength(name) // Компиляция
```

Этот фрагмент кода завершается ошибкой

```
Only safe (?.) or non-null asserted (!!.) calls are allowed on a nullable receiver of
type String?.
```

Kotlin требует, чтобы переменные, допускающие значение null, обрабатывались безопасно — таким образом, чтобы они не могли генерировать исключение пустого указателя. Чтобы код компилировался, он должен корректно обрабатывать случай, когда переменная name равна null:

```
val name: String? = null
fun showNameLength(name: String?) {
    println(if (name == null) 0 else name.length)
    // вскоре мы будем использовать еще более приятный синтаксис
}
```

В Kotlin есть специальные операторы ?. и ?:, которые упрощают работу с сущностями, допускающими значение null:

```
val name: String? = null
fun showNameLength(name: String?) {
    println(name?.length ?: 0)
}
```

В предыдущем примере, когда переменная name не равна null, значения name?.length и name.length одинаковы. Однако, когда переменная name равна null, значение name?.length тоже равно null. Выражение не выбрасывает исключение пустого указателя. Таким образом, первый оператор из предыдущего примера, ?., синтаксически эквивалентен:

```
if (name == null) null else name.length
```

Второй оператор, *оператор Элвиса* ?:, возвращает левое выражение, если оно не равно нулю, или правое выражение в противном случае. Обратите внимание, что выражение в правой части вычисляется, только если левое выражение равно null.

Это эквивалентно:

```
if (name?.length == null) 0 else name.length
```

Тип *Unit*

В Kotlin у *всего* есть значение. Всегда. Как только вы это поймете, нетрудно будет представить, что даже метод, который ничего специально не возвращает, имеет значение по умолчанию. Это значение называется Unit. Unit — это имя ровно одного объекта, значение, которое присваивается, если никакого иного значения нет. Для удобства тип объекта Unit называется Unit.

Вся концепция Unit может показаться странной разработчикам Java, которые привыкли различать выражения (то, что имеет значение) и операторы (у которых его нет).

Условное выражение в Java — отличный пример различия между *оператором* и *выражением*, потому что в нем есть и то и другое! В Java можно написать:

```
if (maybe) doThis() else doThat();
```

Однако нельзя написать следующее:

```
int n = if (maybe) doThis() else doThat();
```

Такие операторы, как if, не возвращают значения. Нельзя присвоить значение оператора if переменной, потому что эти операторы ничего не возвращают. То же относится и к операторам цикла, операторам case и т. д.

Однако у оператора if в Java есть аналог — *троичное выражение*. Поскольку это выражение, оно возвращает значение, которое можно присвоить. Следующий код является допустимым в Java (при условии, что doThis и doThat возвращают целые числа):

```
int n = (maybe) ? doThis() : doThat();
```

В Kotlin нет необходимости в двух условных выражениях, потому что, if — это выражение, возвращающее значение. Например, следующий код абсолютно допустимый:

```
val n = if (maybe) doThis() else doThat()
```

В Java метод со словом void в качестве типа возвращаемого значения подобен оператору. На самом деле, это немного неправильное название, потому что void не яв-

ляется типом. Это зарезервированное слово в языке Java, указывающее на то, что метод не возвращает значение. Когда в Java появились обобщенные типы, был введен тип Void, чтобы заполнить эту пустоту (преднамеренно!). Однако два представления понятия "ничего", ключевое слово и тип, сбивают с толку и несовместимы: функция, чей тип возвращаемого значения Void, должна явно возвращать null.

Kotlin гораздо более последователен: все функции возвращают значение, и у них есть тип. Если код функции не возвращает значение явно, то функция имеет значение типа Unit.

Функциональные типы

Система типов Kotlin поддерживает *функциональные типы*. Например, следующий код определяет переменную func, значение которой является функцией. Это лямбда-выражение { x -> x.pow(2.0) }:

```
val func: (Double) -> Double = { x -> x.pow(2.0) }
```

Поскольку func — это функция, которая принимает один аргумент типа Double и возвращает значение типа Double, это тип (Double) -> Double.

В предыдущем примере мы явно указали тип func. Тем не менее компилятор Kotlin может многое узнать о типе переменной func из присвоенного ей значения. Он знает тип возвращаемого значения, потому что ему известен тип pow. Однако у него недостаточно информации, чтобы определить тип параметра x. Но если ее предоставить, то спецификатор типа для переменной можно опустить:

```
val func = { x: Double -> x.pow(2.0)}
```

Система типов Java не может описывать функциональный тип — здесь нельзя говорить о функциях вне контекста классов, содержащих их. Чтобы сделать нечто похожее в Java, можно было бы использовать функциональный тип Function:

```
Function<Double, Double> func
    = x -> Math.pow(x, 2.0);

func.apply(256.0);
```

Переменной func присвоен анонимный экземпляр типа Function, чей метод apply — это заданное лямбда-выражение.

Благодаря функциональным типам функции могут получать другие функции в качестве параметров или возвращать их в виде значений. Мы называем их *функциями высшего порядка*. Рассмотрим шаблон для типа Kotlin: (A, B) -> C. Он описывает функцию, которая принимает два параметра — типа A и типа B (независимо от того, какие это могут быть типы) — и возвращает значение типа C. Поскольку язык типов Kotlin может описывать функции, A, B и C сами по себе могут быть функциями.

Если это сильно смахивает на метапрограммирование, то лишь потому, что так оно и есть. Внесем больше конкретики. Заменим A из шаблона на (Double, Double) -> Int.

Это функция, которая принимает два параметра Double и возвращает Int. Вместо B просто напишем Double. Теперь у нас есть ((Double, Double) -> Int, Double) -> C.

Наконец, предположим, что наш новый функциональный тип возвращает (Double) -> Int — функцию, которая принимает один параметр Double и возвращает Int. В следующем коде показана полная сигнатура этой гипотетической функции:

```
fun getCurve(
    surface: (Double, Double) -> Int,
    x: Double
): (Double) -> Int {
    return { y -> surface(x, y) }
}
```

Мы только что описали функциональный тип, который принимает два аргумента. Первый аргумент — это функция (surface) с двумя параметрами, оба типа Double, которые возвращают значение типа Int. Второй — Double (x). Наша функция getCurve возвращает функцию, которая принимает один параметр Double (y) и возвращает значение типа Int.

Возможность передавать функции в качестве аргументов в другие функции является основой функциональных языков. Используя функции высшего порядка, можно уменьшить избыточность кода, не создавая при этом новые классы, как в Java (создание подклассов интерфейсов Runnable или Function). При разумном использовании функции высшего порядка улучшают удобочитаемость кода.

Обобщенные типы

Как и в Java, система типов Kotlin поддерживает переменные типов. Например:

```
fun <T> simplePair(x: T, y: T) = Pair(x, y)
```

Данная функция создает объект Pair, в котором оба элемента должны быть одного и того же типа. Учитывая это определение, отметим, что simplePair("Hello", "Goodbye") и simple Pair(4, 5) являются допустимыми, а simplePair("Hello", 5) — нет.

Обобщенный тип, обозначенный как T в определении simplePair, — это переменная типа: значения, которые он может принимать, являются типами Kotlin (в данном примере String или Int). Функция (или класс), использующая переменную типа, называется *обобщенной*.

Переменные и функции

Теперь, когда у нас есть язык типов, обеспечивающий нам поддержку, можно приступить к обсуждению синтаксиса самого Kotlin.

В Java синтаксической сущностью верхнего уровня является класс. Все переменные и методы это члены того или иного класса, а класс представляет собой основной элемент в одноименном файле.

В Kotlin таких ограничений нет. При желании можно поместить всю программу в один файл (пожалуйста, не нужно этого делать), а также можно определять переменные и функции за пределами любого класса.

Переменные

Есть два способа объявить переменную: с помощью ключевых слов `val` и `var`. Ключевое слово является обязательным и идет первым:

```
val ronDeeLay = "the night time"
```

Ключевое слово `val` создает переменную только для чтения: ее нельзя переназначить. Хотя будьте осторожны! Вы можете подумать, что `val`-переменная похожа на переменную Java, объявленную с помощью ключевого слова `final`. Сходство есть, но это не одно и то же! Хотя переменную с ключевым словом `val` нельзя переназначить, она определенно может изменить значение! Переменная `val` в Kotlin больше похожа на поле класса Java, у которого есть метод чтения, но нет метода записи, как показано в следующем коде:

```
val surprising: Double
get() = Math.random()
```

Каждый раз при обращении к переменной `surprising` будет возвращаться другое случайное значение. Это пример свойства без *поля, где хранится его значение*. Мы рассмотрим свойства позже в этой главе. С другой стороны, если бы мы написали `val rand = Random()`, то значение `rand` не изменилось бы, и это было бы больше похоже на переменную с ключевым словом `final` в Java.

Второе ключевое слово, `var`, создает знакомую изменяемую переменную: это как коробочка, в которой содержится последнее, что вы в нее положили.

В следующем разделе мы рассмотрим одну из особенностей Kotlin как функционального языка: *лямбда-выражения*.

Лямбда-выражения

Kotlin поддерживает функциональные литералы: лямбда-выражения. В Koltin лямбда-выражения всегда окружены фигурными скобками. В этих скобках список аргументов находится слева от стрелки `->`, а выражение, представляющее собой значение выполнения лямбда-выражения, располагается справа, как показано в следующем коде:

```
{ x: Int, y: Int -> x * y }
```

По соглашению возвращаемое значение — это значение последнего выражения в теле лямбда-выражения. Например, функция, показанная в следующем коде, имеет тип `(Int, Int) -> String`:

```
{ x: Int, y: Int -> x * y; "down on the corner" }
```

В Kotlin есть очень интересная особенность, позволяющая расширить возможности языка. Если последним аргументом функции является другая функция (функция

высшего порядка), то можно вынести лямбда-выражение, переданное в качестве параметра, за скобки, которые обычно ограничивают фактический список параметров, как показано в следующем коде:

```
// Последний аргумент, "callback ", - это функция
fun apiCall(param: Int, callback:() -> Unit)
```

Данная функция обычно используется так:

```
apiCall(1, { println("I'm called back!")})
```

Но благодаря упомянутой языковой особенности ее также можно использовать следующим образом:

```
apiCall(1) {
    println("I'm called back!")
}
```

Выглядит гораздо приятнее, не так ли? Благодаря этой функции ваш код может быть более удобочитаемым. Более продвинутое использование этой функции — это DSL[4].

Функции-расширения

Что вы делаете, когда вам нужно добавить новый метод в существующий класс, а этот класс из зависимости, исходный код которой вам не принадлежит?

В Java, если у класса нет ключевого слова `final`, можно создать его подкласс. Иногда это неидеальный вариант, потому что приходится управлять еще одним типом, что усложняет проект. При наличии ключевого слова `final` можно определить статический метод внутри некоего собственного служебного класса, как показано в следующем коде:

```
class FileUtils {
    public static String getWordAtIndex(File file, int index) {
        /* Реализация скрыта для краткости */
    }
}
```

В предыдущем примере мы определили функцию для получения слова в текстовом файле по заданному индексу. В месте использования нужно написать `String word = getWordAtIndex(file, 3)`, предполагая, что мы осуществляем статический импорт `FileUtils.getWordAtIndex`. Это нормально, мы уже много лет делаем это в Java, и все работает.

В Kotlin можно делать еще кое-что. Здесь есть возможность определить новый метод класса, даже если он не является реальной функцией-членом этого класса. Таким образом, на самом деле вы не расширяете класс, но в точке использования соз-

[4] DSL (domain specific language) означает *предметно-ориентированный язык*. Пример этого языка в Kotlin — Gradle.

дается впечатление, будто вы добавили метод в класс. Как такое возможно? Путем определения *функции-расширения*, как показано в следующем коде:

```
// объявлено в FileUtils.kt
fun File.getWordAtIndex(index: Int): String {
    val context = this.readText() // слово 'this' соответствует файлу
    return context.split(' ').getOrElse(index) { "" }
}
```

В объявлении функции-расширения слово `this` обозначает получение экземпляра типа (в нашем случае — это `File`). У нас есть доступ только к общедоступным и внутренним атрибутам и методам, поэтому поля `private` и `protected` недоступны — скоро вы поймете почему.

В точке использования нужно написать `val word = file.getWordAtIndex(3)`. Как видите, мы вызываем функцию `getWordAtIndex()` для экземпляра `File`, как если бы у класса `File` была функция-член `getWordAtIndex()`. Это делает точку использования более выразительной и удобной для чтения. Нам не пришлось придумывать имя нового служебного класса: мы можем объявлять функции-расширения непосредственно в корне исходного файла.

Посмотрим на декомпилированную версию `getWordAtIndex`:

```
public class FileUtilsKt {
    public static String getWordAtIndex(
            File file, int index
    ) {
        /* Реализация скрыта для краткости */
    }
}
```

При компиляции сгенерированный байт-код нашей функции-расширения представляет собой эквивалент статического метода, который принимает `File` в качестве первого аргумента. Вмещающий класс `FileUtilsKt` назван в честь исходного файла (*FileUtils.kt*) с суффиксом Kt.

Это объясняет, почему нельзя получить доступ к приватным полям в функции-расширении: мы просто добавляем статический метод, который принимает тип-получатель в качестве параметра.

И это не все! Для атрибутов класса можно объявить *свойства-расширения*. Идея та же — на самом деле вы не расширяете класс, но можете создавать новые атрибуты, доступные с помощью точечной нотации, как показано в следующем коде:

```
// У класса Rectangle есть ширина и высота.
val Rectangle.area: Double
    get() = width * height
```

Обратите внимание, что на сей раз для объявления свойства-расширения мы использовали ключевое слово `val` (вместо `fun`). Выглядит это так: `val area = rectangle.area`.

Функции-расширения и свойства-расширения позволяют расширять возможности классов, используя точечную нотацию, сохраняя при этом разделение ответственности. Вы не загромождаете существующие классы конкретным кодом под конкретные нужды.

Классы

Поначалу кажется, что классы в Kotlin выглядят так же, как и в Java: сначала идет ключевое слово `class`, за которым следует блок, определяющий класс. Тем не менее одна из ключевых особенностей Kotlin — это синтаксис конструктора и возможность объявлять в нем свойства. В следующем коде показано определение простого класса `Point` наряду с вариантами его использования:

```
class Point(val x: Int, var y: Int? = 3)

fun demo() {
    val pt1 = Point(4)
    assertEquals(3, pt1.y)
    pt1.y = 7
    val pt2 = Point(7, 7)
    assertEquals(pt2.y, pt1.y)
}
```

Инициализация класса

Обратите внимание, что в предыдущем коде конструктор `Point` встроен в объявление класса. Он называется *главным конструктором*. Главный конструктор `Point` объявляет два свойства класса, `x` и `y`, оба из которых являются целыми числами. Первое свойство, `x`, доступно только для чтения. Второе, `y`, является изменяемым и поддерживает значения `null`. По умолчанию оно имеет значение 3.

Обратите внимание на существенную важность ключевых слов `var` и `val`! Объявление `class Point(x: Int, y: Int)` сильно отличается от предыдущего объявления, потому что здесь не объявляется никаких свойств-членов. Без ключевых слов идентификаторы `x` и `y` — это просто аргументы конструктора. Например, следующий код сгенерирует ошибку:

```
class Point(x: Int, y: Int?)

fun demo() {
    val pt = Point(4)
    pt.y = 7 // Ошибка! Ожидается переменная
}
```

У класса Point из этого примера только один конструктор, определенный в его объявлении. Однако классы не ограничиваются только им. В Kotlin также можно определять и вторичные конструкторы, и блоки инициализации, как показано в следующем определении класса Segment:

```
class Segment(val start: Point, val end: Point) {
    val length: Double = sqrt(
            (end.x - start.x).toDouble().pow(2.0)
                    + (end.y - start.y).toDouble().pow(2.0))
    init {
        println("Point starting at $start with length $length")
    }

    constructor(x1: Int, y1: Int, x2: Int, y2: Int) :
            this(Point(x1, y1), Point(x2, y2)) {
        println("Secondary constructor")
    }
}
```

В этом примере есть и другие интересные вещи. Прежде всего, обратите внимание, что вторичный конструктор должен делегировать работу главному конструктору, : this(...), в объявлении. У конструктора может быть блок кода, но сначала требуется явно делегировать выполнение операции главному конструктору.

Возможно, больший интерес представляет порядок выполнения кода в предыдущем объявлении. Предположим, нужно создать новый экземпляр Segment с помощью вторичного конструктора.

В каком порядке будет появляться вывод?

Что же, попробуем и посмотрим:

```
>>> val s = Segment(1, 2, 3, 4)

Point starting at Point(x=1, y=2) with length 2.8284271247461903
Secondary constructor
```

Довольно интересно. Блок init запускается до блока кода, связанного со вторичным конструктором! С другой стороны, свойства length и start были инициализированы со значениями, предоставленными конструктором. Это означает, что главный конструктор должен быть запущен еще до блока init.

На самом деле Kotlin гарантирует следующий порядок: главный конструктор (если он есть) запускается первым. После его завершения блоки init запускаются в порядке объявления (сверху вниз). Если новый экземпляр создается с использованием вторичного конструктора, то блок кода, связанный с этим конструктором, выполняется последним.

Свойства

Переменные Kotlin, объявляемые с помощью ключевых слов `val` или `var` в конструкторе или на верхнем уровне класса, на самом деле определяют *свойство*. Свойство в Kotlin напоминает сочетание поля Java и его метода чтения (если свойство доступно только для чтения и определено с помощью ключевого слова `val`) или его методов чтения и записи (если оно определено с помощью ключевого слова `var`).

Kotlin поддерживает настройку методов доступа и записи для свойства. Для этого у него есть специальный синтаксис, как показано в определении класса `Rectangle`:

```
class Rectangle(val l: Int, val w: Int) {
    val area: Int
        get() = l * w
}
```

Свойство `area` является *синтетическим*: оно вычисляется из значений длины и ширины. Поскольку не имеет смысла присваивать этому свойству значение, здесь используется ключевое слово `val` с доступом только для чтения, а метод `set()` отсутствует.

Используйте стандартную "точечную" нотацию для доступа к значению свойства:

```
val rect = Rectangle(3, 4)
assertEquals(12, rect.area)
```

Для дальнейшего изучения собственных методов чтения и записи свойств рассмотрим класс, который имеет часто используемый хеш-код (возможно, экземпляры хранятся в `Map`) и довольно затратен для расчетов. Мы решили кэшировать хеш-код и присвоить ему значение при изменении значения свойства класса. Первая попытка может выглядеть примерно так:

```
// Этот код не работает (мы увидим, почему)
class ExpensiveToHash(_summary: String) {

    var summary: String = _summary
        set(value) {
            summary = value // неограниченная рекурсия!!
            hashCode = computeHash()
        }

     // здесь идут другие объявления...
     var hashCode: Long = computeHash()

     private fun computeHash(): Long = ...
}
```

Предыдущий код завершится ошибкой из-за неограниченной рекурсии: присваивание значения `summary` — это вызов функции `summary.set()`! Попытка задать значение свойства внутри его собственного метода записи не сработает. Для решения этой проблемы Kotlin использует специальный идентификатор `field`.

Ниже показана исправленная версия кода:

```
class ExpensiveToHash(_summary: String) {

    var summary: String = _summary
        set(value) {
            field = value
            hashCode = computeHash()
        }

    // здесь идут другие объявления...
    var hashCode: Long = computeHash()

    private fun computeHash(): Long = ...
}
```

Идентификатор `field` имеет особое значение только в собственных методах чтения и записи, где он обозначает *поле, содержащее состояние свойства*.

Также обратите внимание на то, что в предыдущем коде демонстрируется идиома для инициализации свойства, у которого есть собственный метод чтения или записи со значением, предоставляемым конструктору класса. Определение свойств в списке параметров конструктора действительно удобно. Однако, если несколько определений свойств в конструкторе имеют собственные методы чтения и записи, это может сильно затруднить чтение конструктора.

Когда свойство с собственным методом чтения и записи нужно инициализировать из конструктора, свойство определяется вместе с этими методами в теле класса. Свойство инициализируется параметром из конструктора (в данном случае `_summary`), что снова иллюстрирует важность ключевых слов `val` и `var` в списке параметров конструктора. `_summary` — это просто параметр, а не свойство класса, потому что он объявляется без ключевого слова.

Модификатор *lateinit*

Бывают случаи, когда значение переменной недоступно в месте ее объявления. Очевидный пример такой ситуации для разработчиков приложений для Android — это UI-виджет, используемый в компоненте `Activity` или `Fragment`. Переменную, используемую на протяжении всей активности для обозначения виджета, можно инициализировать только при запуске метода `onCreate` или `onCreateView`. Например, переменная `button`:

```
class MyFragment: Fragment() {
    private var button: Button? = null // фактическое значение
                                       // будет предоставлено позже
}
```

Переменная должна быть инициализирована. Стандартный способ, поскольку мы пока не знаем значения, состоит в следующем: нужно сделать так, чтобы переменная поддерживала значения null, и инициализировать ее, используя данное значение.

Первый вопрос, который нужно задать себе в такой ситуации: действительно ли необходимо определять эту переменную в данный момент и в данном месте? Будет ли ссылка на переменную button использоваться в нескольких методах или в действительности она используется только в одном или двух конкретных местах? В последнем случае глобальную переменную класса можно полностью исключить.

Однако проблема с использованием типа, допускающего значение null, заключается в том, что всякий раз, когда вы будете использовать в коде переменную button, вам придется проверять ее на допустимость значений null. Например: `button?.setOnClickListener { .. }`. Пара таких переменных, и вы получите большое количество надоедливых вопросительных знаков! Этот код может выглядеть еще более громоздко, особенно если вы привыкли к Java и его простой точечной нотации.

Вы можете спросить: почему Kotlin не позволяет объявить переменную button с использованием типа, не поддерживающего значения null, если вы уверены, что инициализируете ее до того, как что-либо попытается получить к ней доступ? Нет ли способа смягчить правило инициализации компилятора только для этой переменной?

Такое возможно. Это можно сделать с помощью модификатора lateinit, как показано в следующем коде:

```
class MyFragment: Fragment() {
    private lateinit var button: Button // будет инициализирована позднее
}
```

Поскольку переменная объявлена с использованием модификатора lateinit, Kotlin позволит вам объявить ее, не присваивая ей значение. Переменная должна быть изменяемой, var, потому что по определению вы присвоите ей значение позже. Отлично — проблема решена, верно?

Мы, авторы книги, именно так и думали, когда начинали работать с Kotlin. Теперь мы склоняемся к использованию модификатора lateinit только в случае крайней необходимости и вместо этого используем значения, допускающие значение null. Почему?

Используя модификатор lateinit, вы сообщаете компилятору: "У меня нет значения, которое я мог бы дать тебе прямо сейчас. Но я дам тебе его позже, обещаю". Если бы компилятор Kotlin мог говорить, он бы ответил: "Отлично! Ты говоришь, что знаешь, что делаешь. Если что-то пойдет не так, отвечать тебе". Используя модификатор lateinit, вы отключаете защиту от пустых ссылок для своей переменной. Если вы забыли инициализировать переменную или попытались вызвать для нее какой-либо метод, прежде чем он будет инициализирован, то получите исключение UninitializedPropertyAccessException. По сути это то же самое, что и исключение NullPointerException в Java.

Каждый раз при использовании модификатора `lateinit` мы в конечном счете обжигались. Наш код может работать во всех случаях, которые мы предвидели. Мы были уверены, что ничего не упустили... и ошиблись.

Объявляя переменную с использованием модификатора `lateinit`, вы делаете предположения, которые компилятор не может доказать. Когда вы или другие разработчики впоследствии будете выполнять рефакторинг кода, ваша тщательно продуманная архитектура может быть нарушена. Тесты могут перехватить ошибку. Или нет[5]. По нашему опыту использование модификатора `lateinit` всегда приводило к сбоям во время выполнения. Как мы это исправляли? Использовали тип, допускающий значение `null`.

Если вы будете использовать тип, допускающий значение `null`, вместо модификатора `lateinit`, компилятор Kotlin заставит вас проверять код на допустимость значения `null` именно в тех местах, где он может быть таковым. Добавление нескольких вопросительных знаков определенно стоит того, чтобы получить более надежный код.

Свойства с отложенной инициализацией

В программной инженерии принято откладывать создание и инициализацию объекта до тех пор, пока он действительно не понадобится. Данный паттерн известен как *отложенная инициализация*.

Он особенно распространен в Android, поскольку выделение памяти для большого количества объектов во время запуска приложения может привести к увеличению времени запуска. Пример 1.1 — типичный случай отложенной инициализации в Java.

Пример 1.1. Отложенная инициализация в Java

```
class Lightweight {
    private Heavyweight heavy;

    public Heavyweight getHeavy() {
        if (heavy == null) {
            heavy = new Heavyweight();
        }
        return heavy;
    }
}
```

[5] Можно проверить, инициализировано ли свойство `button`, используя следующий код: `this::button.isInitialized`. Можно полагаться на то, что разработчики добавят эту проверку во все нужные места, но это не решит основной проблемы.

Поле heavy инициализируется с новым экземпляром класса Heavyweight (т. е. предположительно, его затратно создавать) только тогда, когда его значение сначала запрашивается, например вызовом lightweight.getHeavy(). Последующие вызовы метода getHeavy() вернут кэшированный экземпляр.

В Kotlin отложенная инициализация является частью языка. При использовании директивы by lazy и предоставлении блока инициализации остальная часть не является явной, как показано в примере 1.2.

Пример 1.2. Отложенная инициализация в Kotlin

```
class Lightweight {
    val heavy by lazy { // Блок инициализации
        Heavyweight()
    }
}
```

Мы объясним этот синтаксис более подробно в следующем разделе.

Обратите внимание, что код из примера 1.1 не является потокобезопасным. Несколько потоков, вызывающих метод getHeavy() класса Lightweight, одновременно могут привести к появлению разных экземпляров Heavyweight.

По умолчанию код из примера 1.2 является потокобезопасным. Вызовы Lightweight::getHeavy() будут синхронизированы, поэтому в блоке инициализации будет находиться только один поток за раз.

Детально контролировать одновременный доступ к блоку с отложенной инициализацией можно с помощью класса LazyThreadSafetyMode.

Значение с отложенной инициализацией в Kotlin не будет инициализировано до тех пор, пока во время выполнения не будет выполнен вызов. При первой ссылке на свойство heavy будет запущен блок инициализации.

Делегаты

Свойства с отложенной инициализацией — это пример более универсальной особенности Kotlin, которая называется *делегированием*. В объявлении используется ключевое слово by для определения делегата, который отвечает за чтение и запись значения свойства. В Java можно было бы сделать что-то подобное, например, с методом записи, который передавал бы свой аргумент в качестве параметра для вызова метода в каком-то другом объекте, делегате.

Поскольку отложенная инициализация — отличный пример мощи идиоматического Kotlin, выделим минутку, чтобы прояснить этот вопрос.

В первой части объявления из примера 1.2 написано val heavy. Это, как мы знаем, объявление переменной только для чтения heavy. Далее следует ключевое слово by,

которое вводит делегат. Ключевое слово by говорит о том, что следующий идентификатор в объявлении — это выражение, которое будет возвращать объект, отвечающий за значение heavy.

Далее идет идентификатор lazy. Kotlin ожидает выражение. Оказывается, lazy — это просто функция! Она принимает один аргумент, лямбда-выражение, и возвращает объект. Возвращаемый объект — это Lazy<T>, где T — это тип, возвращаемый лямбда-выражением.

Реализация Lazy<T> довольно проста: при первом вызове он запускает лямбда-выражение и кэширует его значение. При последующих вызовах он возвращает кэшированное значение.

Это лишь одна из многих разновидностей делегирования свойств. Используя ключевое слово by, также можно определять *наблюдаемые свойства* (см. документацию Kotlin[6], где приводится информация по делегированию свойств). Тем не менее данный вид делегирования является наиболее распространенным вариантом делегирования свойств, используемым в Android.

Объекты-компаньоны

Возможно, вам интересно, что Kotlin сделал со статическими переменными. Не бойтесь; Kotlin использует *объекты-компаньоны*. Объект-компаньон — это *объект-одиночка*, всегда связанный с классом Kotlin. Хотя это и не обязательно, чаще всего определение объекта-компаньона размещается в нижней части связанного с ним класса, как показано здесь:

```
class TimeExtensions {
    // другой код

    companion object {
        const val TAG = "TIME_EXTENSIONS"
    }
}
```

Объекты-компаньоны могут иметь имена, расширять классы и наследовать от интерфейсов. В данном примере объект-компаньон TimeExtension называется StdTimeExtension и наследует от интерфейса Formatter:

```
interface Formatter {
    val yearMonthDate: String
}

class TimeExtensions {
    // другой код

    companion object StdTimeExtension : Formatter {
```

[6] См. **https://oreil.ly/6lTab**.

```
        const val TAG = "TIME_EXTENSIONS"
        override val yearMonthDate = "yyyy-MM-d"
    }
}
```

При ссылке на член объекта-компаньона из класса, который содержит его, нужно указать ссылку с именем вмещающего класса:

```
val timeExtensionsTag = TimeExtensions.StdTimeExtension.TAG
```

Объект-компаньон инициализируется, когда Kotlin загружает связанный с ним класс.

Классы данных

Существует категория классов, настолько распространенных, что в Java для них есть имя: они называются *POJO (Plain Old Java Object)*, или старые добрые Java-объекты. Это простые представления структурированных данных. Они представляют собой набор членов данных (полей), у большинства из которых есть методы чтения и записи, а также несколько других методов: `equals`, `hashCode` и `toString`. Такого рода классы настолько распространены, что Kotlin сделал их частью языка. Они называются *классами данных*.

Можно улучшить определение класса `Point`, сделав его классом данных:

```
data class Point(var x: Int, var y: Int? = 3)
```

В чем разница между этим классом, объявленным с использованием модификатора `data`, и исходным классом, объявленным без него? Попробуем провести простой эксперимент, для начала используя исходное определение `Point` (без модификатора `data`):

```
class Point(var x: Int, var y: Int? = 3)

fun main() {
    val p1 = Point(1)
    val p2 = Point(1)
    println("Points are equals: ${p1 == p2}")
}
```

Результатом выполнения этой небольшой программы будет фраза `"Points are equals: false"`. Причина этого, возможно, неожиданного результата заключается в том, что Kotlin компилирует `p1 == p2` как `p1.equals(p2)`. Поскольку первое определение класса `Point` не переопределяло метод `equals`, это превращается в вызов метода `equals` в классе `Any`, являющемся родительским по отношению к классу `Point`. Реализация `equals` в `Any` возвращает `true` только тогда, когда объект сравнивается сам с собой.

Если мы попробуем сделать то же самое с новым определением `Point` как класса данных, то программа выведет фразу `"Points are equals: true"`. Новое определение ведет себя, как и предполагалось, потому что класс данных автоматически включа-

ет переопределения для методов equals, hashCode и toString. Каждый из этих автоматически сгенерированных методов зависит от всех свойств класса.

Например, версия Point как класса данных содержит метод equals, эквивалентный следующему коду:

```
override fun equals(o: Any?): Boolean {
    // Если это не Point, возвращаем false.
    // Обратите внимание, что null не является Point.
    if (o !is Point) return false

    // Если это Point, то x и y должны быть одинаковыми.
    return x == o.x && y == o.y
}
```

Помимо стандартных реализаций методов equals и hashCode, класс данных также предоставляет метод copy. Вот пример его использования:

```
data class Point(var x: Int, var y: Int? = 3)
val p = Point(1)            // x = 1, y = 3
val copy = p.copy(y = 2)    // x = 1, y = 2
```

Классы данных Kotlin идеально подходят для часто используемой идиомы.

В следующем разделе мы рассмотрим еще один особый вид классов: *классы перечислений*.

Классы перечислений

Помните, когда-то разработчикам говорили, что перечисления слишком затратны для Android? К счастью, теперь так никто даже и не думает: используйте их, как душе угодно!

Классы перечислений в Kotlin очень похожи на перечисления в Java. Они создают класс, который не может быть подклассом и имеет фиксированный набор экземпляров. Как и в Java, перечисления не могут быть подклассами других типов, но могут реализовывать интерфейсы и иметь конструкторы, свойства и методы. Вот пара простых примеров:

```
enum class GymActivity {
    BARRE, PILATES, YOGA, FLOOR, SPIN, WEIGHTS
}

enum class LENGTH(val value: Int) {
    TEN(10), TWENTY(20), THIRTY(30), SIXTY(60);
}
```

Перечисления очень хорошо работают с оператором when. Например:

```
fun requiresEquipment(activity: GymActivity) = when (activity) {
    GymActivity.BARRE -> true
```

```
    GymActivity.PILATES -> true
    GymActivity.YOGA -> false
    GymActivity.FLOOR -> false
    GymActivity.SPIN -> true
    GymActivity.WEIGHTS -> true
}
```

Когда этот оператор используется для присвоения переменной или в качестве тела выражения функции, как в предыдущем примере, он должен быть *исчерпывающим*. Исчерпывающий оператор when охватывает все возможные значения своего аргумента (в данном случае activity). Стандартный способ гарантировать, что он является исчерпывающим, — включить сюда ветку else. else соответствует любому значению аргумента, которое не указано явно в списке.

В предыдущем примере, чтобы быть исчерпывающим, оператор when должен содержать все возможные значения параметра функции, activity. Параметр имеет тип GymActivity и, следовательно, должен быть одним из экземпляров перечисления. Поскольку перечисление имеет известный набор экземпляров, Kotlin может определить, что все возможные значения охвачены как явно перечисленные и позволяют опустить ветку else.

Если ее опустить, мы приобретаем очень хорошее преимущество: когда мы добавим новое значение в перечисление GymActivity, наш код внезапно перестанет компилироваться. Компилятор Kotlin обнаружит, что оператор when больше не является исчерпывающим. Почти наверняка, когда вы добавляете в перечисление новое значение, вы хотите знать все места в коде, которые должны адаптироваться к этому значению. Исчерпывающий оператор when, не включающий значение else, именно это и делает.

Что произойдет, если оператору when не нужно возвращать значение (например, функция, в которой значение оператора when не является значением функции)?

Если оператор when не используется в качестве выражения, то компилятор Kotlin не заставляет его быть исчерпывающим. Однако вы получите предупреждение (желтый флаг в Android Studio), сообщающее, что рекомендуется это сделать, чтобы оператор when в перечислении был исчерпывающим.

Есть одна хитрость, которая заставит Kotlin интерпретировать любой оператор when как выражение (и, следовательно, быть исчерпывающим). Функция-расширение, определенная в примере 1.3, заставляет оператор when возвращать значение, как показано в примере 1.4. Поскольку у нее должно быть значение, Kotlin будет настаивать на том, чтобы оператор when был исчерпывающим.

Пример 1.3. Заставляем оператор when быть исчерпывающим

```
val <T> T.exhaustive: T
    get() = this
```

Пример 1.4. Проверяем, является ли оператор when исчерпывающим

```
when (activity) {
    GymActivity.BARRE -> true
    GymActivity.PILATES -> true
}.exhaustive // Ошибка! Оператор when не является исчерпывающим.
```

Перечисления — это способ создания класса, имеющего конкретный статический набор экземпляров. Kotlin предоставляет интересное обобщение этой возможности — *запечатанный класс*.

Запечатанные классы

Рассмотрим следующий код. Он определяет единственный тип `Result` с двумя подтипами.

Класс `Success` содержит значение, а класс `Failure` — исключение:

```
interface Result
data class Success(val data: List<Int>) : Result
data class Failure(val error: Throwable?) : Result
```

Обратите внимание, что с перечислением так делать нельзя. Все значения перечисления должны быть экземплярами одного типа. Здесь, однако, есть два разных типа, которые являются подтипами `Result`.

Мы можем создать новый экземпляр любого из двух типов:

```
fun getResult(): Result = try {
    Success(getDataOrExplode())
} catch (e: Exception) {
    Failure(e)
}
```

И, снова, выражение `when` — удобный способ управления `Result`:

```
fun processResult(result: Result): List<Int> = when (result) {
    is Success -> result.data
    is Failure -> listOf()
    else -> throw IllegalArgumentException("unknown result type")
}
```

Нам снова пришлось добавить ветку с оператором `else`, потому что компилятор Kotlin не знает, что `Success` и `Failure` — единственные подклассы `Result`. Где-нибудь у себя в программе вы можете создать еще один подкласс `Result` и добавить еще одно возможное значение. Следовательно, компилятору требуется ветка с оператором `else`.

Запечатанные классы делают для типов то, что перечисления делают для экземпляров. Они позволяют объявить компилятору, что существует фиксированный из-

вестный набор подтипов (в данном случае Success и Failure) для определенного родительского типа (Result). Для объявления используйте ключевое слово sealed, как показано в следующем коде:

```
sealed class Result
data class Success(val data: List<Int>) : Result()
data class Failure(val error: Throwable?) : Result()
```

Поскольку класс Result "запечатан", компилятор Kotlin знает, что Success и Failure — единственные возможные подклассы. И снова можно удалить оператор else из выражения when:

```
fun processResult(result: Result): List<Int> = when (result) {
    is Success -> result.data
    is Failure -> listOf()
}
```

Модификаторы видимости

Как в Java, так и в Kotlin модификаторы видимости определяют область видимости переменной, класса или метода. В Java есть три таких модификатора.

`private`

Ссылки видны только классу, в котором они определены, и из внешнего класса, если они определены во внутреннем классе.

`protected`

Ссылки видны классу, в котором они определены, или любым подклассам этого класса. Кроме того, они также видны из классов из того же пакета.

`public`

Ссылки видны везде.

В Kotlin также есть эти три модификатора. Тем не менее существуют тонкие различия. Хотя в Java их можно использовать только с объявлениями членов класса, в Kotlin они применяются с объявлениями членов класса и верхнего уровня.

`private`

Видимость объявления зависит от места определения:

- член класса, объявленный как private, виден только в том *классе*, в котором он определен;
- объявление верхнего уровня видно только в том *файле*, в котором оно определено.

`protected`

Защищенные объявления видны только в классе, в котором они определены, и его подклассах.

`public`

Ссылки видны везде, как и в Java.

Помимо этих трех видов видимости в Java есть четвертый уровень — уровень видимости, *ограниченный рамками пакета*, который делает ссылки видимыми только из классов, находящихся в одном пакете. Объявление ограничено рамками пакета, если у него нет модификаторов видимости. Другими словами, в Java это уровень видимости по умолчанию.

В Kotlin подобной концепции нет[7]. Это может показаться удивительным, потому что разработчики Java часто используют этот уровень видимости, чтобы скрыть детали реализации от других пакетов в том же модуле. В Kotlin пакеты вообще не используются для определения области видимости — это просто пространства имен. Поэтому видимость по умолчанию в Kotlin иная — она *общедоступная*.

Тот факт, что в Kotlin нет уровня видимости, ограниченного рамками пакета, оказывает существенное влияние на то, как мы пишем и структурируем код. Чтобы гарантировать полную инкапсуляцию объявлений (классов, методов, полей верхнего уровня и т. д.), все эти объявления можно размещать в одном файле, и они будут закрытыми (`private`).

Иногда допустимо наличие нескольких тесно связанных классов, разбитых на разные файлы. Однако эти классы не смогут получить доступ к одноуровневым классам из одного и того же пакета, если только они не являются общедоступными (`public`) или внутренними (`internal`). Что такое `internal`? Это четвертый модификатор видимости, поддерживаемый Kotlin, который делает ссылку видимой в любом месте вмещающего *модуля*[8]. С точки зрения модуля, модификаторы `internal` и `public` идентичны. Тем не менее модификатор `internal` представляет интерес, если этот модуль задуман как библиотека — например, это зависимость для других модулей. Объявления с модификатором `internal` не видны из модулей, которые импортируют вашу библиотеку. Таким образом, он полезен, если вы хотите скрыть объявления от внешнего мира.

Модификатор `internal` не предназначен для определения области видимости внутри модуля, что и делает уровень видимости, ограниченный рамками пакета, в Java. В Kotlin это невозможно. Можно немного жестче ограничить видимость, используя модификатор `private`.

Резюме

В табл. 1.1 представлены некоторые ключевые различия между Java и Kotlin.

---

[7] По крайней мере, в Kotlin версии 1.5.20. Пока мы пишем эти строки, Jetbrains рассматривает возможность добавления в язык модификатора для уровня видимости, ограниченного рамками пакета.

[8] Модуль — это набор файлов Kotlin, скомпилированных вместе.

Таблица 1.1. Различия между Java и Kotlin

Особенность	Java	Kotlin
Содержимое файла	Отдельный файл содержит один класс верхнего уровня	Отдельный файл может содержать любое количество классов, переменных или функций
Переменные	Используйте ключевое слово `final`, чтобы сделать переменную неизменяемой. Переменные являются изменяемыми по умолчанию; определяются на уровне класса	Используйте ключевое слово `val`, чтобы сделать переменную доступной только для чтения, или ключевое слово `var`, если речь идет о чтении или записи. Определяются на уровне класса или могут существовать независимо от класса
Автоматическое определение типа	Требуются типы данных. `Date date = new Date();`	Типы данных можно определить автоматически, например `val date = Date()`, или явно, например `val date: Date = Date()`
Преобразование ссылочных типов в примитивные, и наоборот	В языке Java такие примитивные типы данных, как `int`, рекомендуются для более дорогих операций, поскольку они менее затратны, чем, например, `Integer`. Однако у классов-оберток есть много полезных методов	В Kotlin нет примитивных типов "из коробки". Все является объектом. При компиляции для JVM сгенерированный байт-код по возможности выполняет автоматическое преобразование ссылочных типов в примитивные
Модификаторы доступа	Общедоступные и защищенные классы, функции и переменные можно расширить и переопределить	Будучи функциональным языком, Kotlin поощряет неизменяемость, когда это возможно. Классы и функции по умолчанию имеют модификатор `final`
Модификаторы доступа в многомодульных проектах	По умолчанию видимость ограничена рамками пакета	Уровня видимости, ограниченного рамками пакета, нет, доступ по умолчанию открытый. Новый идентификатор `internal` обеспечивает видимость в том же модуле
Функции	Все функции являются методами	В Kotlin есть функциональные типы. Типы данных функции выглядят так, например: `(param: String) -> Boolean`
Допустимость значений `null`	Любой не примитивный объект может содержать значение `null`	Значение `null` можно задать только для ссылок, которые явно поддерживают такое значение и объявляются с помощью суффикса `?`: `val date: Date? = new Date()`

Таблица 1.1 (окончание)

Особенность	Java	Kotlin
Ключевое слово `static` против `const`	Ключевое слово `static` связывает переменную с определением класса, а не с экземпляром	Ключевого слова `static` нет. Используйте модификатор `private` и ключевое слово `const` или объект-компаньон

Поздравляем, вы только что прочитали первую главу, посвященную основам Kotlin. Прежде чем говорить о применении Kotlin для Android, нужно обсудить коллекции и преобразование данных. Понимание основных функций преобразования данных в Kotlin даст вам необходимую основу для понимания Kotlin как функционального языка.

ГЛАВА 2
Фреймворк коллекций Kotlin

В предыдущей главе был представлен обзор синтаксиса языка Kotlin. Как и в любом языке, синтаксис — это основа, но не более того. Когда дело доходит до фактической работы, одного синтаксиса недостаточно. Для этого нужны выражения и идиомы, которые легко собрать в полезный код, а кроме того, необходимо сделать так, чтобы другие разработчики могли с легкостью понимать их и изменять.

Одним из важных аспектов почти каждого современного языка является его *фреймворк коллекций* — способы группировки объектов и библиотеки функций, которые ими управляют.

На момент своего появления фреймворк коллекций Java был самым современным. Сегодня, более двадцати лет спустя, основные структуры данных, предоставляемые новыми языками, не сильно изменились. Все контейнеры, с которыми мы знакомы по фреймворку Java (или даже самые ранние версии `stdlib` из C++), по-прежнему используются: `Iterable`, `Collection`, `List`, `Set` и `Map` (так они называются в Java). Однако в ответ на широкое признание функциональных стилей программирования фреймворки коллекций современных языков, таких как Swift и Scala, обычно предоставляют набор распространенных функций высшего порядка, которые работают с коллекциями: `filter`, `map`, `flatmap`, `zip` и т. д. Вы найдете эти функции во фреймворке коллекций из стандартной библиотеки Kotlin.

В этой главе сначала мы познакомимся с самими коллекциями и несколькими интересными расширениями, которые предоставляет Kotlin. После этого мы подробно рассмотрим некоторые мощные функции высшего порядка, которые работают с коллекциями.

Основные понятия

Фреймворк коллекций Kotlin включает в себя структуры данных из фреймворка коллекций Java в качестве подмножества. Он объединяет базовые классы Java и некоторые нововведения и добавляет преобразования для работы с ними.

Начнем это глубокое погружение в библиотеку коллекций с краткого обзора расширений самих структур данных.

Совместимость с Java

Поскольку идеальная совместимость с Java является основной целью языка Kotlin, типы данных коллекций Kotlin основаны на их аналогах из Java. Эта взаимосвязь показана на рис. 2.1.

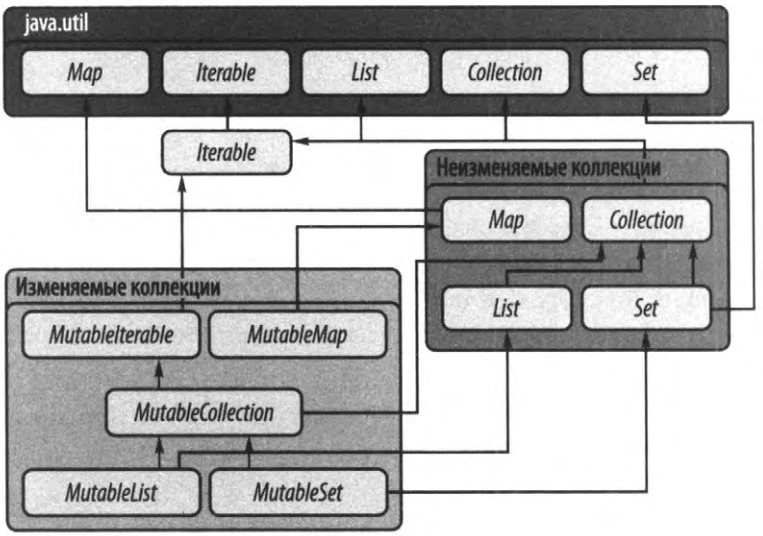


Рис. 2.1. Иерархия типов коллекций Kotlin и ее связь с Java

Делая типы коллекций Kotlin подтипами их аналогов в Java, Kotlin сохраняет все функциональные возможности фреймворка коллекций Java. По большей части Kotlin расширяется, но не изменяет структуру Java. Он просто добавляет новые, функциональные методы.

Есть здесь и одно существенное исключение — изменяемость.

Изменяемость

Возможно, вполне логично, что язык, включающий изменяемость в свой синтаксис, также встраивает изменяемость в свою систему коллекций.

Во фреймворке коллекций Kotlin определяет две различные иерархии типов: одну для изменяемых коллекций и другую для неизменяемых. Это можно увидеть в примере 2.1.

Пример 2.1. Изменяемые и неизменяемые списки

```
val mutableList = mutableListOf(1, 2, 4, 5)
val immutableList = listOf(1, 2, 4, 5)
mutableList.add(4) // компилируется
// Не компилируется: у ImmutableList нет метода 'add'.
immutableList.add(2)
```

Изменяемость — это противоположность *неизменяемости*. Изменяемый объект можно изменить, а неизменяемый нельзя. Данное различие имеет решающее значение при попытке оптимизации кода. Поскольку их нельзя изменять, неизменяемые объекты можно безопасно разделить между несколькими потоками. Однако изменяемый объект нужно явно сделать потокобезопасным, если он должен использоваться совместно. Потокобезопасность требует блокировки или копирования, что может быть затратно.

К сожалению, Kotlin не может гарантировать неизменяемость своих неизменяемых коллекций. У неизменяемых коллекций просто нет функций записи (`add`, `remove`, `put` и т.д.). Особенно, когда коллекция Kotlin передается в код Java, где ограничения неизменяемости Kotlin не обеспечиваются системой типов — нет никакой гарантии, что содержимое коллекции не изменится.

Обратите внимание, что изменяемость коллекции не связана с изменяемостью объекта, содержащегося в коллекции. В качестве очень простого примера рассмотрим следующий код:

```
val deeplist = listOf(mutableListOf(1, 2), mutableListOf(3, 4))

// Не компилируется: "Unresolved reference: add"
deeplist.add(listOf(3))

deeplist[1][1] = 5 // работает
deeplist[1].add(6) // работает
```

Переменная `deeplist` представляет собой `List<MutableList<Int>>`. Это есть и всегда будет список из двух списков. Однако содержимое списков, которые хранит эта переменная, может увеличиваться, уменьшаться и изменяться.

Создатели Kotlin активно исследуют всё на предмет неизменяемости. Прототип библиотеки `kotlinx.collections.immutable` задуман как набор действительно неизменяемых коллекций. Чтобы использовать их в собственном проекте Android или Kotlin, добавьте следующую зависимость в файл `build.gradle`:

```
implementation \
'org.jetbrains.kotlinx:kotlinx-collections-immutable:$IC_VERSION'
```

Хотя *библиотека неизменяемых коллекций Kotlinx* использует самые современные алгоритмы и оптимизирует их таким образом, чтобы они были очень быстрыми по сравнению с другими реализациями неизменяемых коллекций JVM, эти действительно неизменяемые коллекции по-прежнему на порядок медленнее, чем их изменяемые аналоги. В настоящее время с этим ничего не поделаешь.

Однако многие современные разработчики готовы пожертвовать производительностью ради безопасности, которую обеспечивает неизменяемость, особенно в контексте параллелизма[1].

---

[1] Елизаров Р. Интервью о неизменяемой библиотеке Kotlin Collections по электронной почте. 8 октября 2020 г.

Перегруженные операторы

Kotlin поддерживает организованную способность перегружать значения определенных инфиксных операторов, в частности + и -. Фреймворк коллекций Kotlin хорошо использует эту способность. В качестве демонстрации рассмотрим простейшую реализацию функции для преобразования List<Int> в List<Double>:

```
fun naiveConversion(intList: List<Int>): List<Double> {
    var ints = intList
    var doubles = listOf<Double>()
    while (!ints.isEmpty()) {
        val item = ints[0]
        ints = ints - item
        doubles = doubles + item.toDouble()
    }
    return doubles
}
```

Не делайте этого. Единственное, в чем состоит эффективность данного примера, — это демонстрация использования двух инфиксных операторов: + и -. Первый оператор добавляет элемент в список, а второй удаляет из него элемент.

Операнд слева от оператора + или - может определять поведение этого оператора. Контейнеры, когда они появляются слева от + или -, определяют две реализации для каждого из двух операторов: одна, когда правый операнд — это другой контейнер, а вторая, когда это не так.

При добавлении объекта, относящегося к категории *non-container*, в контейнер создается новый контейнер, в котором есть все элементы из левого операнда (контейнера) с добавлением нового элемента (правый операнд). При добавлении двух контейнеров создается новый контейнер, у которого есть все элементы от обоих.

Аналогичным образом при извлечении объекта из контейнера создается новый контейнер, где есть все, кроме первого вхождения левого операнда. При извлечении одного контейнера из другого создается новый контейнер, содержащий элементы левого операнда, при этом *все* вхождения *всех* элементов правого операнда удалены.

Операторы + и - сохраняют порядок при упорядочении базового контейнера. Например:

```
(listOf(1, 2) + 3)
    .equals(listOf(1, 2, 3))    // истина
(listOf(1, 2) + listOf(3, 4))
    .equals(listOf(1, 2, 3, 4)) // истина
```

Создание контейнеров

В Kotlin нет способа для выражения литералов контейнера. Например, не существует синтаксического способа составить список (List) чисел 8, 9 и 54. Равно как нельзя составить множество (Set) строк "Дадли" и "Мазер". Вместо этого есть удобные методы создания почти таких же элегантных контейнеров. В коде из примера 2.1 показаны два простых варианта создания списков. Существуют также методы со словом Of на конце для создания изменяемых и неизменяемых списков, множеств и ассоциативных массивов.

Чтобы создать ассоциативный массив, нужно знать один хитрый трюк. Функция mapOf принимает список Pairs в качестве аргумента. Каждая из пар предоставляет ключ (первое значение пары) и значение (второе значение пары). Напомним, что Kotlin поддерживает расширенный набор инфиксных операторов. Среди них есть оператор to, который создает новую пару с левым операндом в качестве первого элемента и правый операнд в качестве второго элемента. Объедините эти две функции, и вы благополучно сможете создать ассоциативный массив:

```
val map = mapOf(1 to 2, 4 to 5)
```

Тип содержимого контейнера выражается с использованием синтаксиса обобщений, очень похожего на тот, что есть в Java. Например, тип переменной map из предыдущего кода — это Map<Int,Int>, контейнер, который сопоставляет ключи Int с их значениями типа Int.

Компилятор Kotlin довольно толково определяет типы содержимого контейнеров, созданных фабричными методами. Очевидно, что в этом примере:

```
val map = mutableMapOf("Earth" to 3, "Venus" to 4)
```

тип map — MutableMap<String, Int>. А как насчет этого?

```
val list = listOf(1L, 3.14)
```

Kotlin выберет ближайший тип в дереве иерархии типов, который является предком всех элементов контейнера (этот тип называется *типом верхней границы*). В данном случае он выберет тип Number, ближайшего предка Long и Double. Переменная list имеет автоматически определяемый тип List<Number>.

Однако можно добавить String, как показано ниже:

```
val list = mutablelistOf(1L, 3.14, "e")
```

Единственный тип, который является предком всех элементов (Long, Double и String), — это тип Any. Тип переменной list — MutableList<Any>.

Однако, как вы помните из *главы 1*, типы Any и Any? — это не одно и то же. Следующий код не будет компилироваться:

```
list.add(null)  // Ошибка: Null не может быть значением типа Any,
                // который не поддерживает значения null
```

Чтобы список мог содержать значение null, пришлось бы явно указать его тип:

```
val list: MutableList<Any?> = mutablelistOf(1L, 3.14, "e")
```

Теперь можно создавать коллекции. Но что нам с ними делать?

Функциональное программирование

Мы работаем с ними! Почти все операции, которые мы будем здесь обсуждать, основаны на парадигме функционального программирования. Чтобы понять их контекст и мотивацию, рассмотрим парадигму.

Объектно-ориентированное программирование (ООП) и *функциональное программирование* (ФП) представляют собой парадигмы проектирования программного обеспечения. Архитекторы программного обеспечения осознали перспективы функционального программирования вскоре после его появления в конце 1950-х годов. Однако на раннем этапе функциональные программы, как правило, были медленными, и только недавно функциональный стиль смог бросить вызов более прагматичной императивной модели производительности. По мере того как программы становятся более сложными и трудными для понимания, параллелизм становится неизбежным, а оптимизация компилятора улучшается, функциональное программирование превращается из симпатичной академической игрушки в полезный инструмент, которым должен владеть каждый разработчик. Функциональное программирование поощряет *неизменяемость*. В отличие от функций в коде, математические функции ничего не меняют. Они ничего не "возвращают". У них просто есть значение. Подобно тому, как "4" и "2 + 2" обозначают одно и то же число, конкретная функция, вычисляемая с конкретными параметрами, — это просто имя (возможно, подробное имя!) значения. Поскольку математические функции не меняются, они не подвластны времени, что чрезвычайно полезно при работе в параллельном окружении.

Хотя это разные парадигмы, они могут сосуществовать. Java, безусловно, разрабатывался как объектно-ориентированный язык, а полностью совместимый с ним Kotlin может воспроизводить алгоритмы Java почти дословно. Однако, как было сказано в предыдущей главе, истинная мощь Kotlin заключается в расширяемых возможностях функционального программирования. Нередко кто-то начинает писать фразы вроде "Java на Kotlin". По мере того как разработчики начинают чувствовать себя комфортнее, они, как правило, склоняются к более идиоматичному языку Kotlin, что во многом связано с применением возможностей функционального программирования.

Сравнение функционального и процедурного программирования: простой пример

В следующем коде показан способ работы с коллекцией с использованием процедурного программирования:

```
fun forAll() {
    for (x in collection) { doSomething(x) }
}
```

В этом примере цикл for перебирает список. Он выбирает элемент из коллекции и присваивает его переменной x. После этого он вызывает метод doSomething для элемента и делает это для каждого элемента из списка.

Единственным ограничением коллекции является тот факт, что у вас должен быть способ получить каждый из ее элементов ровно один раз. Именно эта возможность инкапсулируется типом Iterable<T>.

Функциональная парадигма, безусловно, менее сложна: никаких дополнительных переменных и специального синтаксиса. Всего один вызов метода:

```
fun forAll() = collection.forEach (::doSomething)
```

Метод forEach принимает в качестве аргумента функцию. Этот аргумент, в данном случае doSomething, — это функция, которая принимает один параметр типа, содержащегося в коллекции. Другими словами, если коллекция представляет собой список значений типа String, то функция doSomething должна выглядеть так: doSomething(s: String). Если коллекция — это Set<Freeptootsie>, то она должна выглядеть так: doSomething(ft: Freeptootsie). Метод forEach вызывает свой аргумент (doSomething) с каждым элементом коллекции в качестве параметра.

Может показаться, что разница незначительная, но это не так. Метод forEach — гораздо лучшее разделение ответственности.

У типа Iterable<T> есть состояние, он упорядочен и зависит от времени. Любой, кто когда-либо имел дело с исключением ConcurrentModificationException, знает: вполне возможно, что состояние итератора может не совпадать с состоянием коллекции, которую он перебирает. Хотя оператор forEach в Kotlin не полностью защищен от исключения ConcurrentModificationException, эти исключения возникают в коде, который фактически является параллельным.

Еще более важно, что механизм, используемый коллекцией для применения переданной функции к каждому из своих элементов, полностью зависит от самой коллекции. В частности, здесь нет внутреннего соглашения о порядке, в котором функция будет вычисляться для элементов коллекции.

Коллекция может, например, разделить свои элементы на группы. Она может передать каждую из этих групп отдельному процессору, а затем повторно собрать результаты. Такой подход особенно интересен в то время, когда количество ядер в процессоре быстро увеличивается. Контракт Iterator<T> не может поддерживать данный вид параллельного выполнения.

Функциональное программирование в Android

У Android причудливая история функционального программирования. Поскольку его виртуальная машина не имеет ничего общего с Java, улучшения в языке Java не всегда доступны разработчикам приложений для Android. Некоторые наиболее важные изменения в Java, включая лямбда-выражения и ссылки на методы, не поддерживались в Android в течение некоторого времени после того, как они появились в Java 8.

Хотя Java мог компилировать эти новые функции, а DEX (байт-код Android) мог даже представлять их (хотя, возможно, неэффективно), инструментальная цепочка Android не могла преобразовать скомпилированный байт-код Java в код DEX, который можно было бы запускать в системе Android.

Первой попыткой восполнить этот пробел был пакет *RetroLambda*. Затем последовали другие решения для дополнительных библиотек, иногда с запутанными правилами (например, в случае с Android Gradle Plugin [AGP] версии 3.0 и выше, если вы хотели использовать API Java Streams, нужно было ориентироваться как минимум на API Android 24).

С появлением Kotlin все эти ограничения ушли. Последние версии AGP будут поддерживать функциональное программирование даже в старых версиях Android. Теперь можно использовать полный пакет коллекций Kotlin в любой поддерживаемой платформе.

Функции-преобразователи

В этом разделе вы увидите, как Kotlin привносит возможности функционального программирования в коллекции, чтобы обеспечить элегантные и безопасные способы управления ими. Так же, как и в предыдущей главе, где мы не стали изучать весь синтаксис Kotlin, в этой главе мы не будем пытаться рассмотреть все библиотечные функции Kotlin. Не обязательно запоминать их все. Тем не менее для идиоматического и эффективного использования Kotlin важно освоить несколько ключевых функций и понять, как они работают.

Булевы функции

Удобный набор функций коллекции возвращает логическое значение, чтобы указать, есть ли у коллекции заданный атрибут или нет. Функция `any()`, например, вернет `true`, если коллекция содержит хотя бы один элемент. При использовании с предикатом, например `any { predicate(it) }`, `any` вернет `true`, если предикат возвращает `true` для любого элемента коллекции:

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.any()                  // истина
val isAnyOdd = nums.any { it % 1 > 0 }  // истина
val isAnyBig = nums.any { it > 1000}    // ложь
```

Когда лямбда-выражение принимает только один аргумент, а компилятор Kotlin может выяснить это с помощью автоматического определения типов (обычно это возможно), можно опустить объявление параметра и использовать неявный параметр `it`. В предыдущем примере он указан дважды, в определениях предикатов метода `any`.

Еще одна булева функция, `all { predicate }`, возвращает `true`, только если каждый элемент в списке соответствует предикату:

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.all { it % 1 > 0 } // ложь
```

Противоположностью `any` является функция `none`. Без предиката эта функция возвращает `true`, только если в коллекции нет элементов. С предикатом `none { predicate }` возвращает `true`, только если предикат не возвращает `true` ни для одного из элементов коллекции. Например:

```
val nums = listOf(10, 20, 100, 5)
val isAny = nums.none()            // ложь
val isAny4 = nums.none { it == 4 } // истина
```

Фильтры

Базовая функция `filter` возвращает новую коллекцию, содержащую только те элементы исходной коллекции, которые соответствуют заданному предикату. В этом примере переменная `numbers` будет содержать список с единственным значением `100`:

```
val nums = listOf(10, 20, 100, 5)
val numbers = nums.filter { it > 20 }
```

Функция `filterNot` действует наоборот. Она возвращает элементы, которые *не* соответствуют предикату. В этом примере переменная `numbers` будет содержать три элемента, 10, 20 и 5: элементы переменной `nums`, не превышающие 20:

```
val nums = listOf(10, 20, 100, 5)
val numbers = nums.filterNot { it > 20 }
```

Очень удобным частным случаем `filterNot` является функция `filterNotNull`. Она удаляет все значения `null` из коллекции:

```
val nums = listOf(null, 20, null, 5)
val numbers = nums.filterNotNull() // { 20, 5 }
```

В этом примере переменная `numbers` будет списком, состоящим из двух элементов — 20 и 5.

Функция *map*

Функция `map` применяет свой аргумент к каждому элементу коллекции и возвращает коллекцию результирующих значений. Обратите внимание, что она не изменяет коллекцию, к которой применяется, а возвращает новую.

Вот определение функции `map` для типа `Array`:

```
inline fun <T, R> Array<out T>.map(transform: (T) -> R): List<R>
```

Давайте разберем ее.

В левой части видно слово inline. "Веселая" часть уже должна быть понятна. Но что насчет inline?

Ключевое слово inline велит компилятору Kotlin копировать байт-код для функции непосредственно в двоичный файл всякий раз при вызове метода, вместо того чтобы генерировать передачу в одну скомпилированную версию. Когда количество инструкций, необходимых для вызова функции, составляет значительную долю от общего количества, необходимого для ее запуска, имеет смысл использовать функцию с ключевым словом inline как компромисс между пространством и временем. Иногда так можно устранить затраты на дополнительное выделение памяти под объекты, которые требуются для некоторых лямбда-выражений.

Далее идут <T, R> — это две свободные переменные типа, используемые в определении функции. Мы к ним еще вернемся.

После следует описание приемника, Array<out T>. Функция map представляет собой функцию-расширение для типа Array: это функция для массива, элементы которого имеют тип T (или один из суперклассов T, например Any).

Далее идет параметр map. Это функция transform: (T) -> R. Она принимает в качестве аргумента что-то типа T и возвращает что-то типа R. Что же, это интересно! Массив, к которому будет применяться функция, заполнен объектами типа T! Функцию можно применить к элементам массива.

Наконец, идет тип возвращаемого значения. Это List<R>, список, элементы которого имеют тип R. R — это то, что вы получите, если примените преобразование к элементам массива (T).

Все работает. Вызов map для массива с функцией, которую можно применить к элементам массива, вернет новый список, содержащий результаты применения функции к каждому элементу массива.

Вот пример, который возвращает список начальных дат для документации по персоналу, где эти начальные даты хранятся в виде строк:

```
data class Hire(
    val name: String,
    val position: String,
    val startDate: String
)

fun List<Hire>.getStartDates(): List<Date> {
    val formatter = SimpleDateFormat("yyyy-MM-d", Locale.getDefault())
    return map {
        try {
            formatter.parse(it.startDate)
        } catch (e: Exception) {
            Log.d(
                "getStartDates",
```

```
                "Unable to format first date. $e")
            Date()
        }
    }
}
```

Возможно, вы задаетесь вопросом: что произойдет, если функция-преобразователь не вернет значение? Но у функций Kotlin *всегда* есть значение! Например:

```
val doubles: List<Double?> = listOf(1.0, 2.0, 3.0, null, 5.0)
val squares: List<Double?> = doubles.map { it?.pow(2) }
```

В этом примере переменная `squares` будет списком `[1.0, 4.0, 9.0, null, 25.0]`. Из-за условного оператора `?.` в функции-преобразователе значение — это квадрат аргумента, если этот аргумент не равен нулю. Однако в противном случае функция имеет значение `null`.

В библиотеке Kotlin есть несколько вариантов функции `map`. Один из них, `mapNotNull`, решает следующие ситуации:

```
val doubles: List<Double?> = listOf(1.0, 2.0, 3.0, null, 5.0)
val squares: List<Double?> = doubles.mapNotNull { it?.pow(2) }
```

Значение переменной `squares` в данном примере равно `[1.0, 4.0, 9.0, 25.0]`.

Еще один вариант функции `map` — `mapIndexed`. `mapIndexed` также принимает функцию в качестве аргумента. Однако, в отличие от `map`, аргумент `mapIndexed` принимает элемент коллекции в качестве второго параметра (а не первого и единственного параметра, как аргумент `map`). Аргумент `mapIndexed` принимает в качестве первого параметра значение `Int` — порядковый номер, указывающий позицию в коллекции элемента, который является вторым параметром: 0 для первого элемента, 1 для второго и т. д.

Существуют функции отображения для большинства объектов, подобных коллекциям. Есть даже аналогичные функции для `Map` (хотя они и не являются подтипами `Collection`): функции `Map::mapKeys` и `Map::mapValues`.

Функция *flatMap*

Функция `flatMap` может показаться абстрактной и не особенно полезной, что делает ее трудной для понимания. Но оказывается, что хоть она и абстрактная, польза от нее все-таки есть. Начнем с аналогии. Предположим, вы решили обратиться к членам вашей старой школьной команды по дебатам, но не знаете, как с ними связаться. Однако вы помните, что у вас есть фотоальбомы за все четыре года, что вы учились в школе, и что в каждом фотоальбоме имеется фотография команды.

Вы решили разделить процесс установления контакта с участниками на два этапа. Сначала вы рассмотрите каждую фотографию, постараетесь идентифицировать каждого изображенного на ней человека и составите список людей, которых вы идентифицировали. Затем вы объедините четыре списка в один список всех членов команды.

Вот как работает эта функция! Все дело в контейнерах. Обобщим.

Предположим, у вас есть контейнер с неким содержимым. Это CON<T>. В примере с фотоальбомом CON<T> — это четыре фотографии, Set<Photo>. Далее у вас есть функция, которая отображает T -> KON<R>. То есть она берет элемент CON и превращает его в контейнер KON нового типа, элементы которого относятся к типу R. Вы идентифицировали каждого человека на одной из фотографий и создали список имен людей на фото. KON — это бумажный список, а R — это имя человека.

Результатом работы функции flatMap в данном примере является сводный список имен.

Вот функция flatMap для CON<T>:

```
fun <T, R> CON<T>.flatMap(transform: (T) -> KON<R>): KON<R>
```

Обратите внимание, просто для сравнения, чем flatMap отличается от map. У функции map для контейнера CON, использующего ту же функцию-преобразователь, следующая сигнатура:

```
fun <T, R> CON<T>.map(transform: (T) -> KON<R>): CON<KON<R>>
```

Функция flatMap "уплощает" один из контейнеров.

Раз уж мы заговорили об этом, рассмотрим очень распространенный пример использования функции flatMap:

```
val list: List<List<Int>> = listOf(listOf(1, 2, 3, 4), listOf(5, 6))
val flatList: List<Int> = list.flatMap { it }
```

Переменная flatList будет иметь значение [1, 2, 3, 4, 5, 6].

Этот пример может сбить с толку. В отличие от предыдущего примера, в котором набор фотографий был преобразован в списки имен, а затем эти списки были объединены, здесь оба типа контейнеров CON и KON одинаковы: это List<Int>. Из-за этого может быть сложно увидеть, что происходит на самом деле.

Однако просто чтобы доказать, что все работает, проделаем упражнение по привязке величин в этом несколько запутанном примере к типам в описании функции. Функция применяется к List<List<Int>>, поэтому мы должны получить List<Int>. Функция-преобразователь — это функция тождества. Другими словами, это (List<Int>) -> List<Int>: она возвращает свой параметр. Это означает, что KON<R> также должен иметь тип List<Int>, а R — это Int. Таким образом, функция flatMap вернет KON<R>, List<Int>.

Все работает.

Группировка

Помимо фильтрации стандартная библиотека Kotlin предоставляет еще один небольшой набор функций-расширений для преобразования коллекций, которые группируют элементы коллекции. Сигнатура функции groupBy, например, выглядит так:

```
inline fun <T, K> Array<out T>
    .groupBy(keySelector: (T) -> K): Map<K, List<T>>
```

Как это часто бывает, можно интуитивно понять поведение этой функции, просто взглянув на информацию о типе. `groupBy` — это функция, которая принимает массив (`Array`) (в данном случае `Array`, но существуют эквиваленты и для других типов контейнеров). Для каждого элемента массива применяется метод `keySelector`. Он неким образом помечает объект значением типа `K`. Тип значения, возвращаемого из метода `groupBy`, — это ассоциативный массив каждой из этих меток для списка элементов, которым `keySelector` присвоил эту метку.

Нам поможет пример:

```
val numbers = listOf(1, 20, 18, 37, 2)
val groupedNumbers = numbers.groupBy {
    when {
        it < 20 -> "less than 20"
        else -> "greater than or equal to 20"
    }
}
```

Переменная `groupedNumbers` теперь содержит `Map<String, List<Int>>`. У массива есть два ключа: "меньше 20" (`"less than 20"`) и "больше или равно 20" (`"greater than or equal to 20"`). Значение первого ключа — это список `[1, 18, 2]`. Значение второго — `[20, 37]`.

Ассоциативные массивы, сгенерированные из функций, хранят порядок элементов из исходной коллекции — в списках, которые представляют собой значения ключей, получившиеся в результате ассоциативного массива.

Сравнение итераторов и последовательностей

Предположим, вы собираетесь покрасить свой стол. Вы решили, что он будет выглядеть намного лучше, если у него будет приятный оттенок коричневого, а не обычный рыжеватый цвет. Вы направляетесь в магазин красок и обнаруживаете, что здесь есть около 57 цветов, которые могут вам подойти.

Что дальше? Вы покупаете образцы каждого из этих цветов, чтобы забрать домой? Почти наверняка нет! Вместо этого вы покупаете образцы двух или трех цветов, которые кажутся вам многообещающими, чтобы испробовать их. Если они окажутся не такими, как вам хочется, вы вернетесь в магазин и купите еще три. Вместо того чтобы покупать образцы всех цветов-кандидатов и перебирать их, вы создаете процесс, который позволит вам получить следующие цвета-кандидаты, учитывая те, которые вы уже опробовали.

Аналогичным образом последовательность отличается от итератора. Итератор — это способ получить каждый элемент из существующей коллекции ровно один раз. Коллекция уже существует. Все, что нужно сделать итератору, — это упорядочить ее.

С другой стороны, последовательность не обязательно поддерживается коллекцией. Последовательности поддерживаются *генераторами*. Генератор — это функция, предоставляющая следующий элемент в последовательности. Если вам нужно больше образцов краски, у вас есть способ получить их: вы возвращаетесь в мага-

зин и покупаете еще. Не обязательно покупать все и перебирать их. Вам просто нужно купить пару, потому что вы знаете, как получить еще.

Вы можете остановиться, когда найдете нужный цвет, и, если повезет, это произойдет до того, как вы заплатите за образцы всех возможных цветов.

В Kotlin поиск краски для стола можно выразить следующим образом:

```
val deskColor = generateSequence("burnt umber") {
    buyAnotherPaintSample(it)
}.first { looksGreat(it) }

println("Start painting with ${deskColor}!")
```

Это эффективный алгоритм. В среднем те, кто занимаются окраской столов и используют его, покупают только 28 образцов красок вместо 57.

Поскольку последовательности "ленивы" — они генерируют следующий элемент только при необходимости — они могут быть очень и очень полезны при оптимизации операций, даже для коллекций с фиксированным содержимым. Предположим, например, что у вас есть список URL-адресов, и вы хотите знать, какой из них является ссылкой на страницу, содержащую изображение кота. Это можно было бы сделать следующим образом:

```
val catPage = listOf(
    "http://ragdollies.com",
    "http://dogs.com",
    "http://moredogs.com")
    .map { fetchPage(it) }
    .first { hasCat(it) }
```

Этот алгоритм скачает все страницы. Если сделать то же самое, используя последовательность:

```
val catPage = sequenceOf(
    "http://ragdollies.com",
    "http://dogs.com",
    "http://moredogs.com")
    .map { fetchPage(it) }
    .first { hasCat(it) }
```

то скачается лишь первая страница. Последовательность предоставит первый URL-адрес функция `map` извлечет его, и функция `first` будет удовлетворена. Другие страницы скачиваться не будут.

Однако будьте осторожны! Не запрашивайте все элементы бесконечной коллекции! Этот код, например, в конечном счете выдаст ошибку `OutOfMemory`:

```
val nums = generateSequence(1) { it + 1 }
    .map { it * 7 }                 // Прекрасно
    .filter { it mod 10000 = 0 }    // Тоже неплохо
    .asList()                       // СБОЙ!
```

Пример

Рассмотрим все это на примере.

Мы только что познакомились с несколькими удобными функциями, предоставляемыми стандартной библиотекой Kotlin для управления коллекциями. Используя их, можно создавать надежные реализации сложной логики. В качестве иллюстрации обратимся к примеру на основе реального приложения, которое используется на заводе по производству авиационных двигателей.

Проблема

Компания Bandalorium Inc. производит авиационные двигатели. Каждая деталь двигателя имеет уникальный идентификатор в виде серийного номера. Она проходит строгий процесс контроля качества, в ходе которого регистрируются числовые измерения ряда критических характеристик.

Атрибутом детали двигателя является любая измеримая характеристика. Например, это может быть внешний диаметр трубы. Электрическое сопротивление провода может быть вторым атрибутом. Третьим может быть способность части отражать определенный цвет. Единственное требование состоит в том, что измерение атрибута должно давать одно числовое значение.

Одна из вещей, которую компания хочет отслеживать, — это точность процесса производства. Ей необходимо контролировать размеры производимых деталей и следить за тем, меняются ли они с течением времени.

Таким образом, задача состоит в следующем.

Дан список измерений атрибутов деталей, изготовленных в течение определенного интервала времени (скажем, три месяца). Нужно создать отчет в формате CSV, аналогичный тому, что показан на рис. 2.2. Как видите, отчет должен быть отсортирован по времени проведения измерения.

	A	B	C	D	E
1	Date	Serial	AngleOfAttack	ChordLength	PaintColor
2	27-07-2020 15:15:00	HC14	15.08	0.71	7,951,688.0
3	27-07-2020 15:25:00	HC13	15.11	0.69	-
4	27-07-2020 15:35:00	HC12	15.05	0.7	-
5	27-07-2020 15:45:00	HC11	15.1	0.68	2201331.0

Рис. 2.2. Пример вывода в формате CSV

Сейчас самое время отложить книгу в сторону на минутку и подумать, как бы вы подошли к решению этой проблемы. Может быть, просто набросать достаточное количество высокоуровневого кода, чтобы быть уверенными в том, что сможете ее решить.

Реализация

Атрибут в Kotlin можно было бы представить следующим образом:

```
data class Attr(val name: String, val tolerance: Tolerance)

enum class Tolerance {
    CRITICAL,
    IMPORTANT,
    REGULAR
}
```

`name` — это уникальный идентификатор атрибута. В классе `Tolerance` указана значимость атрибута для качества конечного продукта: `CRITICAL`, `IMPORTANT` или `REGULAR`.

У каждого атрибута, вероятно, есть много другой связанной с ним информации. Наверняка есть запись единиц измерения (сантиметры, джоули и т. д.), описание приемлемых значений и, возможно, процедура, используемая для их измерения. В данном примере мы не будем обращать внимание на эти вещи.

Измерение атрибута для конкретной детали двигателя включает в себя:

- серийный номер измеряемой детали;
- временну́ю метку, указывающую время, когда было выполнено измерение;
- измеряемое значение.

Таким образом, измерение можно смоделировать в Kotlin следующим образом:

```
data class Point(
    val serial: String,
    val date: LocalDateTime,
    val value: Double)
```

Наконец, нам нужен способ связать измерение с измеряемым атрибутом. Моделируем связь:

```
data class TimeSeries(val points: List<Point>, val attr: Attr)
```

`TimeSeries` связывает список измерений с атрибутами.

Сначала мы создаем заголовок CSV-файла: заголовки столбцов, которые образуют первую строку (пример 2.2). Первые два столбца называются `date` и `serial`. Имена остальных столбцов — это отдельные имена атрибутов в наборе данных.

Пример 2.2. Создаем заголовок

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    val distinctAttrs = timeSeries
        .distinctBy { it.attr } ❶
        .map { it.attr }        ❷
        .sortedBy { it.name }   ❸
```

```
    val csvHeader = "date;serial;" +
        distinctAttrs.joinToString(";") { it.name } +
        "\n"

    /* Код удален для краткости */
}
```

❶ Используйте функцию `distinctBy`, чтобы получить список экземпляров `TimeSeries`, у которых разные значения атрибута `attr`.

❷ У нас есть список отдельных экземпляров `TimeSeries` из предыдущего шага, а нам нужен только `attr`, поэтому мы используем функцию `map`.

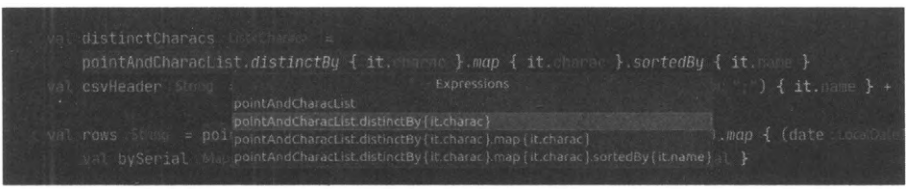
❸ Наконец, мы выполняем сортировку по алфавиту с помощью функции `sortedBy`. Это не обязательно, но почему бы и нет?

Теперь, когда у нас есть список различных характеристик, несложно отформатировать заголовок с помощью функции `joinToString`. Она преобразует список в строку, используя разделитель строк для вставки между каждым элементом списка. При необходимости даже можно указать префикс и/или постфикс.

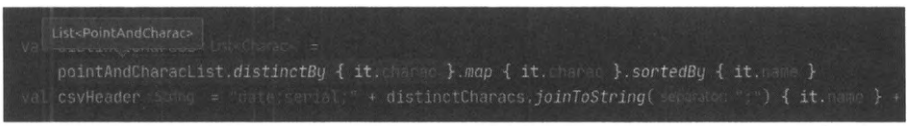
Часто полезно иметь возможность находить типы значений, возвращаемых из функций-преобразователей коллекций. Например, в примере 2.2, если вы активируете подсказки при вводе кода, то получите только тип всей цепочки, который определяется автоматически (тип переменной `differentAttrs`). В IntelliJ и Android Studio есть неплохая штука, которая может нам помочь!

Щелкните кнопкой мыши на `distinctCharacs` в исходном коде.

Нажмите сочетание клавиш <Ctrl>+<Shift>+<P>. Появится раскрывающийся список.



Выберите нужный шаг, и у вас перед глазами появится автоматический определяемый тип!



После создания заголовка мы создаем содержимое CSV-файла. Это самая техническая и интересная часть.

Остальная часть файла, которую мы пытаемся воспроизвести, сортирует данные по дате. Для каждой заданной даты предоставляется серийный номер детали, а затем измерение этой детали для каждого интересующего нас атрибута. Здесь нужно будет немного подумать, потому что в созданной нами модели эти вещи не связаны напрямую. `TimeSeries` содержит данные только по одному атрибуту, а нам потребуются данные по нескольким.

Обычный подход в этой ситуации состоит в том, чтобы выполнить объединение и уплощение (flattening) исходных данных в более удобную структуру, как показано в примере 2.3.

Пример 2.3. Объединение и уплощение данных

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    /* Код удален для краткости */

    data class PointWithAttr(val point: Point, val attr: Attr)

    // Для начала объединяем и уплощаем данные,
    // чтобы можно было работать со списком PointWithAttr.
    val pointsWithAttrs = timeSeries.flatMap { ts ->
        ts.points.map { point -> PointWithAttr(point, ts.attr) }

    /* Код удален для краткости */
}
```

На данном этапе мы связываем `Point` и соответствующий `Attr` в едином объекте `PointAndAttr`, что очень похоже на соединение двух таблиц в SQL.

Функция `flatMap` преобразует список объектов `TimeSeries`. "Под капотом" функция, применяемая `flatMap`, использует функцию `map`, `series.points.map { ... }` для создания списка `PointAndAttrs` для каждой точки в `TimeSeries`. Если бы вместо `flatMap` мы использовали функцию `map`, то создали бы `List<List<PointAndAttr>>`. Помните, однако, что функция `flatMap` уплощает верхний слой контейнера, поэтому результатом будет `List<PointAndAttr>`.

Теперь, когда мы "распространили" информацию об атрибутах в каждый экземпляр `Point`, создать CSV-файл будет довольно просто.

Мы сгруппируем список `pointWithAttrs` по дате, чтобы создать `Map<LocalDate>, List<PointWithAttr>`. Этот ассоциативный массив будет содержать список `pointWithAttrs` для каждой даты. Поскольку в примере, кажется, есть вторичная сортировка (по серийному номеру детали), нам придется сгруппировать каждый из списков в ранее сгруппированном массиве по порядковому номеру. Остальное — это просто форматирование строки, как показано в примере 2.4.

Пример 2.4. Создаем строки данных

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    /* Код удален для краткости */

    val rows = importantPointsWithAttrs.groupBy { it.point.date } ❶
    .toSortedMap()                                              ❷
    .map { (date, ptsWithAttrs1) ->
        ptsWithAttrs1
            .groupBy { it.point.serial }                        ❸
            .map { (serial, ptsWithAttrs2) ->
                listOf(                                         ❹
                    date.format(DateTimeFormatter.ISO_LOCAL_DATE),
                    serial
                ) + distinctAttrs.map { attr ->
                    val value = ptsWithAttrs2.firstOrNull { it.attr == attr }
                    value?.point?.value?.toString() ?: ""
                }
            }.joinToString(separator = "") {                    ❺
                 it.joinToString(separator = ";", postfix = "\n")
            }
    }.joinToString(separator = "")

    return csvHeader + rows                                     ❻
}
```

❶ Выполняем группировку по дате, используя функцию `groupBy`.

❷ Сортируем ассоциативный массив (по дате). Это не обязательно, но отсортированный файл легче читать.

❸ Выполняем группировку по серийному номеру.

❹ Создаем список значений для каждой строки.

❺ Форматируем каждую строку и собираем все строки с помощью функции `joinToString`.

❻ Наконец, возвращаем заголовок и строки в виде `String`.

Теперь предположим, что мы получили дополнительный запрос на отчет только по атрибутам, которые являются критическими (`CRITICAL`) или важными (`IMPORTANT`). Нам просто нужно использовать функцию `filter`, как показано в примере 2.5.

Пример 2.5. Фильтрация критических и важных образцов

```
fun createCsv(timeSeries: List<TimeSeries>): String {
    /* Код удален для краткости */

    val pointsWithAttrs2 = timeSeries.filter {
        it.attr.tolerance == Tolerance.CRITICAL
                || it.attr.tolerance == Tolerance.IMPORTANT
    }.map { series ->
        series.points.map { point ->
            PointWithAttr(point, series.attr)
        }
    }.flatten()

    /* Код удален для краткости */
    return csvHeader + rows
}
```

Вот и все!

Чтобы протестировать этот код, можно использовать предопределенный ввод и проверить, соответствует ли вывод нашим ожиданиям. Мы не будем показывать здесь полноценный набор модульных тестов, а просто приведем пример вывода в формате CSV, как показано в примере 2.6.

Пример 2.6. Демонстрация приложения

```
fun main() {
    val dates = listOf<LocalDateTime>(
        LocalDateTime.parse("2020-07-27T15:15:00"),
        LocalDateTime.parse("2020-07-27T15:25:00"),
        LocalDateTime.parse("2020-07-27T15:35:00"),
        LocalDateTime.parse("2020-07-27T15:45:00")
    )
    val seriesExample = listOf(
        TimeSeries(
            points = listOf(
                Point("HC11", dates[3], 15.1),
                Point("HC12", dates[2], 15.05),
                Point("HC13", dates[1], 15.11),
                Point("HC14", dates[0], 15.08)
            ),
```

```
            attr = Attr("AngleOfAttack", Tolerance.CRITICAL)
        ),
        TimeSeries(
            points = listOf(
                Point("HC11", dates[3], 0.68),
                Point("HC12", dates[2], 0.7),
                Point("HC13", dates[1], 0.69),
                Point("HC14", dates[0], 0.71)
            ),
            attr = Attr("ChordLength", Tolerance.IMPORTANT)
        ),
        TimeSeries(
            points = listOf(
                Point("HC11", dates[3], 0x2196F3.toDouble()),
                Point("HC14", dates[0], 0x795548.toDouble())
            ),
            attr = Attr("PaintColor", Tolerance.REGULAR)
        )
    )
    val csv = createCsv(seriesExample)
    println(csv)
}
```

Если вы используете строку csv в качестве содержимого файла с расширением *csv*, то можете открыть его, используя свой любимый инструмент для работы с электронными таблицами. На рис. 2.2 показано, что именно мы получили с помощью FreeOffice.

Использование функционального программирования для преобразования данных, как в этом примере, особенно надежно. Почему? Сочетая null-безопасность и функции из стандартной библиотеки, можно создавать код, у которого мало побочных эффектов либо вообще их нет. Добавьте любой список PointWithAttr, который можно себе представить. Если хотя бы один экземпляр Point имеет значение null, код даже не будет компилироваться. Каждый раз, когда результат преобразования возвращает результат, который может быть нулевым, язык заставляет вас учитывать этот сценарий. Здесь мы сделали это на этапе 4 с помощью функции firstOrNull.

Всегда приятно, когда код компилируется и делает именно то, что ожидалось, с первой попытки. При использовании null-безопасности и функционального программирования такое случается часто.

Резюме

Будучи функциональным языком, Kotlin использует замечательные идеи, такие как отображение, архивирование и другие преобразования. Он даже позволяет создавать собственные преобразования данных, используя мощь функций высшего порядка и лямбда-выражений.

- Коллекции Kotlin включают весь API коллекций Java. Кроме того, библиотека предоставляет все распространенные преобразования, такие как отображение, фильтрация, группировка и многое другое.
- Kotlin поддерживает встроенные функции для более эффективного преобразования данных.
- Библиотека коллекций Kotlin поддерживает последовательности — способ работы с коллекциями, которые определяются намерением, а не расширением. Последовательности подходят, когда получение следующего элемента очень затратно, или даже для коллекций неограниченного размера.

Если вы когда-либо работали с такими языками, как Ruby, Scala или Python, то, возможно, что-то покажется вам знакомым. И это не удивительно! Архитектура Kotlin основана на многих принципах, которые лежали в основе разработки этих языков.

Написать более функциональный код для Android так же просто, как использовать операции преобразования данных, предлагаемые стандартной библиотекой Kotlin. Теперь, когда вы познакомились с синтаксисом Kotlin и прониклись духом функционального программирования, в следующей главе мы рассмотрим основы Android. Разворот в сторону Kotlin как официального языка для разработки приложений для Android произошел еще в 2017 г., поэтому Kotlin сильно повлиял на развитие этой операционной системы в последние годы. Так будет и в ближайшем будущем.

ГЛАВА 3
Основы Android

Первые две главы книги представляют собой беглый обзор языка Kotlin. В этой главе мы рассмотрим окружение, в котором будет использоваться Kotlin: Android.

Android — это операционная система, такая же, как Windows и MacOS. В отличие от них, она работает на ядре Linux, как Ubuntu и Red Hat, и в отличие от Ubuntu и Red Hat, Android очень сильно оптимизирована для работы на мобильных устройствах, в частности те из них, которые питаются от батареи.

Наиболее значительная из этих оптимизаций касается понятия "приложение". В частности, вы увидите, что у приложений для Android гораздо больше общего с веб-приложениями, нежели с привычными настольными приложениями.

Но мы еще вернемся к этому. Для начала подробнее изучим окружение Android. Мы будем рассматривать эту операционную систему как стек — что-то вроде слоеного пирога.

Стек Android

На рис. 3.1 показан один из способов представления Android: в виде набора компонентов. Каждый уровень в стеке имеет конкретную задачу и предоставляет определенные службы, и каждый из них использует возможности уровней, которые находятся ниже.

Пойдем снизу вверх:

- аппаратное обеспечение;
- ядро;
- системные службы;
- среда выполнения Android;
- приложения.

Аппаратное обеспечение

В нижней части стека Android, конечно же, находится аппаратное обеспечение. Хотя оно и не является частью стека, важно понимать, что аппаратное обеспечение,

для которого был разработан Android, накладывает на систему довольно жесткие ограничения. Безусловно, наиболее существенным из этих ограничений является мощность. В случае большинства распространенных операционных систем предполагается бесконечный источник питания. Android такого себе позволить не может.

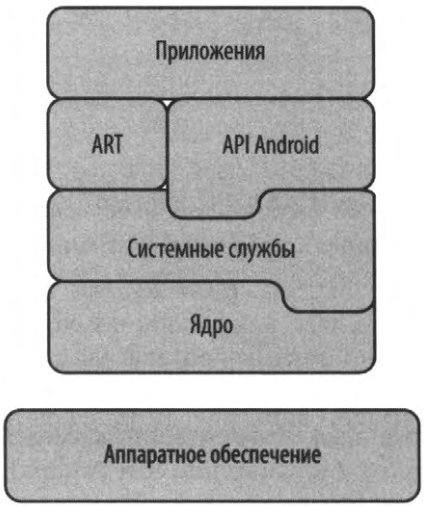


Рис. 3.1. Стек Android

Ядро

Android работает на базе ядра Linux. Ядро отвечает за предоставление основных служб, которые ожидают увидеть разработчики: файловая система, потоки и процессы, доступ к сети, интерфейсы для аппаратных устройств и т. д. Linux — это бесплатная операционная система с открытым исходным кодом, поэтому она пользуется популярностью среди производителей аппаратного обеспечения и устройств.

Поскольку операционная система Android работает на базе ядра Linux, у нее есть некоторое сходство с распространенными дистрибутивами Linux: Debian, Centos и т. д. Однако на уровнях выше ядра это сходство уменьшается. В то время как наиболее распространенные дистрибутивы Linux в значительной степени основаны на семействе системного программного обеспечения GNU (и должны называться GNU/Linux), программное обеспечение Android несколько иное. Как правило, обычные приложения Linux невозможно запускать непосредственно в системе Android.

Системные службы

Уровень системных служб большой и сложный. Он включает широкий спектр утилит, от кода, работающего как часть ядра (драйверы или модули ядра) и приложений с длительным временем запуска, управляющих различными служебными зада-

чами (демоны), до библиотек, реализующих стандартные функции, такие как криптография и медиапрезентация.

Данный уровень объединяет несколько системных служб, уникальных для Android. Среди них Binder — важная система взаимодействия процессов в Android; ART, который заменил Dalvik в качестве Android-аналога виртуальной машины Java; и Zygote — контейнер приложений для Android.

Среда выполнения Android

Уровень, располагающийся над системными службами, — это реализация *среды выполнения Android* (Android Runtime Environment, ART). ART — это набор библиотек, которые используются из приложения с помощью операторов `import`: *android.view*, *android.os* и т. д. Это службы, предоставляемые нижележащими уровнями, доступными для вашего приложения. Они интересны тем, что обычно реализованы на двух языках: Java и C или C++.

Часть реализации, которую импортирует приложение, скорее всего, будет написана на Java. Однако код Java использует механизм Java Native Interface (JNI) для вызова низкоуровневого кода, обычно написанного на языке C или C++. Это код, который фактически взаимодействует с системными службами.

Приложения

Наконец, на вершине стека находятся приложения. Приложения во вселенной Android на самом деле являются частью стека. Они состоят из индивидуально адресуемых компонентов, которые могут "вызывать" другие приложения. "Набор номера", "Камера" и "Контакты" — все это примеры приложений Android, которые используются другими приложениями в качестве служб. Это окружение, в котором выполняется приложение. Итак, вернемся к анатомии самого приложения.

Прикладное окружение Android

Приложения Android — это программы, переведенные с исходного языка (Java или Kotlin) на переносимый промежуточный язык DEX. Код DEX устанавливается на устройство и интерпретируется виртуальной машиной ART при запуске приложения.

Почти каждый разработчик знаком со стандартным прикладным окружением. Операционная система создает "процесс" — своего рода виртуальный компьютер, который полностью принадлежит приложению. Система запускает код приложения в процессе, где у него есть собственная память, собственные процессоры и т. д., полностью независимые от других приложений, которые могут работать на том же устройстве. Приложение работает до тех пор, пока само не решит остановиться.

В Android все по-другому. Эта система не мыслит категориями приложений. Например, приложения для Android не содержат эквивалент Java-метода `public static`

void main, используемого для запуска типичных приложений Java. Приложения для Android — это библиотеки компонентов. Среда выполнения Android, Zygote, управляет процессами, жизненными циклами и т. д. Она вызывает компоненты приложения только, когда они ей нужны. Как упоминалось ранее, по этой причине приложения для Android очень похожи на веб-приложения: они представляют собой набор компонентов, развернутых в контейнере.

Другой конец жизненного цикла — завершение работы приложения — возможно, даже более интересен. В иных операционных системах внезапная остановка приложения (kill -9 или "Принудительный выход") происходит редко и только тогда, когда приложение ведет себя неправильно. В Android это наиболее распространенный способ завершения работы приложения. Работа почти каждого приложения в конечном счете будет внезапно завершена.

Как и в большинстве фреймворков веб-приложений, компоненты реализованы как подклассы базовых классов шаблонов. Подклассы компонентов переопределяют методы, вызываемые фреймворком, чтобы обеспечить поведение, зависящее от приложения. Часто при вызове одного из этих методов у суперкласса есть важная работа. В таких случаях переопределяющий метод в подклассе должен вызывать переопределяемый им метод суперкласса.

Android поддерживает четыре типа компонентов:

- Activity;
- Service;
- BroadcastReceiver;
- ContentProvider.

Как и в веб-приложении, реализация этих компонентов должна быть зарегистрирована в манифесте: файле с расширением *xml*. Файл манифеста называется, что неудивительно, *AndroidManifest.xml*. Контейнер Android анализирует этот файл как часть загрузки приложения. Компоненты приложения (а не какое-то всеобъемлющее приложение) — это основные единицы приложения Android. Они индивидуально адресуемые, и их можно публиковать отдельно, чтобы их могли использовать другие приложения.

Итак, как же приложение выбирает компонент? С помощью объекта Intent.

Намерения и фильтры намерений

В Android компоненты запускаются с помощью объектов Intent (намерение). Intent — это небольшой пакет, определяющий имя компонента, на который он нацелен. У него есть дополнительное пространство, в котором он может указать конкретное действие (он хочет, чтобы компонент-приемник его выполнил), и несколько параметров запроса. Можно рассматривать намерение как вызов функции: имя класса, имя конкретной функции в этом классе и параметры вызова. Система дос-

тавляет намерение целевому компоненту. Компонент должен выполнить запрашиваемую услугу.

Интересно отметить, что в соответствии с компонентно-ориентированной архитектурой в Android фактически не существует способа запуска приложения. Клиенты запускают компонент, возможно, Activity (активности), зарегистрированный как основной для приложения, значка которого пользователь только что коснулся на странице пользовательского интерфейса. Если приложение, которому принадлежит активность, еще не работает, оно будет запущено в качестве побочного эффекта.

Intent может указать свою цель явно, как показано здесь:

```
context.startActivity(
    Intent(context, MembersListActivity::class.java)))
```

Этот код запускает Intent в MembersListActivity. Обратите внимание, что вызов, в данном случае это startActivity, должен быть согласован с типом запускаемого компонента: в данном случае Activity. Существуют и другие похожие методы запуска объектов Intent для иных типов компонентов (startService для Service и т. д.).

Объект Intent, инициированный этой строкой кода, называется явным, потому что определяет имя конкретного уникального класса в уникальном приложении (идентифицируемом Context, о котором мы скоро поговорим), которому должен быть доставлен.

Поскольку они идентифицируют конкретную уникальную цель, явные объекты Intent быстрее и безопаснее, чем неявные. Есть места, где система Android по причинам, связанным с безопасностью, требует использования явных объектов Intent. Даже если в них нет необходимости, по возможности следует отдавать предпочтение им.

В приложении к компоненту всегда можно получить доступ, используя явный объект Intent. К компоненту из другого общедоступного приложения также всегда можно получить доступ явно. Тогда зачем вообще использовать неявный вариант? По той причине, что неявные объекты Intent допускают динамическое разрешение запроса.

Представьте, что приложение электронной почты, которое вы использовали на своем телефоне на протяжении многих лет, позволяет редактировать сообщения с помощью внешнего редактора. Теперь можно предположить, что для этого оно запускает объект Intent, который может выглядеть примерно так:

```
val intent = Intent(Intent.ACTION_EDIT))
intent.setDataAndType(textToEditUri, textMimeType);
startActivityForResult(intent, reqId);
```

Цель, указанная здесь, *не* является явной. Объект Intent не указывает ни Context, ни полное имя компонента в контексте. Он является *неявным*, и Android позволит любому компоненту зарегистрироваться для его обработки.

Для получения неявных объектов Intent компоненты регистрируются с помощью IntentFilter. На самом деле приложение Awesome Code Editor, которое мы устано-

вили всего 15 минут назад, регистрируется для получения намерения, показанного в предыдущем коде, путем включения в манифест фильтра намерений, подобного этому:

```
<manifest ...>
  <application
    android:label="@string/awesome_code_editor">
    ...>
    <activity
      android:name=".EditorActivity"
      android:label="@string/editor">
      <intent-filter>
        <action
          android:name="android.intent.action.EDIT" />
        <category
          android:name="android.intent.category.TEXT" />
      </intent-filter>
    </activity>
  </application>
</manifest>
```

Как видно, фильтр намерений соответствует намерению, которое запускает приложение электронной почты.

При установке приложения Awesome Code Editor система Android анализирует манифест приложения и замечает, что `EditorActivity` утверждает, будто может обрабатывать действие `EDIT` для содержимого элемента `<category>` (для получения подробной информации обратитесь к документации для разработчиков Android[1]). Она запоминает этот факт.

В следующий раз, когда ваша почтовая программа запросит редактор, система Android включит приложение Awesome Code Editor в список редакторов, которые она предлагает вам использовать. Вы только что обновили программу для работы с электронной почтой, просто установив еще одно приложение.

Система Android постепенно увеличивала ограничения на использование неявных объектов `Intent` в последних выпусках. Поскольку они могут быть перехвачены любым случайно установленным приложением, несмотря на их мощь, такие объекты не являются безопасными. Последние версии Android наложили новые строгие ограничения на их использование. В частности, начиная с версии 30, для получения множества неявных объектов `Intent` регистрация в манифесте невозможна.

[1] См. **https://oreil.ly/oJNkY**.

Контекст

Поскольку компоненты Android — это просто подсистемы, работающие в крупном контейнере, им нужен какой-то способ обращения к контейнеру, чтобы они могли запрашивать у него службы. Из компонента контейнер воспринимается как контекст (Context). Контексты бывают двух видов: компонентные и прикладные. Рассмотрим каждый из них.

Контекст компонента

Мы уже видели такой вызов:

```
context.startActivity(
    Intent(context, MembersListActivity::class.java)))
```

Здесь Context используется дважды. Сначала запуск Activity — это функция, которую компонент запрашивает у фреймворка, Context. В данном случае был вызван метод startActivity. Затем, чтобы сделать объект Intent явным, компонент должен идентифицировать уникальный пакет, содержащий компонент, который он хочет запустить. Конструктор Intent использует context, переданный в качестве первого аргумента, чтобы получить уникальное имя для приложения, которому принадлежит context: данный вызов запускает экземпляр Activity, принадлежащий этому приложению.

Context — это абстрактный класс, предоставляющий доступ к различным ресурсам, в том числе:

- запуск других компонентов;
- доступ к системным службам;
- доступ к SharedPreferences, ресурсам и файлам.

Два компонента Android, Activity и Service, сами по себе являются контекстами. Помимо этого, они также являются компонентами, которыми контейнер Android рассчитывает управлять, что может привести к проблемам. Посмотрите на код, приведенный в примере 3.1.

Пример 3.1. Не делайте так!

```
class MainActivity : AppCompatActivity() {
    companion object {
        var context: Context? = null;
    }

    override fun onCreate() {
        if (context == null) {
            context = this // НЕТ!
        }
```

```
    }
    // ...
}
```

Наши разработчики решили, что было бы очень удобно иметь возможность писать что-то вроде `MainActivity.context.startActivity(...)` в любом месте приложения. Для этого они сохранили ссылку на экземпляр `Activity` в глобальной переменной, где она будет доступна на протяжении всего жизненного цикла приложения. Что может пойти не так?

Есть две вещи, которые могут пойти не так: одна плохая, а другая ужасная. Плохо, когда фреймворк Android знает, что `Activity` больше не нужен, и хотел бы высвободить его для сбора мусора, но не может этого сделать. Ссылка в этом объекте-компаньоне не даст освободить `Activity` на протяжении всего времени существования приложения. Происходит утечка. Объекты `Activity` не маленькие, и утечка памяти в данном случае — весомая проблема.

Вторая (гораздо худшая) вещь, которая может пойти не так, состоит в том, что вызов метода в кэшированной активности может привести к катастрофическому сбою. Как мы вскоре поясним, как только система Android решит, что экземпляр `Activity` больше не используется, она удаляет его. С ним покончено, и она никогда не будет использовать его снова. В результате объект может оказаться в несогласованном состоянии. Вызов методов для него может привести к сбоям, которые трудно диагностировать и воспроизвести.

Хотя проблему в этом фрагменте кода довольно легко увидеть, есть и гораздо более утонченные варианты. В следующем коде может быть аналогичная проблема:

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    // ...
    NetController.refresh(this::update)
}
```

Здесь это не так просто увидеть, но обратный вызов `this::update` — это ссылка на метод `this`, активность, которая содержит метод `onCreate`. После завершения работы метода `onCreate` `NetController` содержит ссылку на активность, которая не соблюдает жизненный цикл. Это может привести к любой из описанных ранее проблем.

Прикладной контекст

Существует еще один вид контекста. Когда Android запускает приложение, то обычно создает одиночный экземпляр класса `Application`. Этот экземпляр является контекстом, и, хотя у него есть жизненный цикл, по сути этот цикл совпадает с жизненным циклом приложения. Поскольку у него длительный жизненный цикл, вы можете вполне безопасно хранить ссылки на него в других местах с подобным циклом. Этот код, похожий на опасный код, что мы продемонстрировали ранее,

довольно безопасен, поскольку контекстом, ссылку на который он хранит, является ApplicationContext:

```
class SafeApp : Application() {
    companion object {
        var context: Context? = null;
    }

    override fun onCreate() {
        if (context == null) {
            context = this
        }
    }
    // ...
}
```

Не забудьте: для того, чтобы система Android могла использовать собственный подкласс Application вместо варианта по умолчанию, класс SafeApp должен быть зарегистрирован в манифесте. Например:

```
<manifest ...>
  <application
    android:name=".SafeApp"
    ...>
    ...
  </application>
</manifest>
```

Теперь, когда фреймворк создаст ApplicationContext, это будет экземпляр SafeApp, а не экземпляр Application, который использовался бы в противном случае.

Есть и другой способ получить ApplicationContext. Вызов метода Context.getApplicationContext() для любого контекста, включая сам ApplicationContext, всегда будет возвращать контекст приложения с длительным жизненным циклом. Но есть и плохие новости: ApplicationContext не панацея, поскольку он не является активностью. Его реализации методов Context ведут себя иначе, чем в случае с Activity. Например, и, вероятно, это больше всего раздражает, нельзя запускать Activity из ApplicationContext. В ApplicationContext есть метод startActivity, но он просто генерирует сообщение об ошибке при любых обстоятельствах, кроме ограниченных случаев.

Компоненты приложения Android: строительные блоки

Наконец, мы можем сосредоточиться на самих компонентах, которые и составляют суть приложения.

Жизненными циклами компонентов приложений Android управляет фреймворк Android, который создает и удаляет их в соответствии со своими потребностями.

Обратите внимание, что сюда непременно входит создание экземпляров! Код приложения *ни при каких обстоятельствах* не должен создавать новый экземпляр компонента.

Напомним, что существует четыре типа компонентов:

- `Activity`;
- `Service`;
- `Broadcast receiver`;
- `Content provider`.

Помните также, что следующие описания — это не что иное, как краткие обзоры, где, возможно, обращается внимание на потенциальные подводные камни или особенности, представляющие интерес. Существует обширная, полная и авторитетная документация для разработчиков приложений для Android[2].

Компонент *Activity* и его друзья

Компонент `Activity` управляет одной страницей пользовательского интерфейса приложения. Это аналог сервлета веб-приложения. Он использует богатую библиотеку виджетов для рисования одной интерактивной страницы. Виджеты (кнопки, текстовые поля и т. п.) представляют собой основные элементы пользовательского интерфейса и сочетают в себе экранное представление с исходной коллекцией, которая задает поведение виджетов. Мы обсудим их подробнее в ближайшее время.

Как упоминалось ранее, важно понимать, что `Activity` — это не приложение! Активности эфемерны и гарантированно существуют только до тех пор, пока страница, которой они управляют, видна. Когда эта страница становится невидимой либо потому, что приложение открывает другую страницу, либо потому, что пользователь, например, отвечает на телефонный звонок, нет никакой гарантии, что Android сохранит экземпляр `Activity` или любое состояние, которое этот компонент представляет.

На рис. 3.2 показана диаграмма состояний, управляющих жизненным циклом `Activity`. Методы, показанные как переходы между состояниями, идут попарно и представляют собой основы четырех состояний, которые может принимать экземпляр `Activity`: *удален*, *создан*, *запущен* и *выполняется*. Методы вызываются строго по порядку. Например, после вызова метода `onStart` Android вызовет только один из двух возможных методов: `onResume`, чтобы перейти в следующее состояние, или `onStop`, чтобы вернуться в предыдущее.

Первая пара— это методы `onCreate` и `onDestroy`. Говорят, что между ними *создается* экземпляр `Activity`. При создании нового экземпляра `Activity` почти сразу же вызывается метод `onCreate`. До тех пор, пока это не произойдет, `Activity` находится в несогласованном состоянии, и большинство функций этого компонента не будут ра-

[2] См. **https://oreil.ly/PJABc**.

ботать. Отметим, в частности, что бóльшая часть возможностей Activity недоступна из его конструктора, что неудобно.

Рис. 3.2. Жизненный цикл Activity

Метод onCreate — идеальное место для любой инициализации, которую экземпляр Activity должен выполнить только один раз. Сюда почти всегда входят настройка иерархии представлений (обычно путем расширения макета в формате XML), установка контроллеров представлений или презентаторов, а также подключение обработчиков событий касания и ввода текста.

Точно так же экземпляры Activity не следует использовать после вызова метода onDestroy. Activity снова находится в несогласованном состоянии, и фреймворк Android больше не будет его использовать. (Например, не будет вызывать метод onCreate, чтобы оживить его.) Однако будьте осторожны: метод onDestroy не обязательно является лучшим местом для выполнения важного завершения! Android вызывает его только в крайнем случае. Вполне возможно, что работа приложения будет завершена до того, как будет закончена работа всех методов onDestroy.

Экземпляр Activity можно удалить из его собственной программы, вызвав метод finish().

Следующая пара методов — onStart и onStop. Первый метод, onStart, будет вызываться только для экземпляра Activity, находящегося в состоянии "создан". Он переводит его в состояние готовности, которое называется *"запущен"*. Запускаемый экземпляр Activity можно частично увидеть за диалоговым окном или другим приложением, которое лишь частично занимает экран. В этом состоянии экземпляр Activity должен быть полностью окрашен, но не должен ожидать ввода пользователя. Правильно написанный экземпляр Activity не будет запускать анимацию или другие ресурсоемкие задачи, пока находится в состоянии "запущен".

Метод onStop будет вызываться только для запущенного экземпляра Activity. Он возвращает его в состояние "создан".

Последняя пара методов — это onResume и onPause. Между ними страница Activity находится в центре внимания устройства и является целью пользовательского ввода. Говорят, что он *выполняется*. Эти методы будут вызываться только для экземпляра Activity, который находится в состоянии "запущен" или "выполняется".

Наряду с onCreate, методы onResume и onPause являются наиболее важными в жизненном цикле Activity. Это место, где страница оживает, запуская, скажем, обновление данных, анимацию и все другие вещи, которые делают пользовательский интерфейс отзывчивым.

Рекомендуется соблюдать сочетание этих методов: начального и конечного. Если вы запускаете что-то в начальном методе пары, останавливайте это в конечном методе той же пары. Попытка запустить, скажем, опрос сети в методе onResume и остановить его в методе onStop приведет к ошибкам, которые трудно будет обнаружить.

Фрагменты

Объекты Fragment — это дополнение, добавленное в стабильный релиз компонентоподобных функциональных возможностей Android только в версии 3 (Honeycomb, 2011). Они появились как способ сделать возможным совместное использование реализаций пользовательского интерфейса на экранах с настолько разными формами и размерами, что это влияет на навигацию: в частности, в телефонах и планшетах.

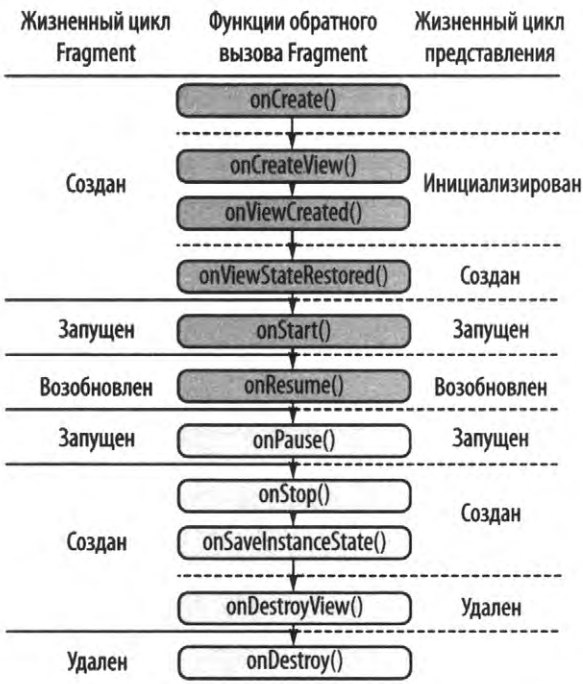


Рис. 3.3. Жизненный цикл Fragment

Объекты Fragment — это не контексты. Хотя они и содержат ссылку на базовую активность на протяжении большей части своего жизненного цикла, их не регистрируют в манифесте. Они создаются в коде приложения, и их нельзя запускать с помощью объектов Intent. Кроме того, они достаточно сложны. Сравните диаграмму

состояний Fragment на рис. 3.3 с диаграммой Activity! Подробное обсуждение того, как (или, если на то пошло, даже нужно ли) использовать эти объекты, выходит за рамки данной книги. Однако, если говорить в двух словах, можно рассматривать объект Fragment как элемент *iframe* на веб-странице: почти Activity, встроенный в Activity. Это полноценные логические единицы пользовательского интерфейса, которые можно собирать различными способами для формирования страницы.

Как показано, жизненные циклы объектов Fragment аналогичны (хотя и сложнее) жизненным циклам Activity. Однако Fragment полезен только, когда он привязан к Activity. Это основная причина, по которой его жизненный цикл более сложный: на его состояние могут влиять изменения состояния Activity, с которой он связан.

Кроме того, как экземпляр Activity доступен на программном уровне в несогласованном состоянии до вызова метода onCreate, так же и Fragment доступен на программном уровне, прежде чем будет привязан к Activity. Объекты Fragment следует использовать с большой осторожностью, прежде чем будут вызваны методы onAttach и onCreateView.

Стек переходов назад

Android поддерживает парадигму, которую иногда называют навигацией по принципу *карточной колоды*. При переходе на новую страницу она размещается поверх предыдущей. Когда пользователь нажимает кнопку **Назад**, текущая страница выталкивается из стека, чтобы показать ту, которая была на экране ранее. Эта парадигма интуитивно понятна довольно большому количеству пользователей: нажимайте на новые карты сверху и выталкивайте их, чтобы вернуться туда, где вы находились.

На рис. 3.4 текущая активность называется SecondActivity. Нажатие кнопки **Назад** приведет к тому, что активность MainActivity займет экран.

Обратите внимание, что, в отличие от веб-браузера, в Android не поддерживается навигация с переходом вперед. Как только пользователь нажимает кнопку **Назад**, он не может просто вернуться на всплывающую страницу. Система Android считает: чтобы организовать вывод, можно удалить SecondActivity (в данном случае), если ей потребуются ресурсы.

Рис. 3.4. В стеке переходов назад страницы Activity хранятся согласно принципу "последним пришел — первым ушел" (LIFO)

Объекты Fragment также могут помещаться в стек переходов назад как часть транзакции (рис. 3.5).

Рис. 3.5. Транзакция экземпляра `Fragment` в стеке переходов назад будет отменена до того, как содержащая его активность будет извлечена

Добавление экземпляра `Fragment` в стек переходов назад может быть особенно полезным в сочетании с тегированием, как показано в следующем коде:

```
// Добавляем фрагмент новой вкладки
supportFragmentManager.beginTransaction()
    .replace(
        R.id.fragment_container,
        SomeFragment.newInstance())
    .addToBackStack(FRAGMENT_TAG)
    .commit()
```

Этот код создает новый экземпляр `SomeFragment` и добавляет его в стек переходов назад с идентификатором `FRAGMENT_TAG` (строковая константа). Как показано в следующем коде, можно использовать `supportFragmentManager`, чтобы извлечь *все* из стека, вплоть до тега:

```
manager.popBackStack(
    FRAGMENT_TAG,
    FragmentManager.POP_BACK_STACK_INCLUSIVE)
```

Если стек переходов назад пуст, то при нажатии кнопки **Назад** пользователь возвращается на страницу пользовательского интерфейса.

Службы

`Service` — это компонент Android, который почти точно представляет собой `Activity` без пользовательского интерфейса. Это может показаться немного странным, учитывая, что единственной причиной существования `Activity` является тот факт, что этот компонент управляет пользовательским интерфейсом!

Система Android была разработана для аппаратного обеспечения, которое сильно отличается от того, что распространено сейчас. Выход первого телефона, работающего под управлением Android, HTC Dream, был анонсирован в сентябре 2008 г. У него было очень мало физической памяти (192 Мбайт), он вообще не поддерживал виртуальную память и мог запускать лишь небольшое количество приложений одновременно.

Разработчикам приложений для Android нужен был способ узнать, когда приложение не выполняет полезную работу, чтобы можно было использовать его память для других целей.

Легко понять, когда Activity не выполняет полезную работу. У этого компонента только одна задача — управлять видимой страницей. Если бы приложения состояли только из экземпляров Activity, было бы легко определить, когда один из них больше не является полезным и его работу можно завершить. Когда ни один из них не отображается, приложение не делает ничего полезного, и его можно удалить. Это так просто.

Проблема возникает, когда приложению необходимо выполнять длительные задачи, не привязанные к какому-либо пользовательскому интерфейсу: мониторинг местоположения, синхронизация набора данных по сети и т. д. В то время как система Android определенно предвзято относится к тому, что "если пользователь этого не видит, зачем это делать?", она неохотно признает существование длительных задач и придумала службы для их обработки.

Хотя службы по-прежнему используются, бóльшую часть работы, для которой они были созданы в более ранних версиях Android с более ограниченным аппаратным обеспечением, теперь можно выполнять с использованием других методов. WorkManager — отличный способ управлять повторяющимися задачами. Существуют и другие, более простые и удобные в сопровождении способы запуска задач в фоновом режиме, в рабочем потоке. Достаточно чего-то простого, например класса-"одиночки".

Однако службы по-прежнему существуют и играют важную роль.

На самом деле есть два разных вида компонента Service: *подключаемая* служба и *запускаемая*. Несмотря на то что базовый класс Service, как ни странно, является шаблоном для обоих, эти два вида полностью ортогональны. Отдельная служба может быть либо запускаемой, либо подключаемой, либо и тем и другим.

У обоих видов Service есть методы onCreate и onDestroy, которые ведут себя точно так же, как и в случае с Activity. Поскольку у Service нет пользовательского интерфейса, ему не нужны иные шаблонные методы Activity.

Однако у служб есть и другие шаблонные методы. Какие из них реализует конкретная служба, зависит от того, является ли она запускаемой или подключаемой.

Запускаемые службы

Запускаемая служба инициируется путем отправки ей объекта Intent. Хотя можно создать запускаемую службу, которая возвращает значение, это некрасиво и сложно и может указывать на то, что сюда можно было бы внести улучшения. По большей части запускаемые службы работают по принципу "выстрелил и забыл": что-то вроде "поместите это в базу данных" или "отправьте это в сеть".

Чтобы запустить службу, отправьте ей объект Intent. Он должен определить ее имя, возможно, явно, передав текущий контекст и класс службы. Конечно, если служба

предоставляет несколько функций, то объект `Intent` может также указывать, для вызова какой из них он предназначен. Кроме того, он может предоставлять параметры, подходящие для вызова.

Служба получает объект `Intent` в качестве аргумента вызова от фреймворка Android для метода `Service.onStart`. Обратите внимание, что это происходит не в "фоновом режиме"! Метод `onStart` выполняется в основном или UI-потоке. Он анализирует содержимое `Intent` и соответствующим образом обрабатывает запрос.

Правильно функционирующая запускаемая служба будет вызывать метод `Service.stopSelf()` всякий раз при завершении работы. Этот вызов похож на метод `Activity.finish()`: он сообщает, что экземпляр службы больше не выполняет полезную работу и его можно удалить.

На самом деле современные версии Android не особо обращают внимание на то, сама ли остановилась служба или нет. Выполнение служб приостанавливается и, возможно, даже прекращается при менее добровольных критериях (см. документацию для разработчиков Android[3]).

Подключаемые службы

Подключаемая служба — это механизм взаимодействия процессов (interprocess communication, IPC) в Android. Подключаемые службы обеспечивают канал обмена данными между клиентом и сервером, который не зависит от процесса: это приложение с двумя концами. Подключаемые службы или, по крайней мере, каналы обмена данными, которые они предоставляют, лежат в основе Android. Это механизм, с помощью которого приложения отправляют задачи системным службам.

Подключаемая служба сама по себе мало что делает. Это всего лишь *фабрика* для Binder, полудуплексного IPC-канала. Хотя полное описание IPC-каналов Binder и их использования выходит за рамки данной книги, их структура будет знакома пользователям других распространенных механизмов взаимодействия процессов. Эта система показана на рис. 3.6.

Обычно служба предоставляет *прокси-объект*, который выглядит как простой вызов функции. Он осуществляет *маршалинг* идентификатора запрашиваемой службы (по сути, это имя функции) и ее параметров путем преобразования их в данные, которые можно передавать по соединению: обычно это агрегаты очень простых типов данных, таких как целые числа и строки. Данные, прошедшие маршалинг, передаются, в данном случае через модуль ядра Binder, *объекту-заглушке*, предоставляемому подключаемой службой, что и является целью подключения.

Объект-заглушка выполняет *демаршалинг* данных, преобразовывая их обратно в вызов функции для реализации службы. Обратите внимание, что функция прокси и функция реализации службы имеют одинаковую сигнатуру: они реализуют один и тот же интерфейс (IService, как показано на рис. 3.6).

---

[3] См. **https://oreil.ly/yGloh**.

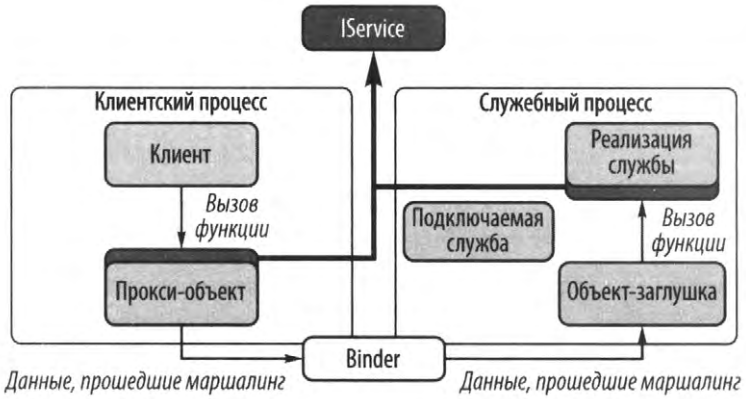


Рис. 3.6. Binder IPC

В Android этот механизм *широко* используется при реализации системных служб. Функции, которые на самом деле представляют собой вызовы удаленных процессов, являются фундаментальной частью этой операционной системы. Экземпляр класса ServiceConnection обозначает соединение с подключаемой службой. Следующий код демонстрирует его использование:

```
abstract class BoundService<T : Service> : ServiceConnection {
    abstract class LocalBinder<out T : Service> : Binder() {
        abstract val service: T?
    }

    private var service: T? = null

    protected abstract val intent: Intent?

    fun bind(ctxt: Context) {
        ctxt.bindService(intent, this, Context.BIND_AUTO_CREATE)
    }

    fun unbind(ctxt: Context) {
        service = null
        ctxt.unbindService(this)
    }

    override fun onServiceConnected(name: ComponentName, binder: IBinder) {
        service = (binder as? LocalBinder<T>)?.service
        Log.d("BS", "bound: ${service}")
    }
```

```
    override fun onServiceDisconnected(name: ComponentName) {
        service = null
    }
}
```

Подкласс `BoundService` предоставляет тип службы, с которой будет установлена связь, и предназначенный для нее объект `Intent`.

На стороне клиента инициируется подключение с использованием вызова функции `bind`. В ответ фреймворк инициирует подключение к удаленному объекту подключаемой службы. Удаленный фреймворк вызывает метод `onBind` подключаемой службы с объектом `Intent`. Подключаемая служба создает и возвращает реализацию `IBinder`, которая также является реализацией интерфейса, запрашиваемого клиентом. Обратите внимание, что зачастую это ссылка на саму подключаемую службу. Другими словами, часто `Service` — это не только фабрика, но и реализация.

На стороне службы используется реализация, предоставляемая подключаемой службой, для создания заглушки на удаленной стороне. После этого сторона службы уведомляет клиентскую сторону о том, что все готово. Фреймворк на стороне клиента создает прокси-объект, а затем, наконец, вызывает метод `onServiceConnected` класса `ServiceConnection`. Теперь у клиента есть активное подключение к удаленной службе. Выгода налицо!

Как можно догадаться по наличию метода `onServiceDisconnected`, клиент в любой момент может потерять соединение с подключаемой службой. Хотя обычно соответствующее уведомление приходит незамедлительно, вполне возможно, что вызов службы от клиента завершится ошибкой еще до того, как будет получено уведомление о разрыве соединения.

Как и в случае с запускаемой службой, код подключаемой службы не работает в фоновом режиме. Если явно не указано иное, то ее код выполняется в основном потоке приложения. Однако это может сбивать с толку, поскольку подключаемая служба может работать в основном потоке *другого* приложения.

Если код в реализации службы должен выполняться в фоновом потоке, то за это отвечает именно реализация службы. Будучи асинхронными, вызовы подключаемой службы от клиента не могут управлять потоком, в котором работает сама служба.

Службы, как и любой другой компонент, должны быть зарегистрированы в манифесте приложения:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application...>
        <service android:name=".PollService"/>
    </application>
</manifest>
```

Провайдеры контента

ContentProvider — это REST-подобный интерфейс для данных, находящихся в приложении. Поскольку это API, а не просто прямой доступ к данным, ContentProvider может осуществлять очень тонкий контроль того, что он публикует и для кого он это делает. Внешние приложения получают доступ к ContentProvider с помощью интерфейса Binder IPC, через который ContentProvider может получать информацию о процессе запроса, имеющихся у него правах доступа и виде запрашиваемого доступа.

Раньше приложения для Android часто делились данными, просто помещая их в общедоступные файлы. И даже тогда система Android поощряла использование провайдеров контента.

В интересах безопасности в более поздних версиях Android обмениваться файлами напрямую стало сложнее, что сделало провайдеров контента более актуальными.

Android Jetpack

Хотя провайдеры контента предоставляют доступ к хранимым данным, у вас должно быть некое хранилище, из которого можно читать и в которое можно записывать данные. В качестве варианта Android Jetpack предлагает библиотеку Room. Как написано в официальной документации, Room предоставляет "уровень абстракции, обеспечивающий более надежный доступ, используя всю мощь SQLite".

Для получения дополнительной информации о том, как сохранять данные в локальной базе данных с помощью Room, обратитесь к документации для разработчиков Android[4].

У ContentProvider есть одна особенно интересная способность. Она заключается в том, что он может передавать открытый файл другой программе. Программе, выполняющей запрос, не нужно иметь способ доступа к файлу напрямую, используя путь к файлу. ContentProvider может создавать файл, который он передает, любым удобным для него способом. Однако, передавая открытый файл, он выходит из цикла, что дает программе, выполняющей запрос, прямой доступ к данным.

Между клиентом и данными не остается ни провайдера контента, ни какого-либо иного механизма взаимодействия процессов. Клиент просто читает файл так, как если бы сам открыл его.

Как обычно, приложение публикует ContentProvider, объявляя его в манифесте:

```
<application...>
  <provider
    android:name="com.oreilly.kotlin.example.MemberProvider"
    android:authorities="com.oreilly.kotlin.example.members"
```

[4] См. **https://oreil.ly/9OwGH**.

```
    android:readPermission="com.oreilly.kotlin.example.members.READ"/>
</application>
```

Здесь мы сообщаем, что приложение содержит класс `com.oreilly.kotlin.example.MemberProvider`, который должен быть подклассом `android.content.ContentProvider`. Далее идет описание URL-адреса **content://com.oreilly.kotlin.example.members**. Наконец, объявление требует, чтобы приложения, выполняющие запрос, имели права `com.oreilly.kotlin.example.members.READ`, чтобы получить доступ, и что даже в этом случае они получат доступ только для чтения.

У провайдеров контента именно тот API, который можно было бы ожидать от REST-интерфейса.

`query()`

Извлекает данные из определенной таблицы.

`insert()`

Вставляет новую строку в провайдере контента и возвращает URI контента.

`update()`

Обновляет поля существующей строки и возвращает количество обновленных строк.

`delete()`

Удаляет существующие строки и возвращает количество удаленных строк.

`getType()`

Возвращает тип данных MIME для данного URI контента.

`ContentProvider` для `MemberProvider`, вероятно, реализует лишь первый из этих методов, потому что предназначен только для чтения.

BroadcastReceiver

Первоначально компонент `BroadcastReceiver` был своего рода шиной данных. Слушатели могли подписываться, чтобы получать уведомления о событиях, представляющих интерес. Однако, по мере того как система достигала зрелости, экземпляры `BroadcastReceiver` оказались слишком затратными и были слишком подвержены проблемам, связанным с безопасностью, чтобы их можно было использовать повсеместно. В основном они остаются инструментом, используемым системой, чтобы отправлять приложениям сигналы о важных событиях.

Возможно, наиболее распространенный вариант использования `BroadcastReceiver` — это способ запуска приложения, даже если от пользователя не поступало запроса на это.

Содержимое элемента `<action>`, `android.intent.action.BOOT_COMPLETED`, рассылается Android, как только операционная система становится стабильной после перезагрузки.

Для получения этого сообщения приложение можно зарегистрировать, например:

```
<receiver android:name=".StartupReceiver">
    <intent-filter>
        <action android:name="android.intent.action.BOOT_COMPLETED"/>
    </intent-filter>
</receiver>
```

Если приложение сделает это, то будет запущен `StartupReceiver` для получения сообщения при перезагрузке системы. Как отмечалось ранее, побочный эффект запуска `StartupReceiver` состоит в том, что приложение, содержащее приемник, также запускается.

Приложения использовали это как способ создания *демона* — приложения, которое всегда работает. Будучи несовершенным и ненадежным (даже в ранних версиях Android поведение менялось от версии к версии), этот трюк работал достаточно хорошо, и многие приложения использовали его. Несмотря на то что в версию Android 26 были внесены некоторые довольно радикальные изменения в управлении фоновыми процессами (`BroadcastReceiver` нельзя зарегистрировать для неявных сообщений в манифестах; его нужно регистрировать динамически, используя `Context.registerReceiver`), разработчики продолжают искать способы использовать его.

Для Android 26 есть исключения из правила неявного объекта `Intent`. Получение SMS-сообщений, изменение локали, обнаружение USB-устройств и некоторые другие намерения *разрешены*, и приложения можно регистрировать в манифестах для их получения. `ACTION_USB_ACCESSORY_ATTACHED`, `ACTION_CONNECTION_STATE_CHANGED` и наш дорогой старый друг `ACTION_BOOT_COMPLETED` входят в число разрешенных намерений. Для получения дополнительной информации ознакомьтесь с документацией для разработчиков Android[5].

`Activity`, `Service`, `ContentProvider` и `BroadcastReceiver` — это четыре компонента, которые являются основными строительными блоками приложения Android. По мере того как система Android развивалась и совершенствовалась, в ней появились новые абстракции, скрывающие эти базовые механизмы. Современное приложение для Android может напрямую использовать только один или два таких строительных блока, а многие разработчики никогда не будут писать `ContentProvider` или `BroadcastReceiver`.

Важный урок, который стоит повторить, заключается в том, что приложение для Android не является "приложением" в традиционном смысле этого слова. Оно больше похоже на веб-приложение: это набор компонентов, которые предоставляют услуги фреймворку по запросу.

---

[5] См. **https://oreil.ly/PMNhM**.

Архитектуры приложений Android

До сих пор в этой главе мы обсуждали архитектуру системы Android. Понимания того, что архитектура важна для любого серьезного разработчика Android, недостаточно, чтобы знать, как писать отказоустойчивые программы без ошибок. В качестве доказательства достаточно взглянуть на множество инструментов и абстракций, опробованных и заброшенных за годы существования Android. Тем не менее время и опыт отточили тактику Android и значительно упростили путь к созданию надежного и удобного в сопровождении приложения.

MVC: основы

Первоначально паттерн для приложений с пользовательским интерфейсом назывался Model-View-Controller (MVC — Модель-Представление-Контроллер). Нововведение, предложенное этим паттерном, было гарантией того, что представление — то, что отображалось на экране — всегда было согласованным. Здесь используется однонаправленный поток данных.

Все начинается с пользователей. Они что-то видят на экране (*представление*: "Я же говорило, это был цикл!") и в ответ на увиденное предпринимают какие-то действия. Они касаются экрана, что-то печатают, говорят, что угодно. Они делают нечто, что изменяет состояние приложения.

Пользовательский ввод обрабатывается *контроллером*. У контроллера две обязанности. Во-первых, он упорядочивает ввод. Для любого конкретного пользовательского события — скажем, нажатия кнопки **Стоп** — все другие пользовательские события происходят либо до этого нажатия, либо после него. Два события никогда не обрабатываются одновременно.

Тот факт, что контроллер однопоточный, является одним из наиболее важных аспектов исходного паттерна MVC. Предыдущие многопоточные стратегии (включая библиотеку Abstract Window Toolkit [AWT] языка Java) часто приводили к кошмару, порождаемому взаимными блокировками, поскольку сообщения, идущие в противоположных направлениях — от пользователя к пользователю, пытались захватить одни и те же мьютексы в разном порядке.

Второй обязанностью контроллера является преобразование пользовательского ввода в операции с *моделью*.

Модель — это бизнес-логика приложения. Вероятно, она сочетает в себе какое-то хранилище хранимых данных и, возможно, подключение по сети с правилами для объединения и интерпретации ввода из контроллера. В идеальной архитектуре MVC это единственный компонент, в котором хранится текущее состояние приложения.

Опять же, в идеале модель может отправить представлению только одно сообщение: "произошли изменения". Когда представление получает подобное сообщение, оно выполняет свою работу и запрашивает состояние приложения у модели, интерпретирует его и отображает на экране. То, что оно отображает, всегда представляет собой согласованный снимком модели. В этот момент пользователь может увидеть новое состояние и предпринять в ответ новые действия. Цикл продолжается.

Хотя паттерн MVC был довольно революционным в момент своего появления, его можно улучшить.

Виджеты

Как уже упоминалось ранее в контексте компонента `Activity`, виджет — это отдельный класс, объединяющий представление и контроллер. После предыдущего обсуждения паттерна MVC и акцента на его разделении, возможно, покажется странным обнаружить такие классы, как `Button`, `TextBox` и `RadioButton`, которые явно сочетают их.

Виджеты не нарушают архитектуру MVC. В каждом виджете по-прежнему есть отдельный код представления и контроллера. Часть, относящаяся к контроллеру, никогда не общается напрямую с представлением, а представление не получает события от контроллера. Эти разделы независимы; они просто объединены в один удобный контейнер.

Объединение двух функций кажется довольно очевидным. Что толку от изображения кнопки, которую можно разместить в любом месте экрана, если она не реагирует на нажатие? Было бы полезно, если бы средство отображения компонентов пользовательского интерфейса и механизм, обрабатывающий ввод для него, были частью одного и того же компонента.

Локальная модель

С появлением Интернета, браузеров и длительной задержки, нужной для всего цикла MVC, разработчики начали понимать необходимость сохранения состояния экрана в виде отдельной локальной модели пользовательского интерфейса. Со временем они по-разному стали называть этот компонент, в зависимости от других особенностей паттерна проектирования, в котором он используется. Во избежание путаницы в оставшейся части этой главы мы будем называть его *локальной моделью.*

Использование локальной модели порождает новый паттерн, который представляет собой своего рода двухуровневый MVC — его даже называют "Восьмерка". Когда пользователь предпринимает действие, контроллер обновляет локальную модель вместо собственно модели, потому что обновление модели может происходить через подключение по сети. Локальная модель — это не бизнес-логика. Она максимально просто отражает состояние представления: какие кнопки включены, какие выключены, какой текст в каком поле находится, а также цвет и длину столбцов на графике.

В ответ на действие локальная модель делает две вещи. Сначала она уведомляет представление о том, что все изменилось, чтобы представление могло повторно отобразить экран из нового состояния локальной модели. Кроме того, с помощью кода, аналогичного простому контроллеру MVC, локальная модель пересылает изменения состояния модели. В ответ модель в конце концов уведомляет — на этот раз локальную модель — о том, что произошло обновление и что локальная модель должна синхронизироваться. Вероятно, это приводит ко второму запросу на обновление самого представления.

Паттерны Android

В Android, независимо от паттерна, объект `Activity` — или, возможно, его двоюродный брат `Fragment` — берет на себя роль представления. Это более или менее предписывается структурой объекта `Activity`: это то, что владеет экраном и имеет доступ к виджетам, образующим представление. Однако со временем, как и подобает пользовательскому интерфейсу на базе MVC, объекты `Activity` становятся все проще и проще. В современном Android-приложении `Activity`, скорее всего, будет делать немногим больше, нежели расширять представление, делегировать события, поступающие от пользователя, в локальную модель и наблюдать за представляющим интерес состоянием этой модели, перерисовывая себя в ответ на обновление.

Model-View-Intent

Одна из самых старых версий MVC, принятая сообществом Android, называлась Model-View-Intent (MVI). Этот паттерн отделяет `Activity` от модели с помощью намерений и их данных. Хотя эта структура обеспечивает превосходную изоляцию компонентов, она может быть довольно медленной, а код для создания намерений довольно громоздким. Хотя она по-прежнему успешно используется, новые модели в значительной степени вытеснили ее.

Model-View-Presenter

Целью всех этих паттернов на базе MVC является ослабление связанности между тремя компонентами и обеспечение однонаправленного потока информации. Однако в простой реализации представление и локальная модель содержат ссылки друг на друга. Возможно, представление получает экземпляр локальной модели от какой-то фабрики, а затем регистрируется в ней. Хотя это и незаметно и не зависит от очевидного направления потока информации, наличие ссылки на объект определенного типа и является связанностью.

За последние несколько лет в паттерн MVC был внесен ряд усовершенствований в попытке уменьшить эту связанность. Хотя эти усовершенствования часто приводили к лучшему коду, различия между ними и сами имена, используемые для их идентификации, не всегда были ясны.

Одно из самых ранних усовершенствований заменяет ссылки друг на друга в представлении и локальной модели ссылками на интерфейсы. Этот паттерн часто называют Model-View-Presenter (MVP). В его реализациях локальная модель содержит ссылку не на `Activity` представления, а просто на реализацию некоего интерфейса. Интерфейс описывает минимальный набор операций, которые локальная модель может ожидать от своего "партнера". По сути, он не знает, что представление — это представление; единственное, что он видит, — это операции по обновлению информации.

Представление перенаправляет события пользовательского ввода своему презентатору. Презентатор, как описано ранее, реагирует на события, при необходимости обновляя состояние локальной модели и модели. Затем он уведомляет представление о необходимости перерисовки. Поскольку презентатор точно знает, какие именно изменения произошли, вероятно, он может запросить, чтобы представление обновляло только затронутые разделы, вместо того чтобы принудительно перерисовывать весь экран.

Однако самым важным атрибутом этой архитектуры является тот факт, что презентатор (название локальной модели в этой архитектуре) можно протестировать, используя модульные тесты. Тестам нужно только сымитировать интерфейс, который представление предоставляет презентатору, чтобы полностью изолировать его от представления. Чрезвычайно тонкие представления и тестируемые презентаторы позволяют создавать гораздо более надежные приложения.

Но можно сделать еще лучше. Локальная модель может вообще не содержать ссылок на представление!

Model-View-ViewModel

С появлением Jetpack компания Google поддерживает архитектуру Model-View-ViewModel (MVVM). Поскольку на внутреннем уровне она поддерживается современной платформой Android, это наиболее распространенный и наиболее обсуждаемый паттерн для современных приложений Android.

В этом паттерне, как обычно, либо `Activity`, либо `Fragment` берут на себя роль представления. Код представления будет настолько простым, насколько это возможно, часто он будет полностью находиться в подклассе `Activity` или `Fragment`. Возможно, для некоторых сложных представлений потребуются отдельные классы для отображения изображений или `RecyclerView`. Однако даже они будут создаваться и устанавливаться в представлении непосредственно `Activity` или `Fragment`.

ViewModel отвечает за объединение команд, необходимых для обновления представления и внутренней модели. Новая особенность этого паттерна заключается в том, что для передачи изменений состояния локальной модели в представление используется единый интерфейс `Observable`.

Вместо нескольких интерфейсов презентатора, используемых в шаблоне MVP, ViewModel представляет просматриваемые данные в виде коллекции объектов

Observable. Представление просто регистрируется в качестве наблюдателя и реагирует на уведомления об изменениях в данных, которые они содержат.

Библиотека Jetpack называет эти объекты LiveData. Объект LiveData — это класс, который хранит данные и за поведением которого можно наблюдать. У него имеется единый обобщенный интерфейс, уведомляющий подписчиков об изменениях в базовых данных.

Подобно MVP, MVVM упрощает создание объектов-имитаций и модульное тестирование. Он вводит важную новую функцию — учет жизненного цикла.

Внимательный читатель заметит, что версия паттерна MVP, описанная ранее, делает именно то, о чем мы предупреждали в примере 3.1: она хранит ссылку на Activity, объект с жизненным циклом, контролируемым Android, в длительном объекте!

Приложения предоставлены сами себе, чтобы убедиться, что ссылка не будет существовать дольше, чем целевой объект.

Реализация паттерна MVVM, поддерживаемая Jetpack, значительно уменьшает эту проблему. В этой реализации единственными ссылками на представление являются подписки на объекты LiveData. Эти объекты идентифицируют наблюдателей Fragment и Activity и автоматически отменяют их регистрацию по окончании их жизненного цикла.

Приложения, созданные с помощью версии паттерна MVVM от JetPack, могут быть довольно элегантными. Для широкого спектра приложений представление будет содержать один простой декларативный метод, рисующий экран. Он регистрирует этот метод в качестве наблюдателя для объектов Observable от ViewModel. ViewModel преобразует пользовательский ввод в вызовы внутренней модели и обновляет объекты Observable в ответ на уведомления от модели. Это так просто.

Резюме

Поздравляем, вы успешно осилили пугающий объем информации в очень короткой главе!

Помните, что бóльшая часть этого материала является основополагающей. Неважно, усвоите ли вы всю представленную здесь информацию. На самом деле, вполне возможно, что вы никогда не соприкоснетесь, например, с ContentProvider или BroadcastReceiver. Используйте то, что является практичным для вас, и подходите к освоению элементов только тогда, когда они становятся полезными.

Вот несколько ключевых моментов, которые нужно учитывать.

- Приложение для Android не является "приложением" в традиционном смысле этого слова. Оно больше похоже на веб-приложение: это набор компонентов, которые предоставляют службы фреймворку по запросу.
- Android — это очень специализированный дистрибутив на базе ядра Linux. Каждое приложение рассматривается как отдельный "пользователь", и у него имеется свое хранилище файлов.

- Android состоит из четырех видов компонентов: `Activity`, `Service`, `ContentProvider` и `BroadcastReceiver`. Они должны быть зарегистрированы и, возможно, иметь права доступа в манифесте Android.
  - Объекты `Activity` — это пользовательский интерфейс приложения Android. Они начинают свой жизненный цикл с метода `onCreate`, становятся доступными для взаимодействия с пользователем после `onResume` и могут быть прерваны (`onPause`) в любое время.
  - Экземпляры класса `Fragment` — это сложные существа со своими жизненными циклами. Их можно использовать для организации независимых UI-контейнеров на странице пользовательского интерфейса.
  - Службы могут быть запускаемыми и/или подключаемыми. API версии 26 начал внедрять ограничения на использование служб в фоновом режиме, поэтому есть общее правило, которое гласит: если пользователь каким-либо образом взаимодействует с задачей, то служба должна стать службой переднего плана.
  - Если `BroadcastReceiver` не использует неявное намерение, явно разрешенное системой с помощью действия, то, вероятно, необходимо динамически регистрировать его из кода приложения.
- Аккуратно используйте контекст активности. Жизненный цикл `Activity` не находится под контролем приложения. Ссылка на экземпляр `Activity` *должна* учитывать его фактический жизненный цикл.
- Универсальные программные архитектуры Android, такие как MVI, MVP и MVVM, разработаны таким образом, чтобы объекты `Fragment` и `Activity` были компактными, а также поощрялось лучшее разделение ответственностей и тестирование с учетом жизненного цикла.

Теперь, когда мы рассмотрели основные правила и изучили игровое поле, наше путешествие к созданию структурированных сопрограмм в Kotlin официально стартует. В следующей главе мы начнем применять эти основы для изучения памяти и многопоточности в Android. Понимание того, как организована эта система, позволит выявить проблемы, которые мы будем решать в последующих главах.

ГЛАВА 4

Параллельное программирование в Android

Эта глава не посвящена конкретно языку Kotlin. В ней будут представлены некоторые вопросы, связанные с *параллельным программированием*, о котором пойдет речь в оставшейся части книги. Здесь также будет представлен ряд инструментов, уже доступных разработчикам приложений для Android, для управления задачами параллельного программирования.

Параллельное программирование имеет репутацию своего рода темного искусства: это то, чем занимаются самопровозглашенные волшебники и к чему новички прикасаются на свой страх и риск. Конечно, написание правильных программ с использованием параллельного программирования может быть довольно сложной задачей. Это особенно верно, потому что ошибки в таких программах не всегда проявляются сразу. Почти невозможно выполнить проверку на предмет наличия ошибок параллелизма, и их может быть чрезвычайно трудно воспроизвести, даже если известно, что они есть.

Разработчику, обеспокоенному опасностями параллельного программирования, нужно помнить следующее:

- почти все, что вы делаете каждый день, *за исключением* программирования, выполняется параллельно. Вы неплохо ладите с параллельным окружением. Это в программировании все происходит по порядку, что можно считать странным;
- если вы пытаетесь понять проблемы, связанные с параллельным программированием, то вы на правильном пути. Даже неполное понимание параллелизма лучше, чем копирование образца кода и скрещивание пальцев;
- параллельное программирование — это принцип работы Android. Все, кроме самого тривиального приложения для этой системы, требует параллельного выполнения. Можно и дальше продолжать этот список!

Прежде чем углубляться в детали, определимся с терминами.

Первый термин — это *процесс*. Процесс — это память, которую может использовать приложение, и один или несколько потоков выполнения. Пространство памяти

принадлежит процессу — другие процессы не могут повлиять на это[1]. Обычно приложение выполняется как единый процесс.

Отсюда, конечно же, вытекает второй термин: *поток*. Поток — это последовательность инструкций, выполняемых по очереди.

А это приводит нас к термину, который в какой-то степени определяет всю остальную часть книги: *потокобезопасность*. Набор инструкций является потокобезопасным, если при его выполнении несколькими потоками никакое возможное упорядочение инструкций, выполняемых потоками, не может привести к результату, который нельзя было бы получить, если бы каждый из потоков выполнял код полностью, в некотором порядке, по одному за раз. Сложновато для понимания, но на самом деле речь здесь идет о том, что код дает одинаковые результаты независимо от того, выполняются ли несколько потоков одновременно или последовательно, один за другим. Это означает, что запуск программы дает предсказуемые результаты.

Так как же сделать программу потокобезопасной? По поводу этого есть очень много идей. Мы хотели бы предложить вариант, который ясен, относительно прост для понимания и всегда верен. Просто следуйте одному, довольно ясному и довольно простому правилу. Мы сформулируем его через несколько страниц. Однако сначала подробнее обсудим, что означает потокобезопасность.

Потокобезопасность

Мы уже упомянули, что потокобезопасный код не может дать результат при одновременном выполнении несколькими потоками, который нельзя было бы получить при упорядочивании выполнения потоков по одному за раз. Это определение, однако, не очень полезно на практике: никто не будет тестировать все возможные порядки выполнения.

Возможно, мы можем справиться с этой проблемой, рассмотрев распространенные способы, при которых код может быть *небезопасным для потоков*.

Ошибки, связанные с потокобезопасностью, можно разделить на несколько категорий. Две наиболее важные из них — это *атомарность* и *видимость*.

Атомарность

Почти все разработчики понимают проблемы атомарности. Следующий код не является потокобезопасным:

```
fun unsafe() { globalVar++ }
```

Несколько потоков, выполняющих этот код, могут мешать друг другу. Каждый поток, выполняющий его, может считывать одно и то же значение переменной

[1] Процессы могут совместно использовать часть памяти (как в случае с Binder), но делают они это под усиленным контролем.

globalVar — пусть это будет 3. Каждый поток может увеличивать это значение, чтобы получить 4, а затем обновить переменную, чтобы у нее было значение 4.

Даже если бы 724 потока выполнили этот код, переменная globalVar могла бы иметь значение 4, когда все потоки завершат выполнение.

Невозможна ситуация, в которой каждый из 724 потоков мог бы выполнить этот код последовательно, и при этом значение globalVar было бы равно 4. Поскольку результат одновременного выполнения кода может отличаться от любого возможного результата, генерируемого последовательным выполнением, согласно нашему определению данный код не является потокобезопасным.

Чтобы превратить код в потокобезопасный, нужно сделать операции чтения, инкремента и записи для переменной globalVar *атомарными*. Атомарной называется операция, которую нельзя прервать другим потоком. Если операции чтения, инкремента и записи будут атомарными, то два потока не смогут увидеть одно и то же значение globalVar, и программа гарантированно будет вести себя, как и ожидалось.

Атомарность легко понять.

Видимость

Вторую категорию ошибок, связанных с потокобезопасностью, видимость, обнаружить гораздо труднее. Этот код также не является потокобезопасным:

```
var shouldStop = false

fun runner() {
    while (!shouldStop) { doSomething() }
}

fun stopper() { shouldStop = true }
```

Поток, выполняющий функцию runner, может так и не остановиться, даже если другой поток выполнит функцию stopper. Поток, выполняющий функцию runner, может так и не заметить, что значение shouldStop изменилось на true.

Причина этого — оптимизация. И аппаратные средства (с использованием регистров, многоуровневых кэшей и т. д.), и компилятор (используя понятия "объявление", "переупорядочивание" и т. д.) делают все возможное, чтобы ваш код работал быстро. Для этого инструкции, которые на самом деле выполняет аппаратное обеспечение, могут быть не очень похожи на исходный код Kotlin. В частности, хотя вы считаете, что shouldStop — это одна переменная, у аппаратного обеспечения, вероятно, есть для нее как минимум два представления: одно в регистре и одно в основной памяти.

Вам определенно это нужно! Вряд ли вы захотите, чтобы циклы в вашем коде зависели от доступа к основной памяти вместо использования кэшей и регистров. Быстрая память оптимизирует код, поскольку время доступа к ней на несколько порядков меньше, чем к основной памяти.

Однако, чтобы код из примера работал, нужно объяснить компилятору, что он не может хранить значение `shouldStop` в локальной памяти (регистре или кэше). Если, как предлагается, есть несколько представлений `shouldStop` в разных видах аппаратной памяти, то компилятор должен быть уверен, что значение, хранящееся в быстром локальном представлении `shouldStop`, помещается в память, видимую для всех потоков. Это называется *публикацией* значения.

Есть способ сделать это. Это аннотация `@Synchronized`. Синхронизация сообщает компилятору: он должен убедиться, что любые побочные эффекты кода, выполняемого в синхронизированном блоке, видны всем другим потокам до того, как исполняющий поток покинет блок.

Таким образом, синхронизация касается не столько аппаратного обеспечения или хитрых и сложных критериев того, что должно быть защищено, а что — нет. Синхронизация — это контракт между разработчиком и компилятором. Если вы не синхронизируете код, компилятор может выполнить любую оптимизацию, которая может оказаться безопасной, на основе последовательного выполнения. Если где-то есть другой код в другом потоке, который делает работу компилятора недействительной, нужно синхронизировать код.

Итак, если вы хотите писать потокобезопасный код, просто следуйте короткому и ясному правилу. Перефразируем библию параллельного программирования на языке Java (Java Concurrency[2]) на практике и скажем так: "Всякий раз, когда несколько потоков получают доступ к конкретной переменной состояния и один из них может записать в нее данные, все они должны координировать свой доступ к ней с помощью синхронизации".

Кстати, обратите внимание, что в этой цитате для синхронизации не проводится различие между доступом для чтения и доступом для записи. Если не гарантируется, что *никто* не изменит разделяемое состояние, то все операции доступа, чтения или записи должны быть синхронизированы.

Модель многопоточного выполнения Android

Как было отмечено в *главе 3*, одним из следствий архитектуры MVC является однопоточный пользовательский интерфейс (представление и контроллер). Хотя многопоточный пользовательский интерфейс и выглядит очень заманчиво (конечно, вариант "один поток для представления и один поток для контроллера" сработал бы...), попытки создать его были прекращены еще в 1970-е годы, когда стало ясно, что всё неизбежно заканчивается клубком взаимных блокировок.

С момента повсеместного внедрения паттерна MVC стандартный дизайн пользовательского интерфейса представляет собой очередь, обслуживаемую одним потоком (в Android — *основным* или *UI-потоком*). Как показано на рис. 4.1, события — как исходящие от пользователя (щелчки мышью по кнопке, касания, ввод текста

[2] См. **https://oreil.ly/4zx8L**.

и т. д.), так и исходящие от модели (анимация, запросы на перерисовку или обновление экрана и т. д.) — ставятся в очередь и в конечном счете обрабатываются по порядку одним UI-потоком.

Рис. 4.1. UI-поток

Именно так работает пользовательский интерфейс Android. Основной поток приложения (исходный поток процесса приложения) становится его UI-потоком. В рамках инициализации поток входит в замкнутый цикл. На протяжении всего оставшегося жизненного цикла приложения он одну за другой удаляет задачи из очереди канонического пользовательского интерфейса и выполняет их. Поскольку методы пользовательского интерфейса всегда выполняются в одном потоке, компоненты пользовательского интерфейса не пытаются быть потокобезопасными.

Звучит здорово, правда? Однопоточный пользовательский интерфейс, и не нужно беспокоиться о потокобезопасности. Однако есть проблема. Чтобы понять ее, нам придется отвлечься и поговорить об опыте взаимодействия конечных пользователей, которые работают с устройствами под управлением Android. В частности, нужно будет изучить детали отображения видео.

Когда каждый поток содержит ресурс, который требуется другому, это называется *взаимной блокировкой*: ни один из потоков не может продолжать работу. Например, один поток может содержать виджет, отображающий значение, и требовать отображения контейнера, содержащего это значение. В то же время другой поток может содержать контейнер и требовать виджет. Взаимоблокировок можно избежать, если все потоки всегда захватывают ресурсы в одном и том же порядке.

Пропуск кадров

Из обширного опыта просмотра фильмов и телевизионных программ мы знаем, что человеческий мозг можно обмануть, заставив воспринимать движение как непрерывное, даже если оно таковым не является. Серия неподвижных изображений, быстро демонстрируемых одно за другим, может казаться наблюдателю плавной и непрерывной. Скорость, с которой выводятся изображения, известна как *частота кадров*. Она измеряется в *кадрах в секунду* (frames per second, fps).

Стандартная частота кадров для фильмов составляет 24 кадра в секунду, что неплохо работало на протяжении всего золотого века Голливуда. В старых телевизорах использовалась частота кадров 30 кадров в секунду. Как вы можете себе представить, более высокая частота кадров даже лучше обманывает мозг, чем медленная. Даже если вы не можете точно определить, что вы чувствуете, при просмотре видео с высокой частотой кадров и видео с более низкой частотой кадров, вы, вероятно, заметите разницу. Более быстрый вариант будет казаться "плавнее".

Во многих устройствах, работающих под управлением Android, используется частота кадров, равная 60 кадрам в секунду. Это приводит к перерисовке экрана примерно каждые 16 мс и означает, что UI-поток, единственный поток, обрабатывающий задачи пользовательского интерфейса, должен иметь новое доступное изображение, готовое к показу на экране каждые 16 мс. Если создание изображения занимает больше времени и новое изображение не готово в момент перерисовки экрана, говорят, что кадр пропущен.

Пройдет еще 16 мс, прежде чем снова произойдет обновление экрана и мы увидим новый кадр. Пропуск кадров снижает частоту кадров до 30 кадров в секунду, что близко к порогу, при котором человеческий мозг это замечает. Всего несколько пропущенных кадров могут придать пользовательскому интерфейсу ощущение прерывистости, которое иногда называют "подвисанием".

Рассмотрим очередь задач, показанную на рис. 4.2, при стандартной скорости отображения 60 кадров в секунду.

Рис. 4.2. Задачи в очереди для UI-потока

Первая задача, обрабатывающая ввод символов пользователем, выполняется за 8 мс. Следующая задача, обновляющая представление, является частью анимации. Чтобы все выглядело плавно, анимация должна обновляться не менее 24 раз в секунду. Тем не менее третья задача, обработка щелчка мышью по кнопке, занимает 22 мс. Последняя задача на диаграмме — это следующий кадр анимации. На рис. 4.3 показано, что видит UI-поток.

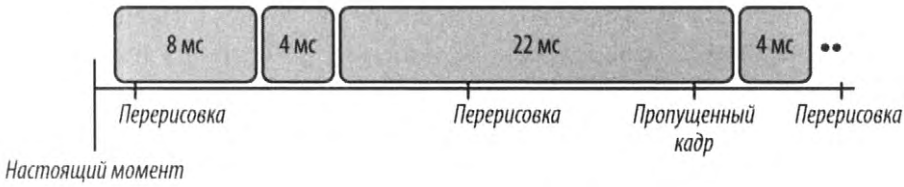


Рис. 4.3. Пропуск кадра

Первая задача завершается за 8 мс. Кадр в буфере дисплея рисуется за 4 мс. Затем UI-поток начинает обрабатывать щелчок мышью. Через пару миллисекунд после обработки щелчка происходит перерисовка, и кадр анимации теперь виден на экране.

К сожалению, спустя 16 мс задача по обработке щелчка еще не завершена. Не обработана задача отрисовки следующего кадра анимации, стоящего за ним в очереди. Когда происходит обновление, содержимое буфера отображения точно такое же, как и во время предыдущего обновления. Анимационный кадр был пропущен.

Компьютерные дисплеи обычно управляются с помощью одного или нескольких буферов дисплея. *Буфер дисплея* — это область памяти, в которой пользовательский код "рисует" то, что будет отображаться на экране. Изредка на *интервале обновления* (примерно 16 мс для дисплея с частотой 60 кадров в секунду) пользовательский код на короткое время блокируется за пределами буфера. Система использует содержимое буфера для отображения экрана, а затем возвращает его в пользовательский код для дальнейших обновлений.

Через несколько миллисекунд, когда задача обработки щелчка мышью завершена, задача анимации получает возможность обновить буфер дисплея. Несмотря на то что буфер дисплея теперь содержит следующий кадр анимации, экран не будет перерисовываться в течение нескольких миллисекунд. Частота кадров для анимации была сокращена вдвое, до 30 кадров в секунду, что опасно близко к видимому мерцанию.

Некоторые новые устройства, такие как Google Pixel 4, имеют возможность обновлять экран с гораздо более высокой частотой кадров. При частоте кадров, которая, например, в два раза выше (120 кадров в секунду), даже если UI-поток пропустит два кадра подряд, ему все равно придется ждать еще 8 мс для следующей перерисовки. Интервал между двумя отображениями в данном случае составляет всего около 24 мс; намного лучше, чем 32 мс при пропуске кадра при 60 кадрах в секунду.

Хотя увеличение частоты кадров может помочь, разработчик Android должен быть бдителен и следить за тем, чтобы приложение пропускало как можно меньше кадров. Если приложение выполняет затратные вычисления, и эти вычисления занимают больше времени, чем ожидалось, оно пропустит временной интервал перерисовки и отбросит кадр. Тогда приложение не будет работать плавно.

Данный сценарий является причиной, по которой просто необходимо использовать параллельное программирование в приложениях для Android. Проще говоря, пользовательский интерфейс является однопоточным, а UI-поток ни при каких обстоятельствах не должен занимать более нескольких миллисекунд.

Единственное возможное решение для нетривиального приложения — передать трудоемкую работу (хранение и поиск в базе данных, сетевые взаимодействия и длительные вычисления) другому потоку.

Утечка памяти

Мы уже рассмотрели одну проблему, связанную с параллелизмом, — потокобезопасность. Компонентная архитектура Android добавляет вторую, не менее опасную проблему — *утечку памяти*.

Утечка памяти происходит, когда объект не может быть освобожден (удален), даже если он больше не нужен. В худшем случае утечка памяти способна привести к ошибке `OutOfMemoryError` и сбою приложения. Однако, даже если дела обстоят не так уж плохо, нехватка памяти может привести к более частому сбору мусора, что снова вызовет "подвисание".

Как обсуждалось в *главе 3*, приложения Android особенно подвержены утечкам памяти, поскольку жизненные циклы некоторых наиболее часто используемых компонентов — `Activity`, `Fragment`, `Service` и т. д. — не контролируются приложением. Экземпляры этих компонентов могут очень легко превратиться в мертвый груз, что особенно верно в случае с многопоточным окружением. Попробуйте передать задачу рабочему потоку:

```
override fun onViewCreated(
    view: View,
    savedInstanceState: Bundle?
) {
    // НЕ ДЕЛАЙТЕ ЭТОГО!
    myButton.setOnClickListener {
        Thread {
            val status = doTimeConsumingThing()
            view.findViewById<TextView>(R.id.textview_second)
                .setText(status)
        }
            .start()
    }
}
```

Идея перенести трудоемкую работу с UI-потока не лишена благородства. К сожалению, у предыдущего кода имеется ряд недостатков. Можете их заметить?

Во-первых, как упоминалось ранее в этой главе, компоненты пользовательского интерфейса Android не являются потокобезопасными и к ним нельзя получить доступ или изменить их за пределами UI-потока. Вызов метода `setText` в этом коде из потока, отличного от UI-потока, неверен. Многие компоненты пользовательского интерфейса Android обнаруживают подобные небезопасные варианты использования и выбрасывают исключение, если оно случается.

Один из способов решения этой проблемы — возврат результатов в UI-поток с помощью одного из методов из набора инструментов Android для безопасной отправ-

ки потоков, как показано далее. Обратите внимание, что у этого кода *по-прежнему* есть недостатки!

```
override fun onViewCreated(
    view: View,
    savedInstanceState: Bundle?
) {
    // И ТАК ДЕЛАТЬ ТОЖЕ НЕ СТОИТ!
    myButton.setOnClickListener {
        Thread {
            val status = doTimeConsumingThing()
            view.post {
                view.findViewById<TextView>(R.id.textview_second)
                    .setText(status)
            }
        }
            .start()
    }
}
```

Данный код устраняет первую проблему (метод пользовательского интерфейса `setText` теперь вызывается из основного потока), но код по-прежнему неверен. Хотя причуды языка затрудняют понимание проблемы, она заключается в том, что поток, недавно созданный в `ClickListener`, содержит неявную ссылку на объект, управляемый Android. Поскольку `doTimeConsumingThing` — это метод `Activity` (или `Fragment`), поток, вновь созданный в обработчике события щелчка по кнопке, содержит *неявную* ссылку на эту активность (рис. 4.4).

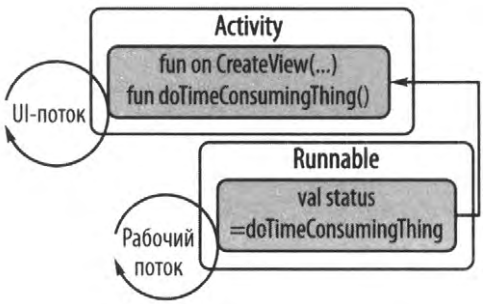


Рис. 4.4. Утечка памяти `Activity`

Было бы более очевидно, если бы мы написали вызов `doTimeConsumingThing` как `this.doTimeConsumingThing`. Однако, если подумать, то становится ясно, что способа вызвать метод `doTimeConsumingThing` для какого-либо объекта (в данном случае экземпляра `Activity`) без ссылки на этот объект нет. Теперь сборщик мусора не может удалить экземпляр `Activity`, пока объект, реализующий интерфейс `Runnable`, который выполняется в рабочем потоке, содержит ссылку на него. Если поток выполня-

ется в течение значительного промежутка времени, происходит утечка памяти `Activity`.

Решить эту проблему значительно сложнее, чем предыдущую. Один из подходов предполагает, что задачи, которые гарантированно содержат такую неявную ссылку только в течение очень короткого периода времени (менее секунды), могут не вызывать проблем. Сама система Android иногда создает такие короткие задачи.

`ViewModel` и `LiveData` гарантируют, что пользовательский интерфейс всегда отображает самые свежие данные и делает это безопасно. В сочетании с `ViewModelScope` от Jetpack и сопрограммами — скоро вы с ними познакомитесь — все эти вещи облегчают управление отменой фоновых задач, которые больше не актуальны, и обеспечивают целостность памяти и потокобезопасность. Без библиотек нам пришлось бы решать все эти проблемы самостоятельно.

Android Jetpack

Тщательный дизайн с использованием контейнеров `LiveData` с учетом жизненного цикла от Jetpacks, как описано в *главе 3*, может помочь устранить утечки памяти и опасность использования компонента Android, завершившего свой жизненный цикл.

Инструменты для управления потоками

На самом деле в коде, который мы только что обсуждали, есть и третий недостаток, глубокий и конструктивный.

Потоки — очень дорогостоящие объекты. Они большие, влияют на сбор мусора, а переключение контекста между ними далеко не бесплатное удовольствие. Создание и удаление потоков, как в коде из примера, довольно расточительно, опрометчиво и, вероятно, влияет на производительность приложения.

Создание большего количества потоков никоим образом не делает приложение способным выполнить больший объем работы: мощность процессора ограничена. Потоки, которые не выполняются, — это просто дорогой способ представления работы, которая еще не завершена.

Рассмотрим, например, что произойдет, если пользователь нажмет `myButton` из предыдущего примера. Даже если операции, которые выполнял каждый из сгенерированных потоков, были быстрыми и потокобезопасными, их создание и удаление замедляло работу приложения.

Более подходящей практикой для приложений является политика потоков: стратегия для всего приложения, основанная на количестве действительно полезных потоков, которая контролирует, сколько потоков выполняется в любой момент времени. Умное приложение поддерживает один или несколько пулов потоков, каждый из которых предназначен для определенной цели и перед каждым из которых стоит очередь. Клиентский код с незавершенной работой ставит в очередь задачи, чтобы

они выполнялись пулом потоков и, при необходимости, восстанавливает результаты задачи.

В следующих двух разделах представлены два примитивных типа, доступные разработчикам Android: `Looper/Handler` и `Executor`.

Looper/Handler

`Looper/Handler` — это фреймворк, состоящий из взаимодействующих классов: `Looper`, `MessageQueue` и поставленных в очередь объектов `Message`, а также одного или несколько обработчиков.

`Looper` запускает цикл обработки сообщений, связанный с потоком. Он инициализируется вызовом методов `Looper.prepare()` и `Looper.start()` из метода `run`, например:

```
var looper = Thread {
    Looper.prepare()
    Looper.loop()
}
looper.start()
```

Второй метод, `Looper.loop()`, заставляет поток войти в тесный цикл, в котором он проверяет очередь `MessageQueue` на наличие задач, удаляет их одну за другой и выполняет. Если задач для выполнения нет, поток "спит", пока новая задача не будет поставлена в очередь.

Если вы поймали себя на мысли, что это звучит смутно знакомо, то вы правы. UI-поток Android — это просто экземпляр `Looper`, созданный из основного потока процесса приложения.

`Handler` — это механизм, используемый для постановки задач в очередь `Looper` для обработки. Создается он следующим образом:

```
var handler = new Handler(someLooper);
```

`Looper` основного потока всегда доступен с помощью метода `Looper.getMainLooper`. Таким образом, создать обработчик, который отправляет задачи UI-потоку, так же просто, как написать этот код:

```
var handler = new Handler(Looper.getMainLooper);
```

Фактически именно так и работает метод `post()`, показанный в предыдущем примере.

Обработчики интересны тем, что они обрабатывают оба конца очереди `Looper`. Чтобы увидеть, как это работает, проследим за отдельной задачей.

Существует несколько методов `Handler` для постановки задачи в очередь. Вот два из них:

- `post(task: Runnable);`
- `send(task: Message).`

Эти методы определяют два немного разных способа постановки задачи в очередь: отправка сообщений и передача объектов, реализующих интерфейс Runnable. На самом деле обработчик всегда ставит объект Message в очередь. Однако для удобства группа методов post...() связывает объект, реализующий интерфейс Runnable, с объектом Message для специальной обработки.

В этом примере мы используем метод Handler.post(task: Runnable), чтобы поставить задачу в очередь. Обработчик получает объект Message из пула, привязывает объект, реализующий интерфейс Runnable и добавляет Message в конец очереди MessageQueue.

Теперь наша задача ожидает выполнения. Когда она достигает начала очереди, Looper подхватывает ее и, что интересно, передает обратно тому же обработчику, который поставил ее в очередь. Один и тот же экземпляр Handler, который ставит задачу в очередь, всегда является экземпляром, который ее запускает.

Это может показаться немного запутанным, пока вы не поймете, что код Handler, отправивший задачу, может выполняться в любом потоке приложения. Однако код Handler, который обрабатывает задачу, всегда выполняется в Looper, как показано на рис. 4.5.

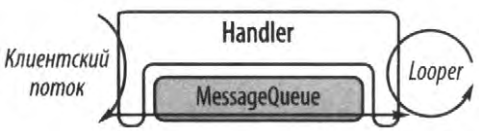


Рис. 4.5. Looper/Handler

Метод Handler, вызываемый Looper для обработки задачи, сначала проверяет, содержит ли Message объект, реализующий интерфейс Runnable. Если да — и поскольку мы использовали один из методов post...(), то и наша задача тоже — то обработчик выполняет Runnable.

Если бы мы использовали один из методов send...(), то Handler передал бы Message собственному переопределяемому методу Handler.handleMessage(msg: Message). Подкласс Handler из этого метода будет использовать атрибут Message, what, чтобы решить, какую конкретную задачу он должен выполнять, и атрибуты arg1, arg2 и obj в качестве параметров задачи.

На самом деле MessageQueue представляет собой отсортированную очередь. Каждый объект Message включает в качестве одного из своих атрибутов самое раннее время, когда он может быть выполнен. В двух предыдущих методах, post и send, просто используйте текущее время (сообщение будет обработано "сейчас же", незамедлительно).

Однако два других метода позволяют ставить задачи в очередь, чтобы ее можно было выполнить в будущем:

- postDelayed(runnable, delayMillis);
- sendMessageDelayed(message, delayMillis).

Задачи, создаваемые с использованием этих методов, будут отсортированы в MessageQueue для выполнения в указанное время.

Как уже было отмечено, Looper может приложить максимум усилий для выполнения задачи только в запрошенное время. Хотя он никогда не будет запускать отложенную задачу раньше времени, если другая задача захватывает поток, задача может выполняться с опозданием.

Looper/Handlers — фантастически универсальный и эффективный инструмент. Система Android широко использует его, особенно вызовы send...(), которые не производят никакого выделения памяти.

Обратите внимание, что Looper может отправлять задачи самому себе. Задачи, которые выполняются, а затем планируются повторно через заданный интервал (используя один из методов ...Delayed()), являются одними из способов, с помощью которых в Android создается анимация.

Также обратите внимание, что поскольку Looper является однопоточным, задача, которая работает только в одном конкретном объекте Looper, не обязательно должна быть потокобезопасной. Нет необходимости в синхронизации или упорядочении, когда задача, даже если она функционирует асинхронно, выполняется только на одном потоке. Как упоминалось ранее, весь UI-фреймворк Android, работающий только в Looper пользовательского интерфейса, зависит от этого предположения.

Исполнители *Executors* и объекты *ExecutorService*

Исполнители (Executors) и объекты ExecutorService появились в Java 5 как часть нового фреймворка параллельного программирования. В нем представлены несколько абстракций параллелизма более высокого уровня, которые позволили разработчикам оставить позади многие детали потоков, блокировок и синхронизации.

Executor — как следует из названия, это утилита, выполняющая поставленные перед ней задачи. Ее контракт — это единственный метод execute(Runnable).

Java предоставляет несколько реализаций данного интерфейса, у каждой из которых своя стратегия выполнения и назначение. Самая простая из них доступна с помощью метода Executors.newSingleThreadExecutor.

Однопоточный исполнитель очень похож на Looper/Handler, который мы рассматривали в предыдущем разделе, — это неограниченная очередь перед одним потоком. Новые задачи ставятся в очередь, а затем удаляются и выполняются по порядку в одном потоке, который обслуживает очередь.

У Looper/Handler и однопоточных исполнителей есть свои преимущества. Например, Looper/Handler сильно оптимизирован, чтобы можно было избежать выделения памяти под объекты. С другой стороны, однопоточный исполнитель заменит поток, если тот будет прерван неудачной задачей.

FixedThreadPoolExecutor — это обобщенный вариант однопоточного исполнителя: вместо одного потока его неограниченная очередь обслуживается фиксированным числом потоков. Как и однопоточный исполнитель, FixedThreadPoolExecutor заменит потоки, когда задачи их остановят. Однако он не гарантирует порядок задач и будет выполнять их одновременно, если аппаратное обеспечение это позволяет.

Однопоточный запланированный исполнитель представляет собой эквивалент `Looper/Handler` в Java. Он похож на однопоточный исполнитель, за исключением того, что, как и у `Looper/Handler`, его очередь сортируется по времени выполнения. Задачи выполняются в порядке времени, а не в порядке отправки. Как и в случае с `Looper/Handler`, длительные задачи могут помешать своевременному выполнению последующих задач.

Если ни одна из этих стандартных утилит выполнения не отвечает вашим потребностям, можно создать собственный экземпляр `ThreadPoolExecutor`, указав такие детали, как размер и порядок очереди, количество потоков в пуле, способ их создания и происходящие события, когда очередь пула заполнена.

Отдельного внимания заслуживает еще один тип исполнителя — `ForkJoinPool`. Его существование объясняется наблюдением, согласно которому иногда одну проблему можно разбить на несколько подзадач, которые могут выполняться параллельно.

Типичный пример такой проблемы — сложение двух массивов одинакового размера. Синхронное решение состоит в том, чтобы выполнить итерацию, `i = 0 .. n - 1`, где `n` — это размер массива, и для каждого `i` вычислить `s[i] = a1[i] + a2[i]`.

Однако возможна умная оптимизация, если разделить задачу на части. В нашем случае задачу можно разбить на `n'` подзадач, каждая из которых вычисляет `s[i] = a1[i] + a2[i]` для некоего `i`.

Обратите внимание, что служба выполнения, создающая подзадачи, которые она должна обрабатывать *сама*, может ставить подзадачи в локальную очередь. Поскольку локальная очередь используется преимущественно одним потоком, конфликтов из-за блокировок очередей почти никогда не возникает. Бóльшую часть времени очередь принадлежит потоку — он один помещает туда что-то и убирает. Это может быть настоящей оптимизацией.

Рассмотрим пул этих потоков, у каждого из которых есть своя быстрая локальная очередь. Предположим, что один из потоков заканчивает всю работу и будет простаивать, а в то же самое время у другого потока пула есть очередь из 200 подзадач, которые нужно выполнить. Бездействующий поток занимает работу. Он захватывает мьютекс очереди занятого потока, захватывает половину подзадач, помещает их в свою очередь и переходит к работе над ними.

Трюк с заниманием работ наиболее полезен, когда параллельные задачи порождают собственные подзадачи. Оказывается, как будет показано, именно такими задачами и являются сопрограммы Kotlin.

Инструменты для управления заданиями

Подобно тому, как при производстве, скажем, автомобилей может быть эффект масштаба, существуют важные оптимизации, требующие крупномасштабного представления системы. Рассмотрим использование радио в мобильном телефоне.

Когда приложению необходимо взаимодействовать с удаленной службой, телефон, обычно находящийся в режиме экономии заряда батареи, должен включить радио, подключиться к ближайшей вышке, договориться о соединении, а затем передать сообщение. Поскольку согласование сеанса связи несет с собой затраты, телефон какое-то время держит соединение открытым. Предполагается, что, когда происходит одно межсетевое взаимодействие, вполне вероятно, что за ним последуют и другие. Однако, когда более минуты или около того сеть не используется, телефон возвращается в состояние покоя и экономии заряда батареи.

Учитывая такое поведение, представьте, что именно происходит, когда несколько приложений звонят домой, каждое в разное время. Когда первое приложение посылает свой сигнал, телефон включает радио, согласовывает соединение, передает сообщение для приложения, немного ждет, а затем возвращается в спящий режим. Однако, как только он снова переходит в спящий режим, следующее приложение пытается использовать сеть. Телефон должен снова включиться, повторно согласовать соединение и т. д. Если для этого используется несколько приложений, радио в телефоне работает на полную мощность практически все время. Кроме того, оно тратит много времени на повторное согласование межсетевого подключения, которое было разорвано всего несколько секунд назад.

Ни одно приложение не может предотвратить такого рода проблемы. Требуется общесистемное представление об использовании батареи и сети для координации нескольких приложений (и у каждого из них свои требования) и оптимизации срока службы батареи.

В Android 8.0 (API версии 26 и выше) введены ограничения на потребление ресурсов приложения. В число этих ограничений входят следующие:

- приложение находится на переднем плане только тогда, когда имеет видимую активность или работает служба переднего плана. Подключаемые и запускаемые службы больше не препятствуют остановке приложения;
- приложения не могут использовать свой манифест для регистрации с целью получения неявных широковещательных сообщений. Кроме того, существуют ограничения на отправку этих сообщений.

Эти ограничения могут затруднить выполнение приложением "фоновых" задач: синхронизация с удаленным устройством, место записи и т. д. В большинстве случаев ограничения можно смягчить с помощью инструментов `JobScheduler` или `WorkManager` из Jetpack.

Всякий раз, когда средние или большие задачи должны быть запланированы более чем на несколько минут в будущем, рекомендуется использовать один из этих инструментов. Размер имеет значение: обновление анимации каждые несколько миллисекунд или планирование еще одной проверки местоположения через пару секунд — это, вероятно, отличная вещь, которой можно заниматься с планировщиком на уровне потоков. Обновление базы данных из вышестоящей системы каждые 10 минут определенно следует выполнять с помощью `JobScheduler`.

JobScheduler

JobScheduler — это инструмент Android для планирования задач (возможно, повторяющихся задач) в будущем. Он достаточно адаптивен и, помимо оптимизации времени автономной работы, обеспечивает доступ к сведениям о состоянии системы, которые приложения должны были выводить на основе эвристики.

По сути, задание JobScheduler — это подключаемая служба. Приложение объявляет специальную службу в своем манифесте, чтобы сделать ее видимой для системы Android. Затем оно планирует задачи для службы, использующей JobInfo.

Если условия JobInfo выполнены, то Android подключает службу задач так же, как описано в *разд. "Подключаемые службы" главы 3*. После подключения задачи Android велит службе запуститься и передает все соответствующие параметры.

Первый шаг в создании задачи JobScheduler — ее регистрация в манифесте приложения. Как это сделать, показано здесь:

```
<service
    android:name=".RecurringTask"
    android:permission="android.permission.BIND_JOB_SERVICE"/>
```

Важная вещь в этом объявлении — права доступа. Если служба не объявлена *именно* с правами android.permission.BIND_JOB_SERVICE, то JobScheduler не сможет ее найти.

Обратите внимание, что другие приложения не видят службу задач. Это не проблема. JobScheduler является частью системы Android и может видеть то, чего не видят обычные приложения.

Следующий шаг в настройке задачи JobScheduler — ее планирование, как показано здесь, в методе schedulePeriodically:

```
const val TASK_ID = 8954
const val SYNC_INTERVAL = 30L
const val PARAM_TASK_TYPE = "task"
const val SAMPLE_TASK = 22158

class RecurringTask() : JobService() {
    companion object {
        fun schedulePeriodically(context: Context) {
            val extras = PersistableBundle()
            extras.putInt(PARAM_TASK_TYPE, SAMPLE_TASK)

            (context.getSystemService(Context.JOB_SCHEDULER_SERVICE)
                as JobScheduler)
                .schedule(
                    JobInfo.Builder(
                        TASK_ID,
                        ComponentName(
```

```
                        context,
                        RecurringTask::class.java
                    )
                )
                    .setPeriodic(SYNC_INTERVAL)
                    .setRequiresStorageNotLow(true)
                    .setRequiresCharging(true)
                    .setExtras(extras)
                    .build()
            )
        }
    }

    override fun onStartJob(params: JobParameters?): Boolean {
        // делаем что-то
        return true;
    }

    override fun onStopJob(params: JobParameters?): Boolean {
        // перестаем это делать
        return true;
    }
}
```

Эта задача будет выполняться каждые SYNC_INTERVAL секунд, но только в том случае, если на устройстве достаточно места и если в настоящее время оно подключено к внешнему источнику питания. Это только два из множества атрибутов, доступных для планирования задачи. Детализация и гибкость планирования — пожалуй, самое привлекательное качество JobScheduler.

Обратите внимание, что JobInfo идентифицирует класс задачи, который нужно запустить, почти так же, как мы определяли цель для объекта Intent в *главе 3*.

Система вызовет метод задачи onStartJob на основе критериев, заданных в JobInfo, когда задача сможет быть запущена. Вот почему существует JobScheduler. Поскольку ему известны расписания и требования для всех запланированных задач, он может оптимизировать планирование в глобальном масштабе, чтобы свести к минимуму воздействие, особенно на батарею.

Будьте осторожны! Метод onStartJob запускается в основном потоке. Если, что весьма вероятно, запланированная задача займет больше нескольких миллисекунд, она должна быть запланирована в фоновом потоке с использованием одной из техник, описанных ранее.

Если метод onStartJob возвращает true, то система разрешает запуск приложения до тех пор, пока не будет вызван метод jobFinished или условия, описанные в JobInfo,

больше не будут выполняться. Если, например, телефон, на котором выполняется RecurringTask из предыдущего примера, был отключен от источника питания, то система тот час же вызовет метод onStopJob() запущенной задачи, чтобы уведомить ее о том, что она должна остановиться.

Когда задача JobScheduler получает вызов метода onStopJob(), она должна остановиться. Документация предполагает, что у задачи есть немного времени, чтобы привести все в порядок и чисто завершить работу. К сожалению, довольно неясно, сколько именно времени подразумевается под словом "немного". Однако предупреждение о том, что "вы несете единоличную ответственность за поведение вашего приложения после получения этого сообщения; ваше приложение, скорее всего, начнет плохо себя вести, если вы его проигнорируете", довольно пугающе.

Если метод onStopJob() возвращает false, то задача не будет планироваться снова, даже если критерии в JobInfo соблюдены: задание было отменено. Повторяющаяся задача всегда должна возвращать true.

WorkManager

WorkManager — это библиотека Android Jetpack, которая является оберткой для JobScheduler. Она позволяет единому коду оптимально использовать современные версии Android — те, что поддерживают JobScheduler, — и по-прежнему работать с устаревшими версиями, которые его не поддерживают.

Хотя службы, предоставляемые WorkManager, а также ее API, аналогичны службам, предоставляемым JobScheduler, оберткой для которого она является, они на один шаг дальше от деталей реализации и на одну абстракцию короче.

Если JobScheduler шифрует разницу между задачей, которая периодически повторяется, и задачей, которая выполняется один раз, в значении true или false из метода onStopJob, то WorkManager делает это явным; существует два типа задач: OneTimeWorkRequest и PeriodicWorkRequest.

При постановке рабочего запроса в очередь всегда возвращается токен, объект WorkRequest, который можно использовать для отмены задачи, если она больше не нужна.

WorkManager также поддерживает построение сложных цепочек задач: "выполните это и это параллельно, и выполните вот то, когда все будет готово". Эти цепочки задач могут даже напомнить вам цепочки, которые мы использовали для преобразования коллекций в *главе 2*.

WorkManager — это наиболее удобный и лаконичный способ гарантировать выполнение необходимых задач (даже если ваше приложение не отображается на экране устройства) и сделать это таким образом, чтобы оптимизировать использование батареи.

Резюме

В этой главе мы представили модель многопоточного выполнения в Android, а также некоторые концепции и инструменты, которые помогут вам эффективно ее использовать. Подведем итоги.

- Потокобезопасная программа — это программа, которая ведет себя независимо от того, как ее выполняют параллельные потоки, таким образом, что ее можно воспроизвести, если бы одни и те же потоки выполняли ее последовательно.
- В модели многопоточного выполнения UI-поток отвечает за:
  - отрисовку представления;
  - отправку событий, возникающих в результате взаимодействия пользователя с пользовательским интерфейсом.
- Программы Android используют несколько потоков, чтобы гарантировать, что UI-поток может свободно перерисовывать экран без пропуска кадров.
- Java и Android предоставляют несколько многопоточных примитивных типов на уровне языка:
  - `Looper/Handler` — это очередь задач, обслуживаемых одним выделенным потоком;
  - исполнители и объекты `ExecutionService` — это конструкции Java для реализации политики управления потоками на уровне приложения.
- ОС Android предлагает архитектурные компоненты `JobScheduler` и `WorkManager` для эффективного планирования задач.

Следующие главы будут посвящены более сложным темам Android и параллельного программирования. В них мы рассмотрим, как Kotlin делает управление параллельными процессами более понятным, простым и менее подверженным ошибкам.

ГЛАВА 5
Потокобезопасность

С появлением в Java 5 пакета *java.util.concurrent* потоки стали широко использоваться для повышения производительности сложных приложений. В графических приложениях они улучшают время отклика за счет снижения нагрузки на основной поток, обрабатывающий информацию для отображения *представлений* — программируемых компонентов, которые видит пользователь и с которыми он может взаимодействовать на экране. Когда поток создается в программе, имеющей концепцию основного или UI-потока, он называется *фоновым*. Фоновые потоки часто получают и обрабатывают события взаимодействия с пользователем, например жесты и ввод текста, другие формы извлечения данных, такие как чтение с сервера или локальные хранилища, такие как база данных или файловая система. На стороне сервера серверные приложения, использующие потоки, обладают лучшей пропускной способностью за счет использования нескольких ядер современных процессоров.

Однако, как вы увидите в этой главе, у использования потоков есть и свои риски. Потокобезопасность можно рассматривать как набор методов и передовых практик для обхода этих рисков. Эти методы включают *синхронизацию*, *мьютексы*, а также сравнение *блокирующих* и *неблокирующих вызовов*. Важны и высокоуровневые концепции, такие как привязка к потоку.

Цель этой главы — познакомить вас с важными концепциями потокобезопасности, которые будут использоваться в последующих главах. Однако мы не станем подробно рассматривать эту тему. Например, мы не будем объяснять *публикацию объектов* или предоставлять подробности о модели памяти Java. Это сложные вопросы, которые мы рекомендуем изучить после того, как вы освоите концепции, изложенные в данной главе.

Пример проблемы, связанной с потокобезопасностью

Чтобы понять, что такое потокобезопасность, мы возьмем простой пример. Когда программа параллельно запускает несколько потоков, каждый из них потенциально делает что-то *одновременно* с другими запущенными потоками. Правда, не факт, что это произойдет. Если все-таки это случится, вам нужно предотвратить доступ к объекту, который изменяется другим потоком, потому что он может прочитать не-

согласованное состояние объекта. То же самое касается одновременных изменений. Гарантия того, что только один поток одновременно может получить доступ к блоку кода, называется *взаимным исключением*. Возьмем, например, следующий код:

```
class A {
    var aList: MutableList<Int> = ArrayList()
    private set

    fun add() {
        val last = aList.last() // эквивалент aList[aList.size - 1]
        aList.add(last + 1)
    }

    init {
        aList.add(1)
    }
}
```

Метод `add()` берет последний элемент списка, прибавляет единицу и добавляет результат в список. Каким будет ожидаемое поведение, если два потока попытаются одновременно выполнить метод `add()`?

Когда первый поток ссылается на последний элемент, у другого потока может быть время для выполнения всей строки `aList.add(last + 1)`[1]. В данном случае первый поток считает 2 для последнего элемента и добавит в список 3. Получившийся в итоге список будет выглядеть следующим образом: `[1, 2, 3]`. Возможен и другой сценарий. Если второй поток не успел добавить новое значение, оба потока прочитают одно и то же значение последнего элемента. Предполагая, что остальная часть выполняется без сбоев, мы получим результат `[1, 2, 2]`. Может случиться еще одна неприятность: если два потока попытаются добавить новый элемент в список одновременно, будет выброшено исключение `ArrayIndexOutOfBoundsException`.

В зависимости от чередования потоков результат может быть разным. Нет никакой гарантии, что мы вообще его получим. Это признаки того, что класс или функция не являются потокобезопасными и могут вести себя некорректно при обращении к ним из нескольких потоков.

Итак, что можно сделать, чтобы исправить это потенциально неправильное поведение? Есть три варианта:

1. Не делиться состоянием между потоками.
2. Делиться неизменяемым состоянием между потоками.
3. Изменить реализацию, чтобы несколько потоков могли использовать наш класс и получать предсказуемые результаты.

---

[1] На самом деле чередование потоков может происходить между строками байт-кода, а не только между строками обычного кода Java.

Существует несколько стратегий обеспечения потокобезопасности, у каждой из которых свои сильные стороны и недостатки, поэтому разработчику важно иметь возможность оценить свои варианты и выбрать тот, который лучше всего соответствует его потребностям.

Первый вариант относительно очевиден. Когда потоки могут работать с полностью независимыми наборами данных, риска обращения к одним и тем же адресам нет.

Второй вариант — использование неизменяемых объектов и коллекций. Неизменяемость — очень эффективный способ проектирования надежных систем. Если поток не может изменить объект, риска прочитать несогласованное состояние из другого потока просто не существует. В нашем примере можно было бы сделать список неизменяемым, но тогда потоки не смогли бы добавлять в него элементы. Это не означает, что данный принцип здесь неприменим. На самом деле это можно сделать, но мы вернемся к этому позже. Нужно упомянуть, что у неизменяемости есть потенциальный недостаток. Из-за копирования объектов ей требуется больше памяти. Например, всякий раз, когда потоку необходимо работать с состоянием другого потока, выполняется копия объекта состояния. При многократном и быстром выполнении неизменяемость способна увеличить объем памяти, что может стать проблемой (особенно в Android).

Третий вариант можно описать так: "Любой поток, выполняющий метод `add`, выполняется перед всеми последующими обращениями к этому методу из других потоков". Другими словами, обращения к методу `add` происходят последовательно, без чередования. Если ваша реализация обеспечивает вышеупомянутое утверждение, то проблем с потокобезопасностью не будет — класс считается потокобезопасным. В мире параллельного программирования предыдущий оператор называется *инвариантом*.

Инварианты

Чтобы должным образом сделать класс или группу классов потокобезопасными, нужно определить инварианты. Инвариант — это утверждение, которое всегда должно быть истинным. Независимо от того, как запланированы потоки, инвариант нельзя нарушать. В нашем случае это можно было бы выразить так (с точки зрения потока):

> при выполнении метода `add` я беру последний элемент списка, и когда добавляю его в список, я уверен, что вставленный элемент больше предыдущего на разницу в единицу.

Математически это можно было бы выразить следующим образом:

$$\text{list}[n] = \text{list}[n-1] + 1.$$

Мы с самого начала видели, что наш класс не является потокобезопасным. Теперь так можно сказать, потому что при выполнении в многопоточном окружении инвариант иногда нарушается или программа просто аварийно завершает работу.

Итак, что можно сделать, чтобы обеспечить соблюдение инвариантов? На самом деле, это сложный вопрос, но мы рассмотрим некоторые наиболее распространенные способы:

- мьютексы;
- потокобезопасные коллекции.

Мьютексы

Мьютексы позволяют предотвратить одновременный доступ к состоянию, которое может быть блоком кода или просто объектом. Такое взаимное исключение также называется *синхронизацией*. Объект, который называется *мьютексом*, гарантирует, что если взять его из потока, ни один другой поток не сможет войти в раздел, защищенный этим мьютексом. Когда поток пытается захватить мьютекс, который есть у другого потока, он блокируется: не может продолжить выполнение, пока блокировка не будет снята. Этот механизм относительно прост в использовании, поэтому разработчики часто обращаются к нему, когда сталкиваются с подобной ситуацией. К сожалению, это также похоже на открытие ящика Пандоры, когда речь идет о таких проблемах, как взаимоблокировки, состояние гонки и т. д. Этих проблем, которые могут возникнуть из-за неправильной синхронизации, так много, что полное их описание выходит далеко за рамки данной книги. Однако позже мы обсудим некоторые из них, например взаимоблокировки во взаимодействующих последовательных процессах.

Потокобезопасные коллекции

Потокобезопасные коллекции — это коллекции, к которым могут обращаться несколько потоков, сохраняя при этом согласованное состояние. `Collections.synchronizedList` — полезный способ сделать список потокобезопасным. Этот метод возвращает список, который оборачивает доступ к списку, переданному в качестве параметра, и регулирует параллельный доступ с помощью внутренней блокировки.

На первый взгляд выглядит интересно и у вас может возникнуть соблазн использовать этот код:

```
class A {
    var list =
        Collections.synchronizedList<Int>(object : ArrayList<Int?>() {
            init {
                add(1)
            }
    })

    fun add() {
        val last = list.last()
```

```
        list.add(last + 1)
    }
}
```

Для справки ниже приводится эквивалент, написанный на Java:

```
class A {
    List<Integer> list = Collections.synchronizedList(
        new ArrayList<Integer>() {{
            add(1);
        }}
    );

    void add() {
        Integer last = list.get(list.size() - 1);
        list.add(last + 1);
    }
}
```

У обеих реализаций есть проблема. Заметили ее?

Можно было бы объявить список следующим образом:

```
var list: List<Int> = CopyOnWriteArrayList(listOf(1))
```

что в Java эквивалентно:

```
List<Integer> list = new CopyOnWriteArray
List<>(Arrays.asList(1));
```

`CopyOnWriteArrayList` — это потокобезопасная реализация `ArrayList`, в которой все изменчивые операции, такие как `add` и `set`, реализованы путем создания новой копии базового массива. Поток *A* может безопасно перебирать список. Если тем временем поток *B* добавит элемент в список, будет создана новая копия, которую можно увидеть только из потока *B*. Само по себе это не делает класс потокобезопасным, потому что `add` и `set` защищены мьютексом. Такая структура данных полезна, когда мы перебираем ее чаще, чем изменяем, поскольку копирование всего базового массива может быть слишком затратным занятием. Обратите внимание, что существует также класс `CopyOnWriteArraySet`, представляющий собой реализацию интерфейса `Set`, но не реализацию `List`.

Мы исправили проблему с параллельным доступом, хотя наш класс по-прежнему не соответствует инварианту. В тестовом окружении мы создали два потока и запустили их. Каждый поток выполняет метод `add()` один раз в одном и том же экземпляре класса. В первый раз, когда мы запускали тест, после того как оба потока завершили свою работу, получившийся в результате список выглядел так: `[1, 2, 3]`. Любопытно, что мы выполняли один и тот же тест несколько раз, и иногда результат был таким: `[1, 2, 2]`. Это происходит по той же причине, что была продемонст-

рирована ранее: когда поток выполняет первую строку в методе add(), другой поток может выполнить весь метод до того, как первый поток продолжит выполнение оставшейся части. Видите, насколько пагубной может быть проблема синхронизации: вроде бы все хорошо, а программа вышла из строя. И легко можно ошибиться, даже на банальном примере.

Правильное решение:

```
class A {
    val list: MutableList<Int> = mutableListOf(1)

    @Synchronized
    fun add() {
        val last = list.last()
        list.add(last + 1)
    }
}
```

Эквивалент на Java:

```
public class A {
    private List<Integer> list = new ArrayList<Integer>() {{
        add(1);
    }};

    synchronized void add() {
        Integer last = list.get(list.size() - 1);
        list.add(last + 1);
    }
}
```

Как видите, фактически нам не нужно было синхронизировать список. Вместо этого следовало бы синхронизировать метод add(). Теперь, когда этот метод сначала выполняется потоком, другой поток блокируется при попытке выполнить его до тех пор, пока первый поток не сделает свою работу. Два потока не выполняют метод одновременно. Теперь инвариант соблюдается.

В этом примере показано, что "под капотом" класс может использовать потокобезопасные коллекции, не являясь при этом потокобезопасным. Считается, что класс или код потокобезопасны, если их инварианты никогда не нарушаются. Эти инварианты и то, как должен использоваться класс согласно их создателям, определяют политику, которая должна быть четко выражена в javadoc.

Это встроенный механизм Java принудительного взаимного исключения. Синхронизированный блок состоит из мьютекса и блока кода. В Java каждый Object можно использовать в качестве мьютекса. Синхронизированный метод — это синхронизированный блок, мьютекс которого — это экземпляр экземпляра класса. Когда поток входит в синхронизированный

блок, он захватывает мьютекс. А когда он выходит из блока, то освобождает его.

Также обратите внимание, что метод add можно объявить, используя оператор synchronized:

```
void add() {
    synchronized(this) {
        val last = list.last()
        list.add(last + 1)
    }
}
```

Поток не может войти в синхронизированный блок, чей мьютекс уже захвачен другим потоком. Как следствие, когда поток входит в синхронизированный метод, он не позволяет другим потокам выполнять какой-либо синхронизированный метод или любой блок кода, защищенный им (это также называется *встроенной блокировкой*).

Привязка к потоку

Еще один способ обеспечить потокобезопасность — убедиться, что только один поток владеет состоянием. Если состояние не видно другим потокам, то риск возникновения проблем, связанных с параллелизмом, просто отсутствует. Например, общедоступная переменная класса (где использование должно быть ограничено основным потоком) является потенциальным источником ошибок, поскольку разработчик (не знакомый с политикой потоков) может использовать эту переменную в другом потоке.

Непосредственным преимуществом привязки к потоку является ее простота. Например, если следовать соглашению, согласно которому каждый класс типа View должен использоваться только из основного потока, то можно избавить себя от синхронизации кода везде, где это возможно. Однако за это придется платить. Корректность клиентского кода теперь лежит на плечах разработчика, который использует наш код. В Android, как было показано в предыдущей главе, управление представлениями должно осуществляться только из UI-потока. Это форма привязки к потоку — пока вы не нарушаете правила, у вас не должно быть проблем с одновременным доступом к объектам, связанным с пользовательским интерфейсом.

Еще одна заслуживающая внимания форма привязки к потоку — это ThreadLocal. Экземпляр ThreadLocal можно рассматривать как провайдер некоего объекта. Этот провайдер гарантирует, что данный экземпляр объекта уникален для каждого потока. Другими словами, каждый поток владеет собственным экземпляром значения. Например:

```
private val myConnection =
        object : ThreadLocal<Connection>() {
```

```
        override fun initialValue(): Connection? {
            return DriverManager.getConnection(connectionStr)
        }
    }
```

Он часто используется в сочетании с соединениями с базой данных с помощью JDBC, которые не являются потокобезопасными. `ThreadLocal` гарантирует, что каждый поток будет использовать собственное соединение с помощью JDBC.

Конфликт потоков

Синхронизация между потоками — сложная вещь, потому что может возникнуть много проблем. Мы только что познакомились с потенциальными проблемами, связанными с потокобезопасностью. Есть еще одна опасность, которая может повлиять на производительность, — это *конфликт потоков*, с которым мы рекомендуем ознакомиться всем программистам. Рассмотрим этот пример:

```
class WorkerPool{
    private val workLock = Any() // В Java мы бы использовали new Object()

    fun work() {
        synchronized(workLock) {
            try {
                Thread.sleep(1000) // имитация задачи с интенсивным вычислением
            } catch (e: Exception) {
                e.printStackTrace()
            }
        }
    }

   // другие методы, которые могут использовать внутреннюю блокировку
}
```

Итак, у нас есть класс `WorkerPool`, который контролирует работу, выполняемую рабочими потоками таким образом, что только один рабочий поток за раз может выполнять работу в методе `work`. Эта ситуация, с которой вы можете столкнуться, когда фактическая работа связана с использованием непотокобезопасных объектов и разработчик решил разобраться с этой проблемой с помощью данной политики блокировки. Вместо `synchronized(this)` для метода `work` был создан специальный мьютекс, потому что другие методы теперь могут вызываться рабочими потоками без взаимного исключения. По этой причине блокировка была названа в честь связанного с ней метода.

Если запущены несколько рабочих потоков и они вызывают метод `work`, то эти потоки будут бороться за один и тот же мьютекс. В конце концов, в зависимости от

чередования потоков, рабочий процесс блокируется, потому что другой поток захватил мьютекс. Это не проблема, если время, затраченное на ожидание мьютекса, значительно меньше, чем оставшееся время выполнения. В противном случае у нас конфликт потоков. Бóльшую часть времени потоки проводят в ожидании друг друга. Затем операционная система может превентивно остановить некоторые потоки, чтобы другие потоки, находящиеся в состоянии ожидания, могли возобновить выполнение. Это еще больше усугубляет ситуацию, поскольку переключение контекста между потоками — удовольствие не бесплатное. Если такое происходит часто, это может привести к снижению производительности.

Будучи разработчиком, вы всегда должны избегать конфликтов между потоками, т. к. они могут быстро снизить пропускную способность, и это будет иметь последствия не только для затронутых потоков, поскольку скорость переключения контекста, вероятно, увеличится, что само по себе влияет на общую производительность.

Один из наиболее эффективных способов предотвратить такую ситуацию — избегать блокирующих вызовов, о чем пойдет речь в следующем разделе.

Сравнение блокирующего и неблокирующего вызовов

До сих пор мы знали, что поток может быть заблокирован при попытке захватить мьютекс, удерживаемый другим потоком. Тогда функция, которая привела к блокировке потока, — это *блокирующий вызов*. Даже если мьютекс можно получить сразу же, тот факт, что вызов потенциально может быть заблокирован, делает его блокирующим. Но это всего лишь частный случай. На самом деле есть два других способа блокировки потока. Первый заключается в выполнении вычислений с интенсивным использованием ЦП — его также называют *счетной* задачей. Второй — дождаться ответа аппаратного обеспечения. Например, это происходит, когда сетевой запрос заставляет вызывающий поток ожидать ответа от удаленного сервера — тогда мы говорим о задаче, ограниченной скоростью ввода/вывода данных[2].

Все остальное, что вызывает быстрый возврат вызова, считается *неблокирующим*.

Если вы собираетесь выполнить блокирующий вызов, не следует делать это из основного потока (также называемого UI-потоком в Android)[3]. Это связано с тем, что

---

[2] Операции ввода-вывода не обязательно блокируются. Неблокирующий ввод-вывод существует, хотя его гораздо сложнее анализировать. Android Link предупредит вас, когда вы выполняете HTTP-запрос в основном потоке, но другие задачи ввода-вывода, такие как чтение файла или запрос к базе данных, этого не делают. Это может быть даже преднамеренной и общепринятой практикой, если делается под чрезвычайно вдумчивым и тщательным наблюдением; хотя такое и возможно, это должно быть редким исключением из стандарта.

[3] Даже в случае с рабочими потоками, выполняющими длительную задачу, например работу с 8-мегапиксельными изображениями, эти блокирующие вызовы могут блокировать пакеты задач, которые ожидает UI-поток.

данный поток запускает цикл обработки событий, который обрабатывает события касания, и все связанные с пользовательским интерфейсом задачи, такие как анимация. Если основной поток блокируется неоднократно и на время, превышающее несколько миллисекунд, это влияет на отзывчивость и является причиной появления ошибок *application not responding* (ANR — приложение не отвечает).

Неблокирующие вызовы — это один из строительных блоков адаптивного приложения. Теперь нужно определить шаблоны, использующие эту технику. Очереди работ — один из них, и в этой книге мы столкнемся с различными формами очередей работ.

Чаще всего термины *"синхронный"* и *"асинхронный"* используются как синонимы слов *"блокирующий"* и *"неблокирующий"*. Хотя концептуально это близкие понятия, использование, например, слова "асинхронный" вместо "неблокирующий" зависит от контекста. Асинхронные вызовы обычно включают в себя концепцию функции обратного вызова, хотя это не обязательно относится к понятию "неблокирующий".

Очереди работ

Взаимодействие между потоками и, в частности, передача работы из одного потока в другой широко используются в Android. Это реализация паттерна проектирования *"производитель — потребитель"*. Применительно к потокам производитель в данном контексте — это поток, генерирующий данные, которые должен обработать поток-потребитель. Вместо взаимодействия производителя с потребителем напрямую через общее изменяемое состояние между ними используется очередь работы, создаваемая производителем. Это отделяет производителя от потребителя, но это не единственное преимущество, в чем мы скоро убедимся. Часто очередь работает по принципу FIFO (first in, first out — первым пришел, первым ушел)[4].

Понять вышесказанное проще, если рассматривать `Queue` как очередь из кинозрителей. Когда приходит первый зритель, он встает в начало очереди. Каждый последующий зритель встает за тем, кто стоит последним. Когда двери открываются и зрители могут войти, сначала пускают первого человека из очереди, затем следующего и т. д., пока вся очередь не опустеет.

Производитель помещает объект в начало очереди, а потребитель помещает его в конец очереди. Метод `put` может быть блокирующим вызовом, но если можно доказать, что бóльшую часть времени он, по сути, не выполняет блокировку (а когда делает это, то на короткое время), то мы получаем очень эффективный способ передачи работы от производителя потребителю неблокирующим образом (с точки зрения производителя), как показано на рис. 5.1.

[4] Не все очереди работ используют такую структуру данных. Некоторые из них более сложные, например `MessageQueue`.

На практике поставленные в очередь объекты часто представляют собой экземпляры `Runnable`, отправляемые фоновым потоком и обрабатываемые основным потоком. Кроме того, дело не ограничивается одним производителем и одним потребителем. Несколько производителей могут отправлять работу в очередь одновременно с несколькими потребителями, берущими работу из очереди. Из этого следует, что очередь должна быть потокобезопасной[5].

Рис. 5.1. Производитель — потребитель

Не путайте очередь со стеком, где вместо FIFO используется принцип LIFO (last in, first out — последним пришел, первым ушел).

На семантическом уровне можно представить стек в виде стопки блинов. Когда повара́ на кухне пекут блины, они кладут свежеиспеченные блины на вершину стопки. Когда посетитель закусочной ест их, он также берет их сверху.

Противодавление

Представьте теперь, что производитель намного быстрее потребителя. Тогда рабочие объекты будут скапливаться в очереди. Если очередь окажется неограниченной, мы рискуем исчерпать ресурсы памяти и, возможно, получить неисправимое исключение: приложение может аварийно завершить работу. Хотя это не только раздражает и неприятно для пользователя, но и в случае необработанной ошибки, подобной этой, вы почти наверняка потеряете всю имеющуюся информацию о состоянии. Если вы не позаботитесь о том, чтобы быть в курсе этого обстоятельства и отреагировать на него, то можете столкнуться с внезапным прекращением работы без возможности выполнить очистку, которую вы обычно выполняете. В Android, если экземпляр Bitmap больше не используется, можно применить метод `recycle`, чтобы помечать любое базовое выделение памяти как уничтоженное и утилизированное сборщиком мусора. При неаккуратном выходе из системы у вас может не оказаться возможности сделать это, и вы рискуете получить утечку данных.

В данном случае разумно будет использовать ограниченную очередь. Но что должно произойти, когда очередь заполнится, а производитель попытается поместить объект?

Мы вернемся к этому, когда будем изучать сопрограммы, но, поскольку сейчас мы говорим только о потоках, ответ таков: он должен блокировать поток производите-

[5] Даже при наличии одного производителя и одного потребителя очередь должна быть потокобезопасной.

ля до тех пор, пока потребитель не возьмет хотя бы один объект из очереди, хотя эта блокировка должна быть частью архитектуры и предвидеть любое обстоятельство или логическую ветвь, которые могут привести пользователя к этому месту в программе. Да, блокировка потока не кажется полезной, но заблокированный производитель позволяет потребителю наверстать упущенное и расчистить достаточно места в очереди, чтобы освободить производителя. Этот механизм известен как *противодавление* — способность потребителя, который не успевает за поступающими данными, замедлять работу производителя. Это очень мощный способ проектирования надежных систем. Реализация противодавления показана в примере 5.1.

Пример 5.1. Пример противодавления

```
fun main() {
    val workQueue = LinkedBlockingQueue<Int>(5) // размер очереди равен 5

    val producer = thread {
        while (true) {
            /* Вставляет один элемент в конец очереди, ожидая,
               если это необходимо, чтобы освободилось место. */
            workQueue.put(1)
            println("Producer added a new element to the queue")
        }
    }

    val consumer = thread {
        while (true) {
            // У нас медлительный потребитель - он "спит" на каждой итерации.
            Thread.sleep(1000)
            workQueue.take()
            println("Consumer took an element out of the queue")
        }
    }
}
```

Начиная с Java 7 для этой цели используется семейство очередей `BlockingQueue` — это интерфейс, и его реализации варьируются от односторонней очереди с `LinkedBlockingQueue` до двусторонней очереди с `LinkedBlockingDequeue` (существуют и другие реализации). Результат примера 5.1:

```
Producer added a new element to the queue
Producer added a new element to the queue
Producer added a new element to the queue
Producer added a new element to the queue
```

```
Producer added a new element to the queue
Consumer took an element out of the queue
Producer added a new element to the queue
Consumer took an element out of the queue
Producer added a new element to the queue
...
```

Видно, что производитель быстро заполнил очередь пятью элементами. Затем на шестой попытке добавить новый элемент он блокируется, т. к. очередь заполнена. Секундой позже потребитель берет элемент из очереди, освобождая производителя, который теперь может добавить новый элемент. На данный момент очередь заполнена. Производитель пытается добавить новые элементы, но снова блокируется. И опять, секундой позже, потребитель берет один элемент — и т. д.

Важно отметить, что вставка элемента в `BlockingQueue` не обязательно приводит к блокировке. Если вы используете метод `put`, то он осуществляет блокировку при заполнении очереди. Поскольку этот метод *может* блокировать, мы говорим, что это блокирующий вызов. Однако для добавления нового элемента доступен еще один метод — `offer`, который пытается тотчас же добавить новый элемент и возвращает логическое значение независимо от того, была ли операция выполнена успешно.

Поскольку метод `offer` не блокирует базовый поток и возвращает `false` только тогда, когда очередь заполнена, мы говорим, что этот метод является неблокирующим.

Если бы в примере 5.1 вместо метода `put` мы использовали `offer`, то производитель не был бы заблокирован, а в выводе появилась бы фраза: `Producer added a new element to the queue`. Противодавления не было бы вообще — не делайте этого!

Метод `offer` может быть полезен в ситуациях, когда есть риск потери работы или блокировка потока производителя не подходит. Те же рассуждения применяются при извлечении объекта из очереди с помощью `take` и `poll`, которые являются блокирующим и не блокирующим методами соответственно.

И наоборот, если потребитель быстрее производителя, то очередь в конечном счете становится пустой. В случае с `BlockingQueue` использование метода `take` с потребителем приведет к блокировке до тех пор, пока производитель не добавит новые элементы в очередь. Таким образом, в данном случае потребитель замедляется, чтобы соответствовать скорости производителя.

Резюме

- Говорят, что класс или код потокобезопасны, если их инварианты никогда не нарушаются. Таким образом, потокобезопасность всегда относится к политике, которая должна быть четко определена в javadoc класса.

- "Под капотом" класс может использовать потокобезопасные структуры данных, не являясь при этом потокобезопасным.
- Избегайте конфликтов между потоками или сокращайте их число, насколько это возможно. Конфликт между потоками часто является следствием плохой стратегии блокировки. Эффективный способ уменьшить этот риск заключается в том, чтобы выполнять неблокирующие вызовы, когда это возможно.
- Очереди работ — это паттерн, с которым часто сталкиваются в Android и других платформах, таких как серверные службы. Он упрощает способ, которым производитель (например, UI-поток) отправляет задачи потребителям (фоновым потокам). Потребители обрабатывают задачи, когда могут. Когда задача завершается, потребитель может использовать другую очередь работ, чтобы отправить исходному производителю результат своей работы.
- Ограниченная очередь `BlockingQueue` блокирует операцию, осуществляемую методом `put`, если она заполнена. Таким образом, слишком быстрый производитель в конечном счете блокируется, что дает потребителям возможность наверстать упущенное. Это реализация противодавления, у которой имеется один существенный недостаток: поток-производитель может быть заблокирован. Возможно ли получить противодавление, не блокируя поток-производитель? Да, и вы увидите это в *главе 9*.

ГЛАВА 6

Организация параллелизма с использованием обратных вызовов

Идиоматический способ организации параллелизма в Kotlin — использование сопрограмм. Однако в течение некоторого времени в Java это можно было делать с помощью потоков и обратных вызовов. Так зачем же нам сопрограммы?

Чтобы ответить на этот вопрос, мы вернемся к типичной реализации Kotlin в Android и обсудим подводные камни использования потоков. Знание слабых мест традиционного подхода является ключом к пониманию мотивации проектирования сопрограмм.

В приложениях Android длительные задачи не должны выполняться в UI-потоке, как было показано в предыдущей главе. Если вы заблокируете основной поток — UI-поток, то у вашего приложения может не хватить ресурсов, необходимых для отрисовки экрана или его соответствующего обновления. Фактически, статический анализатор кода будет жаловаться, если вы попытаетесь выполнить очевидный вызов ввода-вывода (например, установить соединение по протоколу HTTP) в UI-потоке.

Приложение работает гладко, когда основной поток выполняет все свои задачи менее чем за время кадра, которое на большинстве устройств составляет 16 мс. Это довольно короткий промежуток времени, и все блокирующие вызовы, как и сетевые запросы (блокирующий ввод-вывод), должны выполняться в фоновом потоке[1].

Когда вы делегируете задачу другому потоку, то обычно вызываете функцию, которая запускает асинхронное задание. В некоторых случаях это что-то из серии "выстрелил и забыл", но обычно вы ждете результата — и вам нужно его обработать. Это делается путем предоставления функции, которая будет вызываться после завершения задания. Она называется *функцией обратного вызова*. Функция обратного вызова часто принимает аргументы, поэтому фоновый поток обычно вызывает такую функцию с результатом задания. Выполнение вычислений, вызывающих произвольную или внедренную функцию после завершения, называется *паттерном обратного вызова*. Использование функций обратных вызовов достаточно эф-

[1] Неблокирующий ввод-вывод с использованием `java.nio.channels.SocketChannel` можно выполнять в UI-потоке, не блокируя его. Однако бóльшую часть времени при работе с вводом-выводом вы будете использовать блокирующие API, такие как `java.io.InputStream`.

фективно, хотя здесь есть некоторые ограничения и недостатки. В качестве иллюстрации мы реализуем простой, но реалистичный пример на языке Kotlin. Используя эти функции, сопрограммы решают все проблемы, но, прежде чем переходить непосредственно к сопрограммам, важно понять, какую проблему они намерены решить.

Пример: функция обработки покупок

Предположим, вы пишете платную функцию приложения для Android. После регистрации пользователя вы проверяете список покупок, которые этот пользователь уже сделал, а затем работаете с ним. Чтобы получить этот список, воспользуемся объектом `BillingClient`. Обратите внимание, что мы говорим не о фактическом объекте `BillingClient`, предоставляемом фреймворком Android, `com.android.billingclient.api.BillingClient`. Мы используем собственную, гораздо более простую версию базовой концепции, как показано в следующем коде:

```
interface BillingClient {
    fun interface BillingCallback {
        fun onInitDone(provider: PurchasesProvider?)
    }

    /* Реализации должны быть неблокирующими */
    fun init(callback: BillingCallback)
}
```

Типичный поток задач будет таким:

1. Инициализируем подключение к `BillingClient`. Подождите, пока он будет готов — функция обратного вызова предоставляет `PurchasesProvider` или `null` в случае ошибки. Пока мы не будем обрабатывать ошибки.
2. Используем возвращенный `PurchasesProvider` для асинхронного получения пользовательского списка покупок. Программа станет ждать ответа, который будет содержать список и, возможно, некоторые дополнительные метаданные.
3. Реагируем на эту новую информацию; можно показать список покупок с пользовательским интерфейсом, чтобы предоставить еще больше подробностей или запросить статус, отменить товар в заказе и т. д.

В дальнейшем мы будем называть предыдущий поток *логикой*.

Как видите, это всего лишь интерфейс с одним методом, принимающим `BillingCallback` в качестве входных данных. `BillingCallback` объявляется в интерфейсе `BillingClient`, поскольку эта функция обратного вызова используется только в `BillingClient`. Когда интерфейс объявляется в классе или интерфейсе, он сообщает вам о связи между классом и интерфейсом: автор предполагал, что класс не должен зависеть от другой сущности, чтобы предоставить интерфейс. Это позволяет избежать риска нарушения совместимости между классом и интерфейсом. Они связаны,

и при отправке BillingClient вы также отправляете BillingCallback. Обратите внимание, что мы применяем новую функцию interface из Kotlin версии 1.4 вместо классического интерфейса. Это позволит использовать краткий синтаксис, когда мы предоставим реализации. Кроме того, в документации метода init сказано, что реализации должны быть неблокирующими. Если вы не читали предыдущую главу, то учтите: это означает, что какой бы поток ни вызывал этот метод, он не блокируется, ожидая пока метод не вернет управление.

Также в следующем коде показан интерфейс PurchasesProvider:

```
interface PurchasesProvider {
    fun interface PurchaseFetchCallback {
        fun onPurchaseFetchDone(purchases: List<String>)
    }

    fun fetchPurchases(user: String, callback: PurchaseFetchCallback)
}
```

Пока предположим, что мы предоставляем эти абстракции и их реализации. Несмотря на то что в реальном приложении будут использоваться классы, предоставляемые фреймворком, важной частью этого примера является бизнес-логика, а не реализация BillingClient и PurchasesProvider.

Мы надеемся, что будучи разработчиком приложений для Android, вы знакомы с основными понятиями ViewModel из Android Jetpack; но если нет — не волнуйтесь, потому что детали работы ViewModel не являются предметом данного обсуждения. Даже без ViewModel, у вас, вероятно, есть какая-то версия MVC, MVP или MVVM, каждая из которых в значительной степени следует тому же паттерну. Представление занимается презентацией, модель — логикой, а контроллер или компонент ViewModel являются связующим звеном между ними и служат сетью, которая позволяет им общаться. Важная часть — это реализация *логики* в компоненте ViewModel. Все остальное — это код контекста или фреймворка, но, тем не менее, это тоже важно.

На рис. 6.1 показана целевая архитектура.

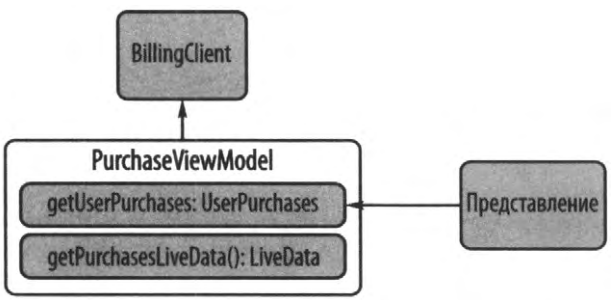


Рис. 6.1. Архитектура Model-View-ViewModel

Теперь предположим, что вы структурировали свое приложение в соответствии с архитектурой с *одной активностью*[2]. Представление должно быть фрагментом, отображающим покупки текущего пользователя. При проектировании следует учитывать жизненный цикл фрагмента. В любой момент устройство можно повернуть, и фрагмент будет воссоздан. Пользователь может вернуться назад, а фрагмент может быть помещен в стек переходов назад, если не был удален.

Именно здесь в игру вступает `LiveData`, компонент с учетом жизненного цикла. Каждый раз, при создании, он запрашивает экземпляр `PurchaseViewModel`. Позже мы подробнее объясним, как это работает.

Создание приложения

В этом разделе мы покажем вам типичную реализацию в приложении для Android. Если вы уже знакомы с ней, можете сразу перейти к следующему разделу, где обсуждается реализация *логики*.

Компонент ViewModel

Таким образом, бизнес-логика реализована внутри `ViewModel` (пример 6.1). Компонент ViewModel требует, чтобы экземпляр `BillingClient` был внедрен в конструктор[3] каким-то другим компонентом, и скоро вы это увидите. `BillingClient` — это зависимость `ViewModel`, а `PurchaseProvider` — это зависимость `BillingClient`.

Представление, взаимодействующее с `ViewModel`, запускает метод `getUserPurchases` (который мы еще не реализовали) в методе чтения свойства `PurchasesLiveData`. Возможно, вы заметили, что тип свойства `PurchasesLiveData` — это `LiveData`, в то время как приватное теневое свойство `_purchases` — это `MutableLiveData`. Это объясняется тем, что `ViewModel` должен быть единственным компонентом для изменения значения `LiveData`. Таким образом, для клиентов `ViewModel` доступен только тип `LiveData`, как показано в примере 6.1.

---

[2] Одна активность и несколько фрагментов.

[3] Разработка интерфейсов, а не реальных реализаций, улучшает тестируемость и переносимость кода. В тестовом окружении фактические реализации зависимостей можно менять на собственные имитированные реализации. Что касается переносимости, предположим, что у вас есть интерфейс `AnalyticsManager`, предоставляющий методы, которые вы реализуете для уведомления службы аналитики. Учитывая, что надежное аналитическое программное обеспечение, используемое как услуга (SaaS), с информационными панелями, визуализацией больших объемов данных и авторизацией само по себе является непростой работой, большинство разработчиков приложений будут использовать стороннюю библиотеку для обработки этой части потока. Если, например, вы переходите от одного провайдера к другому, то пока вы *компонуете* свои взаимодействия в соответствии с интерфейсом `AnalyticsManager`, ваш клиентский код не будет затронут, изменен или потенциально не сможет привести к новой ошибке; все, что обновляется, — это бизнес-логика реализации `AnalyticsManager`.

Пример 6.1. Класс PurchasesViewModel

```
class PurchasesViewModel internal constructor(
    private val billingClient: BillingClient,
    private val user: String
) : ViewModel() {
    private var _purchases = MutableLiveData<UserPurchases>()

    private fun getUserPurchases(user: String) {
        // нужно реализовать
    }

    val purchasesLiveData: LiveData<UserPurchases>
        get() {
            getUserPurchases(user)
            return _purchases
        }
     interface BillingClient { /* удалено для краткости*/ }

     interface PurchasesProvider { /* удалено для краткости */ }
}
```

Мы почти закончили — не хватает только представления.

Представление

В нашей архитектуре представление (компонент View) — это `Fragment`. Как видно из следующего кода, представление зависит от компонента ViewModel. Здесь показано, как использовать компонент ViewModel в представлении:

```
class PurchasesFragment : Fragment() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        /* Создаем ViewModel при первом создании этого фрагмента.
           Воссозданный фрагмент получает тот же экземпляр ViewModel
           после поворота устройства. */
        val factory: ViewModelProvider.Factory = PurchaseViewModelFactory()
❶
        val model by viewModels<PurchasesViewModel> { factory }
❷
        model.purchasesLiveData.observe(this) { (_, purchases) ->
❸
```

```
            // Обновляем пользовательский интерфейс
            println(purchases)
        }
    }
}
```

Каждый раз, когда фрагмент создается, он подписывается на обновления `UserPurchases`, следуя трем этапам.

❶ Создание фабрики для `ViewModel` (помните, у `ViewModel` есть зависимости, и, конечно же, фрагмент не несет ответственности за их поставку). Строго говоря, эту фабрику не следует создавать внутри фрагмента, т. к. теперь она тесно связана с нашим фрагментом — `PurchasesFragment` всегда использует `PurchaseViewModelFactory`. В тестовом окружении, где представление нужно тестировать отдельно, это будет проблемой. Таким образом, фабрику нужно внедрить внутрь фрагмента, используя фреймворк внедрения зависимостей, или сделать это вручную. Чтобы было проще, мы решили создать ее здесь, внутри фрагмента. Для справки фабрика `ViewModel` показана в примере 6.2.

❷ Экземпляр `PurchasesViewModel` мы получаем из функции `viewModels`. Это рекомендуемый способ получения экземпляра `ViewModel`.

❸ Наконец, экземпляр `LiveData` извлекается из `ViewModel` и *отслеживается* экземпляром `Observable` с использованием одноименного метода ("observe"). В данном примере наблюдатель — это всего лишь лямбда-функция, которая выводит список покупок в консоль. В рабочем приложении обычно запускается обновление всех связанных представлений внутри фрагмента.

У `ViewModel` также есть собственный жизненный цикл, который зависит от того, привязан ли компонент ViewModel к фрагменту или активности. В данном примере он привязан к фрагменту. Это можно определить по использованию `viewModels<..>`. Если бы мы указали `ActivityViewModels<..>`, то компонент ViewModel был бы привязан к активности.

Будучи привязанным к фрагменту, компонент ViewModel сохраняется после поворота устройства, но удаляется, если больше не используется (например, когда все фрагменты, которые были привязаны к фрагменту, удаляются, за исключением вращения устройства). Если бы он был привязан к активности, то при повороте устройства просуществовал бы дольше нее, но был бы удален в любой другой ситуации, при которой активность удаляется.

Поскольку `ViewModel` сохраняется за счет изменения конфигурации, что удаляет и воссоздает вмещающую активность, он никогда не должен ссылаться на представление, экземпляр `Lifecycle` или любой экземпляр класса, который может содержать ссылку на контекст активности. Однако он может ссылаться на контекст `Application`.

Если посмотреть на фактический код `BillingClient`, то можно увидеть, что создание `BillingClient.Builder` требует предоставить контекст. Это может быть контекст активности, потому что "под капотом" конструктор вызывает метод

context.getApplicationContext(), а это единственная ссылка на контекст, хранящаяся в BillingClient. ApplicationContext остается неизменным в течение всего жизненного цикла Application. Поэтому у вас не будет утечки памяти из-за ссылки на ApplicationContext где-то в приложении. По этой причине можно безопасно ссылаться на BillingClient в ViewModel.

Как показано в примере 6.2, зависимости ViewModel создаются в классе PurchaseViewModelFactory.

Пример 6.2. Класс PurchaseViewModelFactory

```
class PurchaseViewModelFactory : ViewModelProvider.Factory {
    private val provider: PurchasesProvider = PurchasesProviderImpl()
    private val billingClient: BillingClient = BillingClientImpl(provider)
    private val user = "user" // Вход из службы регистрации

    override fun <T : ViewModel?> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(PurchasesViewModel::class.java)) {
            return PurchasesViewModel(billingClient, user) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

BillingClientImpl — это реальная реализация показанного ранее интерфейса BillingClient (примеры 6.3, 6.4).

Пример 6.3. Класс BillingClientImpl

```
class BillingClientImpl(private val purchasesProvider: PurchasesProvider) :
BillingClient {
    private val executor = Executors.newSingleThreadExecutor()

    override fun init(callback: BillingCallback) {
        /* здесь выполняются асинхронные операции */
        executor.submit {
            try {
                Thread.sleep(1000)
                callback.onInitDone(purchasesProvider)
            } catch (e: InterruptedException) {
                e.printStackTrace()
            }
        }
    }
}
```

Пример 6.4. Класс PurchasesProviderImpl

```
class PurchasesProviderImpl : PurchasesProvider {
    private val executor =
        Executors.newSingleThreadExecutor()

    override fun fetchPurchases(
        user: String,
        callback: PurchaseFetchCallback
    ) {
        /* здесь выполняются асинхронные операции */
        executor.submit {
            try {
                // Имитация блокирующего ввода-вывода
                Thread.sleep(1000)
                callback.onPurchaseFetchDone(
                    listOf("Purchase1", "Purchase2")
                )
            } catch (e: InterruptedException) {
                e.printStackTrace()
            }
        }
    }
}
```

Чтобы соответствовать архитектуре приложения, которую мы установили, методы `init` и `fetchPurchases` должны быть неблокирующими. Этого можно добиться с помощью фонового потока. Из соображений эффективности (*см. следующий раздел*), возможно, вы не захотите создавать новый поток каждый раз при подключении к `BillingClient`. Вместо этого можно использовать пул потоков, который можно создать с помощью экземпляров `ThreadPoolExecutor` напрямую. Кроме того, многие распространенные конфигурации доступны через фабричные методы `java.util.concurrent.Executors`. При использовании метода `Executors.newSingleThreadExecutor()` в вашем распоряжении есть один выделенный поток, который можно эксплуатировать повторно для каждого асинхронного вызова. Возможно, вы считаете, что `PurchasesProviderImpl` и `BillingClientImpl` должны использовать один и тот же пул потоков. Решать вам, хотя для краткости мы здесь этого не делали. В рабочем приложении может быть несколько экземпляров `ThreadPoolExecutor`, которые обслуживают разные его части.

Если посмотреть на то, как функции обратного вызова используются в этих реализациях, можно увидеть, что они вызываются сразу после метода `Thread.sleep()` (который имитирует вызов блокирующего ввода-вывода). Если они явно не отправле-

ны в основной поток, (как правило, через экземпляр класса Handler или метод postValue экземпляра LiveData), то эти функции вызываются в контексте фонового потока. Очень важно знать способ взаимодействия между контекстами потока, в чем вы убедитесь, прочитав следующий раздел.

Нужно быть в курсе того, какой поток выполняет предоставленную функцию обратного вызова, т. к. это зависит от реализации. Иногда эта функция выполняется асинхронно в вызывающем потоке, тогда как в фоновом потоке она может выполняться синхронно.

Реализация логики

Теперь, когда все необходимые компоненты на месте, можно реализовать *логику*. Шаги показаны в примере 6.5.

Пример 6.5. Логика

```
private fun getUserPurchases(user: String) {
    billingClient.init { provider ->                    ❶
        // вызывается из фонового потока
        provider?.fetchPurchases(user) { purchases -> ❷
            _purchases.postValue(UserPurchases(user, purchases))
        }
    }
}
```

❶ Вызовите billingClient.init и предоставьте функцию обратного вызова, которая будет вызываться всякий раз при завершении процесса инициализации клиента. Если клиент предоставляет непустой экземпляр PurchasesProvider, переходите к следующему этапу.

❷ На данном этапе у вас есть экземпляр PurchasesProvider, готовый к использованию. Вызовите метод fetchPurchases, указав текущего пользователя в качестве первого параметра и функцию обратного вызова, которая должна быть выполнена после того, как провайдер сделает свою работу. Посмотрите внимательно на содержимое функции:

```
_purchases.postValue(UserPurchases(user, purchases))
```

В экземпляре MutableLiveData используется либо метод setValue, либо метод postValue. Разница между ними состоит в том, что метод setValue разрешено использовать только в том случае, если он вызывается из основного потока. В противном случае, используя метод postValue, мы добавляем новое значение в очередь, которое MutableLiveData обработает в следующем кадре основного потока. Это реализация паттерна *"очередь работ"* (*см. разд. "Очереди работ" главы 5*) и потокобезопасный способ присвоить MutableLiveData новое значение.

Обсуждение

Вот и все. Все работает или, по крайней мере, соответствует спецификациям. Предлагаем сделать небольшой шаг назад и посмотреть на общую картину. Какова структура функции `getUserPurchases`? Она состоит из вызова функции, которому предоставляется другая функция, сама вызывающая функцию, предоставляемую другой функцией... Похоже на русские матрешки. Такой код уже сложнее понять, а добавление обработки исключений может быстро превратить его в "ад вложений" (рис. 6.2). Чтобы логика нашего примера была простой и легкой для понимания, мы опустили крайние случаи, когда некоторые вызовы API дают сбой; например, проблемы сетевого взаимодействия или ошибки авторизации делают фоновую работу ввода-вывода неустойчивой и склонной к сбоям, и промышленный код должен быть в состоянии справиться с этим.

Рис. 6.2. Использование функций обратного вызова

Код, определяющий, что происходит после ответа `BillingClient` (обратный вызов 2), *включается* в код первой функции обратного вызова. Если вы решите встроить весь этот код, как мы это сделали в примере 6.5, у вас будет несколько уровней отступов, которые будут быстро расти по мере усложнения решаемой задачи. С другой стороны, если вы решите инкапсулировать первую функцию обратного вызова в отдельную функцию, то уменьшите уровень отступа функции `getUserPurchases` и ее очевидную сложность. В то же время вы увеличиваете количество направлений, которым нужно следовать, чтобы полностью понять бизнес-логику.

Это первый недостаток кода, использующего функции обратного вызова. Если не соблюдать осторожность и не следовать продуманной архитектуре, код будет быстро усложняться, и, возможно, его станет трудно сопровождать. Кто-то, вероятно, решит, что даже при соблюдении всех мер предосторожности такой путь опасен. Будучи разработчиками, мы стремимся создать систему, с которой сможем работать мы и наши коллеги.

Используя `CompletableFuture` или другую подобную библиотеку, например RxJava, можно переписать функцию `getUserPurchases`:

```
private void getUserPurchases(String user) {
    billingClient.initAsync()
```

```
        .thenCompose { provider ->
            fetchPurchasesAsync(provider, user)
        }
        .thenAccept { purchases ->
            this.purchases.postValue(...)
        }
}
```

Выглядит немного чище, здесь нет вложенных отступов, и этот код даже обрабатывает исключения должным образом. Однако видно, что в нем используются комбинаторы `thenCompose` и `thenAccept`, которые работают с `CompletableFuture<T>`. Хотя в нашем простом примере присутствуют только два комбинатора, их много, и каждый из них предназначен для определенной цели. Некоторые сочтут кривую обучения другому, незнакомому паттерну и API недостатком этого шаблона.

Структурированный параллелизм

Теперь представьте, что некоторые вызовы API довольно затратны в вычислительном отношении. Например, пользователь приложения переходит к представлению, которое инициирует некоторые из этих вызовов, но поскольку содержимое не загружается мгновенно, он теряет терпение, нажимает кнопку **Назад** и начинает новую серию операций в другой части приложения. В такой ситуации вы не хотите, чтобы дорогостоящие вызовы API продолжали работу, поскольку они могут создать ненужную нагрузку на внутренние серверы или даже на само приложение. Кроме того, что произойдет, если пользовательского интерфейса, который должен обновляться при срабатывании функции обратного вызова, больше не существует? Вероятно, в лучшем случае вы получите исключение `NullPointerException`, а в худшем — утечку памяти. Вместо этого отменим процедуру, инициализированную внутри компонента ViewModel. Как это сделать? Нужно прослушивать конкретное событие жизненного цикла из событий завершения жизненного цикла фрагмента: `onStop`, `onPause` или `onDestroy`. В данном случае вы, вероятно, захотите сделать это в методе `onStop`, который будет запускаться непосредственно перед удалением ресурсов. Метод `onPause` срабатывает каждый раз, когда приложение работает в фоновом режиме в пользу входящего вызова или при переключении между приложениями, а метод `onDestroy` появляется чуть позже, чем нужно. Когда срабатывает событие `onStop`, вам необходимо уведомить компонент ViewModel о том, что он должен остановить всю фоновую обработку. Для этого требуется потокобезопасный способ прерывания потоков. Изменчивое логическое значение `isCancelled` будет проверяться в обратных вызовах, чтобы можно было решить, должны ли они продолжать работу или нет. Так что это определенно возможно, но выглядит громоздко и ненадежно. Что, если эта отмена была сделана автоматически? Представьте, что фоновая обработка была привязана к жизненному циклу компонента ViewModel. В тот момент, когда компонент ViewModel удаляется, вся фоновая обработка отменяется. И это не сказка, у этой концепции даже есть имя — *структурированный параллелизм*.

Утечка памяти

У автоматической отмены "висячих" фоновых потоков есть еще одно преимущество: риск утечки памяти снижается. Функция обратного вызова может содержать ссылку на компонент, у которого либо есть жизненный цикл, либо он является потомком компонента, у которого тот есть. Если этот компонент подходит для сбора мусора, в то время как ссылка на этот компонент существует в каком-то работающем потоке, то память нельзя восстановить, и происходит утечка. Использование `LiveData`, как в предыдущем примере, безопасно, даже если вы не отменяете фоновые задачи. Тем не менее, говоря в более общих чертах, не стоит оставлять задачи работающими впустую.

Отмена не единственная возможная вещь, которая может пойти не так. Существуют и другие ловушки при использовании потоков в качестве примитивных типов для асинхронных вычислений (которые мы будем называть *моделью многопоточного выполнения*), и мы рассмотрим их в следующем разделе.

Ограничения модели многопоточного выполнения

В приложении для Android процессы и задачи всегда конкурируют за память. При наличии только одного основного или UI-потока умный разработчик должен найти способы эффективного манипулирования потоками и работы с ними.

При использовании одного потока асинхронные задачи, отправленные в этот поток, выполняются последовательно — одна за другой. Если выполнение одной из задач занимает вечность, то остальную работу нельзя будет выполнить, пока эта задача не завершится (рис. 6.3).

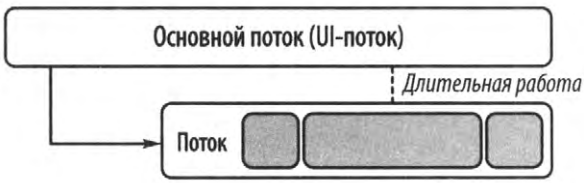


Рис. 6.3. Задачи выполняются последовательно внутри потока

В ситуациях, когда выполнение фоновой задачи может занять много времени, нужно несколько фоновых потоков. Примитивный тип `ThreadPoolExecutor` позволяет запустить несколько потоков и передать им блоки работы для выполнения (рис. 6.4).

Однако наличие бóльшего количества потоков не всегда хорошо. Обратите внимание:

- процессоры могут выполнять только определенное количество потоков параллельно;
- сами потоки затратны с точки зрения памяти — каждый из них обходится по меньшей мере в 64 Кбайт оперативной памяти;

- когда ядро ЦП переключает выполнение с одного потока на другой, происходит *переключение контекста потоков*[4]. Такие переключения не бесплатны. Хотя это не проблема, когда у вас несколько потоков, влияние переключения контекста потоков может быть заметным, если вы будете и дальше добавлять больше потоков. Вы можете достичь точки, когда ваш код будет работать медленнее, чем при использовании меньшего числа потоков.

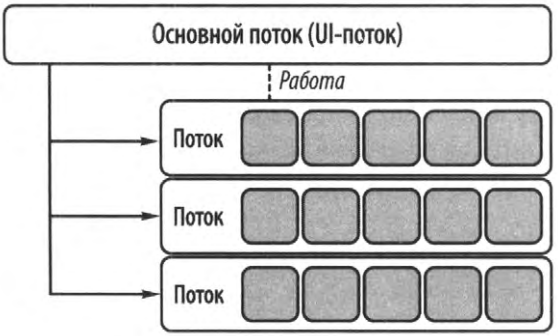


Рис. 6.4. `ThreadPoolExecutor` выполняет всю тяжелую работу по запуску потоков, балансировке нагрузки в этих потоках и даже их завершению

Резюме

- Асинхронную логику можно реализовать с помощью функций обратного вызова. Вы также можете опробовать другие API-интерфейсы, такие как `Handler` и `HandlerThread`. Использование функций обратного вызова может привести к сложным вызовам вложенных функций или к ситуациям, когда поток логики будет разделен на несколько классов, и его трудно будет отслеживать. Если это становится проблемой, то одно из решений — использовать `CompletableFuture` или аналогичный API; такой функциональностью обладает сторонний фреймворк RxJava, но он требует изучения еще одного набора API-интерфейсов, которые могут быстро интегрироваться в бизнес-логику и изменить способ написания кода приложения.
- Чаще всего асинхронная логика связана с получением и обработкой данных, которые затем отображаются в виде экземпляров представления на экране. Для этой цели `ViewModel` из Android Jetpack предлагает компоненты с учетом жизненного цикла, которые помогают создавать более организованный и удобный для сопровождения код.
- Когда жизненный цикл компонента подходит к концу, есть вероятность, что некоторые связанные с ним фоновые задачи теперь следует отменить; в противном

[4] Переключение потоков включает в себя сохранение и загрузку регистров ЦП и карт памяти.

случае они просто будут потреблять память, повышая риск утечки памяти или даже сбоя приложения. Структурированный параллелизм — идеальное решение этой проблемы, которое мы рассмотрим в следующей главе.

- Использование потоков в качестве примитивных типов имеет свои ограничения. Вы должны убедиться, что не создаете слишком много потоков из-за затрат на память, ведь из-за слишком большого количества переключений контекста потока может пострадать производительность.

Сопрограммы предназначены для устранения ограничений *модели многопоточного выполнения*. Следующие четыре главы, в которых основное внимание уделяется сопрограммам, структурированному параллелизму, каналам и потокам, представляют собой "венец" книги и показывают, как Kotlin дает разработчикам приложений для Android реальное преимущество в получении контроля над асинхронными вычислениями.

ГЛАВА 7
Сопрограммы

В предыдущей главе вы познакомились с ловушками модели многопоточного выполнения. В качестве альтернативы этой модели в языке Kotlin есть библиотека *kotlinx.coroutines*, цель которой — устранение упомянутых ранее ограничений. Примитивные типы с поддержкой сопрограмм позволяют разработчикам писать последовательный, асинхронный код при низких затратах. Архитектура сопрограмм включает *функции, поддерживающие возможность приостановки, структурированный параллелизм*, а также *контекст сопрограмм* и *область видимости сопрограмм*. Эти темы тесно связаны друг с другом, и мы рассмотрим каждую из них поэтапно и наглядно.

Что такое сопрограмма?

В официальной документации Kotlin сопрограммы квалифицируются как "легковесные потоки" в попытке использовать существующую и хорошо известную парадигму. Можно рассматривать сопрограммы как *блоки кода, которые можно отправлять неблокирующим потокам.*

Сопрограммы действительно легковесны, но важно отметить, что сами по себе они не являются потоками. На самом деле многие сопрограммы могут работать в одном потоке, хотя у каждой из них свой жизненный цикл. В этом разделе вы увидите, что в действительности это скорее конечные автоматы, где каждое состояние соответствует блоку кода, который в конечном счете будет выполнять какой-то поток.

Возможно, вы удивитесь, обнаружив, что концепция сопрограмм восходит к началу 1960-х годов, когда был создан компилятор Cobol, в котором использовалась идея функций, поддерживающих возможность приостановки и запуска, на языке ассемблера. Сопрограммы также встречаются в таких языках, как Go, Perl и Python.

Библиотека сопрограмм предлагает средства для управления этими потоками "из коробки". Однако при необходимости можно настроить функцию запуска сопрограмм для самостоятельного управления потоками.

Наша первая сопрограмма

В этом разделе мы познакомим вас с большим количеством новой лексики и понятий из пакета *kotlinx.coroutines*. Чтобы обучение прошло гладко, мы решили начать с простого варианта использования сопрограммы и объяснить, как она работает.

В следующем примере, как и в других примерах из этой главы, используется семантика, объявленная в пакете *kotlinx.coroutines*:

```
fun main() = runBlocking{
    val job: Job = launch {
        var i = 0
        while (true) {
            println("$i I'm working")
            i++
            delay(10)
        }
    }

    delay(30)
    job.cancel()
}
```

Метод `runBlocking` запускает новую сопрограмму и блокирует текущий поток до завершения ее работы. Обычно этот метод используется в основных функциях и тестировании, поскольку служит мостом к обычному блокирующему коду.

В блоке кода мы создаем сопрограмму, используя функцию `launch`. Поскольку она создает сопрограмму, это *функция создания* или *запуска сопрограмм*. Позже вы увидите, что существуют и другие подобные функции. Метод `launch` возвращает ссылку на `Job`, представляющий собой жизненный цикл запущенной сопрограммы.

В сопрограмме есть цикл `while`, выполняющийся бесконечно. Далее внизу можно заметить, что позже `job` отменяется. Чтобы продемонстрировать, что это значит, можно запустить программу, и результат будет следующим:

```
0 I'm working
1 I'm working
2 I'm working
```

Похоже, сопрограмма сработала как часы. В тандеме код продолжает выполняться в основном потоке, что дает в общей сложности три строки в течение 30-миллисекундного окна, предоставленного нам вызовом функции `delay()` (рис. 7.1).

С точки зрения использования функция `delay()` подозрительно похожа на функцию `Thread.sleep`. Основное отличие заключается в том, что `Thread.sleep(...)` — это блокирующая функция, а `delay()` — нет. Чтобы продемонстрировать, что имеется в виду, снова обратимся к нашему коду, заменив вызов `delay` в сопрограмме на `Thread.sleep`:

```
fun main() = runBlocking{
    val job: Job = launch {
```

```
        while (true) {
            println("I'm working")
            Thread.sleep(10L)
        }
    }

    delay(30)
    job.cancel()
}
```

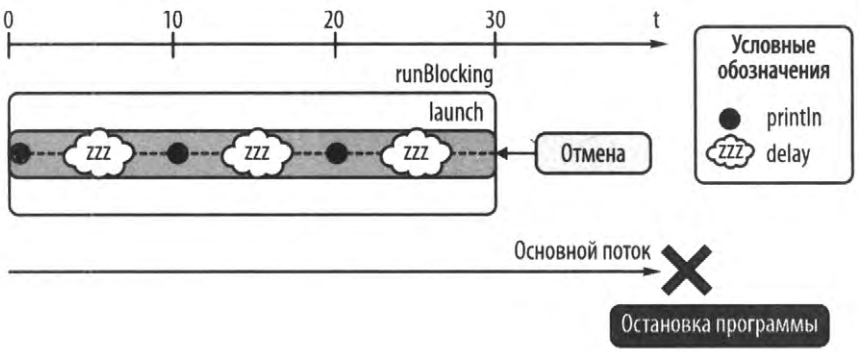


Рис. 7.1. Первая сопрограмма

Посмотрите, что произойдет после повторного выполнения кода. Мы получаем следующий вывод:

```
I'm working
I'm working
I'm working
I'm working
I'm working
I'm working
I'm working
I'm working
I'm working
I'm working
I'm working
.....
```

Теперь кажется, что код работает бесконечно. При выполнении сопрограммы вызов `Thread.sleep(10L)` блокирует основной поток до тех пор, пока сопрограмма, запущенная методом `launch`, не завершится. Поскольку из-за сопрограммы, запущенной

этим методом, основной поток либо "спит", либо выводит результат, сопрограмма не завершается, поэтому выполнение так и будет продолжаться[1] (рис. 7.2).

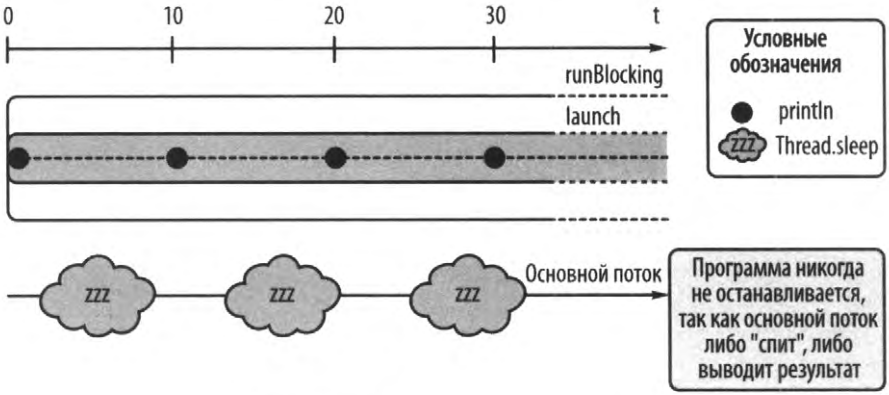


Рис. 7.2. Бесконечная программа

Важно помнить следующее:

- функция `launch` работает по принципу "выстрелил и забыл" — другими словами, она не возвращает результат;
- после вызова сразу же возвращается экземпляр `Job` и запускается новая сопрограмма. `Job` представляет саму сопрограмму, как дескриптор ее жизненного цикла. Сопрограмму можно отменить, вызвав метод `cancel` для экземпляра `Job`;
- сопрограмма, запущенная методом `launch`, возвращает не результат, а ссылку на фоновое задание.

Если, с другой стороны, вам нужно получить результат асинхронного вычисления, следует использовать функцию `async`.

Функция *async*

Функцию `async` можно сравнить с моделью `Future/Promise` в Java для поддержки асинхронного программирования:

```
class WorkingClass() {
    public CompletableFuture<SomeOtherResult> doBothAsync() {
        somethingAsync().thenAcceptBoth(somethingElseAsync()) {
            one, two ->
            // Объединяем здесь результаты обоих вызовов
        };
    }
}
```

[1] В этом сценарии `job.cancel()` не влияет на сопрограмму, запускаемую функцией `launch`. Мы затронем эту тему в следующей главе (сопрограмма должна действовать согласованно, чтобы ее можно было отменить).

Вместо выполнения блокирующего вызова для получения данных функция async тотчас же возвращает обертку вокруг результата. В зависимости от используемой библиотеки эта обертка может называться Future, CompletableFuture, Promise и т. д. Она похожа на дескриптор, по которому можно проверить, доступен результат или нет. При желании поток можно заблокировать с помощью метода Future.get(), пока результат не будет доступен.

Так же, как и Future, функция async *возвращает обертку вокруг результата*; ее тип — Deferred<T> (обобщенный тип — это тип результата), как показано в следующем коде:

```
fun main() = runBlocking{
    val slow: Deferred<Int> = async {
        var result = 0
        delay(1000) // имитируем медленную фоновую работу
        for (i in 1..10) {
            result += i
        }
        println("Call complete for slow: $result")
        result
    }

    val quick: Deferred<Int> = async {
        delay(100) // имитируем быструю фоновую работу
        println("Call complete for quick: 5")
        5
    }

    val result: Int = quick.await() + slow.await()
    println(result)
}
```

Типы данных quick и slow — это будущий результат в качестве реализации Deferred<Int>, также известный как Job с результатом. Вызывая метод await() для каждого экземпляра Deferred<Int>, программа ожидает результат каждой сопрограммы.

На этот раз мы запустили две сопрограммы с помощью функции async. Сам код может дать неплохое представление о том, что может произойти, но все равно выполним его, чтобы увидеть следующий вывод:

```
Call complete for quick: 5
Call complete for slow: 55
60
```

В предыдущей программе мы видим 1000-миллисекундную задержку (delay(1000)) и 100-миллисекундную задержку (delay(100)) — result ожидает завершения обоих заданий, прежде чем вывести результат.

Важно помнить следующее:

- функция async предназначена для *параллельной декомпозиции работы*, т. е. вы *явно* указываете, что некоторые задачи будут выполняться одновременно;
- после вызова этой функции сразу же возвращается экземпляр Deferred — это специализированное задание Job с дополнительными методами, такими как await(). Это экземпляр Job с возвращаемым значением;
- для получения возвращаемого значения вызывается метод await() для экземпляра Deferred, что очень похоже на Future и Promise[2].

Вы, возможно, уже заметили, что примеры с функциями launch и async обернуты вызовом функции runBlocking. Ранее мы упоминали, что она запускает новую сопрограмму и блокирует текущий поток до тех пор, пока работа сопрограммы не завершится. Чтобы лучше понять роль этой функции, для начала нужно предоставить предварительный обзор структурированного параллелизма — концепции, которая будет подробно рассмотрена в следующей главе.

Краткий обзор структурированного параллелизма

Сопрограммы — это не просто еще один причудливый способ запуска фоновых задач. Библиотека сопрограмм построена на парадигме структурированного параллелизма. Прежде чем углубиться в изучение сопрограмм, нужно понимать, что это такое и какие проблемы призвана решать данная библиотека.

Упрощение разработки — достойная цель. В случае структурированного параллелизма это почти счастливый побочный эффект ответа на более общую проблему. Рассмотрим простейшую конструкцию, с которой знаком каждый разработчик: функцию.

Рис. 7.3. Два потока

Функции предсказуемы в том смысле, что они выполняются сверху вниз. Если исключить возможность того, что исключения могут быть выброшены из функции[3], мы знаем, что до функции, возвращающей значение, порядок выполнения является последовательным: сначала выполняется один оператор, потом следующий. А если

[2] Так мы приостанавливаем вызывающую сопрограмму до тех пор, пока не будет получено значение или выброшено исключение, если сопрограмма, запущенная с помощью функции async, будет отменена или завершится ошибкой с исключением. Подробнее об этом *см. далее в этой главе*.

[3] Мы предполагаем, что исключения обрабатываются и не мешают потоку выполнения.

внутри функции программа создает и запускает другой поток? Это абсолютно допустимо, но теперь у нас два потока выполнения (рис. 7.3).

Вызов этой функции дает не только один результат; у него есть побочный эффект — он создает параллельный поток выполнения, что может быть проблематично по следующим причинам.

Исключения не передаются.

Если внутри потока выбрасывается исключение, которое не обрабатывается, то JVM вызывает простой интерфейс `UncaughtExceptionHandler`:

```
interface UncaughtExceptionHandler {
    fun uncaughtException(t: Thread, e: Throwable)
}
```

Можно предоставить обработчик с помощью метода `Thread.setUncaughtExceptionHandler` в экземпляре потока. По умолчанию при создании потока у него нет определенного обработчика `UncaughtExceptionHandler`. Если исключение не перехвачено и конкретный обработчик не задан, вызывается обработчик по умолчанию.

Важно отметить, что во фреймворке Android `UncaughtExceptionHandler` по умолчанию приведет к сбою, завершив системный процесс приложения. Разработчики приложений для Android сделали этот выбор, потому что, как правило, для приложения лучше немедленно прекратить работу и сообщить об ошибке, поскольку система не должна принимать решения от имени разработчика, когда речь идет о необработанных исключениях. В этом случае трассировка стека имеет отношение к реальной проблеме, в то время как восстановление после сбоя может привести к непоследовательному поведению и проблемам, которые не так очевидны, потому что основная причина может крыться гораздо выше в стеке вызовов.

В нашем примере нет ничего, что могло бы сообщить функции, если в фоновом потоке произойдет что-то плохое. Иногда это нормально, потому что ошибки можно обрабатывать непосредственно из фонового потока, но у вас может быть более сложная логика, требующая, чтобы вызывающий код отслеживал проблемы, дабы реагировать по-другому и конкретным образом.

Перед вызовом обработчика по умолчанию задействован механизм. Каждый поток может принадлежать экземпляру `ThreadGroup`, который может обрабатывать исключения. У каждой группы потоков также может быть родительская группа потоков. В рамках фреймворка Android статически создаются две группы: "системная" группа и потомок системной группы, известный как "основная" группа. "Основная" группа всегда делегирует обработку исключений "системной" группе, которая затем делегирует ее `Thread.getDefaultUncaughtExceptionHandler()` в случае отсутствия значения `null`. В противном случае "системная" группа выводит имя исключения и трассировку стека в `System.err`.

Поток выполнения трудно контролировать.

Поскольку поток можно создавать и запускать откуда угодно, представьте, что фоновый поток создает и запускает три новых потока, чтобы делегировать часть работы, или выполняет задачи в ответ на вычисления, выполняемые в контексте родительского потока (рис. 7.4).

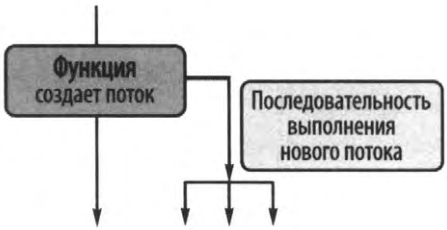


Рис. 7.4. Несколько потоков

Как убедиться, что функция возвращает управление, только когда вся фоновая обработка завершена? Здесь могут возникнуть ошибки: нужно убедиться, что вы ждете, пока все дочерние потоки завершат работу[4]. При использовании реализации на базе `Future` (например, `CompletableFuture`), даже если опустить вызов `Future.get`, это может привести к преждевременному завершению потока выполнения.

Позже, когда фоновый поток и все его дочерние потоки все еще работают, всю эту работу, возможно, придется отменить (пользователь вышел из пользовательского интерфейса, возникла ошибка и т. д.). В этом случае автоматического механизма для отмены всей иерархии задач нет.

При работе с потоками очень легко забыть о фоновой задаче. *Структурированный параллелизм — это не что иное как концепция, предназначенная для решения этой проблемы.*

В следующем разделе мы подробно рассмотрим ее и объясним, как она связана с сопрограммами.

Связь "родитель — потомок" в структурированном параллелизме

До сих пор мы говорили о потоках, которые на предыдущих рисунках обозначались стрелками. Представим себе более высокий уровень абстракции, где родитель может создать несколько потомков (рис. 7.5).

[4] Метод `join()` переводит вызывающий поток в состояние ожидания. Он остается в этом состоянии, пока исходный поток не завершит работу.

Рис. 7.5. Родитель — потомок

Потомки могут выполняться одновременно друг с другом, а также с родителем. Если родитель вызывает ошибку или отменяется, то все его потомки также отменяются[5]. Вот *первое правило структурированного параллелизма*:

"Отмена всегда передается потомкам".

То, как сбой одного потомка влияет на других потомков того же уровня, является параметризацией родителя.

Подобно тому, как родитель может вызвать ошибку или быть отменен, это может случиться с любым из потомков. В случае отмены одного из потомков, согласно первому правилу, мы знаем, что родитель не будет отменен (отмена передается потомкам, а не родителю). Дальнейшие действия в случае неудачи зависят от проблемы, которую вы пытаетесь решить. Сбой одного потомка должен или не должен приводить к отмене других потомков (рис. 7.6). Эти две возможности характеризуют связь "родитель — потомок" и являются параметризацией родителя.

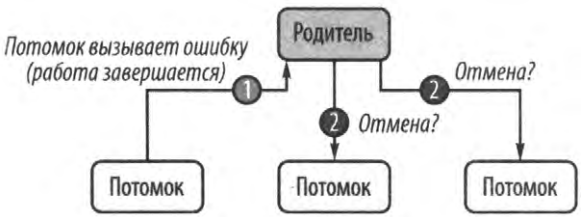


Рис. 7.6. Политика отмены

Родитель всегда ждет завершения всех своих потомков.

[5] Возникновение ошибки соответствует любому аномальному событию, после которого восстановление уже невозможно. Обычно это реализуется с использованием необработанных или сгенерированных исключений.

Можно было бы добавить и другие правила, связанные с передачей исключений, но они будут зависеть от реализации. Пришло время представить несколько конкретных примеров.

Структурированный параллелизм доступен в сопрограммах Kotlin с помощью интерфейсов `CoroutineScope` и `CoroutineContext`. И `CoroutineScope`, и `CoroutineContext` играют роль родителя, изображенного на предыдущих иллюстрациях, в то время как сопрограммы выступают в роли потомков.

В следующем разделе мы подробнее рассмотрим `CoroutineScope` и `CoroutineContext`.

CoroutineScope и CoroutineContext

Мы подробно познакомимся с библиотекой *kotlinx.coroutines*. В следующем разделе будет *много* новых концепций. Хотя они и важны, если вы хотите освоить сопрограммы, не нужно учить все прямо сейчас, чтобы приступить к продуктивной работе с сопрограммами. В этом разделе и в следующей главе будет много примеров, которые дадут вам хорошее представление о том, как работают сопрограммы. Поэтому, возможно, вам будет проще вернуться к этому разделу после того, как вы немного попрактикуетесь.

Теперь, когда вы знаете, что такое структурированный параллелизм, снова вернемся ко всей этой истории с `runBlocking`. Почему бы просто не вызвать функции `launch` или `async` за пределами вызова `runBlocking`?

Следующий код не будет компилироваться:

```
fun main() {
    launch {
        println("I'm working") // не будет компилироваться
    }
}
```

Компилятор сообщает: "Unresolved reference: launch". Это связано с тем, что функции запуска сопрограмм — это функции-расширения `CoroutineScope`.

`CoroutineScope` управляет жизненным циклом сопрограммы в четко определенной области видимости или жизненном цикле. Это объект, который играет роль родителя в структурированном параллелизме, его цель — управлять сопрограммами, которые вы создаете в нем, и отслеживать их. Возможно, вы удивитесь, обнаружив, что в предыдущем примере с функцией `async` блок `runBlocking` уже предоставил `CoroutineScope` для запуска новой сопрограммы. Как? Вот упрощенная сигнатура функции `runBlocking`:

```
fun <T> runBlocking (
    // аргументы функции удалены для краткости
    block: suspend CoroutineScope.() -> T): T { // impl
}
```

Последний аргумент — это функция с приемником типа CoroutineScope. Следовательно, когда вы предоставляете функцию для аргумента блока, в вашем распоряжении есть область видимости сопрограмм, которая может вызывать функции-расширения CoroutineScope. Как показано на рис. 7.7, если включить в Android Studio "подсказки при вводе кода", то можно увидеть параметр типа.

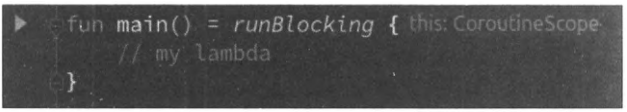


Рис. 7.7. Подсказка при вводе кода в Android Studio

Какова цель runBlocking помимо предоставления объекта CoroutineScope? Этот метод блокирует текущий поток до его завершения. Его можно вызвать из обычного блокирующего кода в качестве моста к коду, содержащему функции, поддерживающие возможность приостановки (мы рассмотрим эти функции далее в этой главе).

Чтобы иметь возможность создавать сопрограммы, нужно соединить код с "обычной" функцией main. Однако следующий пример не будет компилироваться, т. к. мы пытаемся запустить сопрограмму из обычного кода:

```
fun main() = launch {
    println("I'm a coroutine")
}
```

Это связано с тем, что на самом деле launch — это *функция-расширение* CoroutineScope:

```
fun CoroutineScope.launch(
    context: CoroutineContext = EmptyCoroutineContext,
    // другие параметры удалены для краткости
    block: suspend CoroutineScope.() -> Unit
): Job { /* реализация */ }
```

Поскольку обычный код не предоставляет экземпляр CoroutineScope, вызывать из него функции запуска сопрограмм напрямую нельзя.

Так кто же этот CoroutineContext? Чтобы ответить на этот вопрос, нужно разобраться в деталях CoroutineScope.

Если посмотреть на исходный код, видно, что это интерфейс:

```
interface CoroutineScope {
    val coroutineContext: CoroutineContext
}
```

Другими словами, CoroutineScope — это контейнер для CoroutineContext.

Цель CoroutineScope — инкапсуляция параллельных задач (сопрограмм и других областей видимости) путем применения структурированного параллелизма. Области видимости и сопрограммы образуют древовидную архитектуру с областью видимости в корне (рис. 7.8).

Рис. 7.8. Древовидная связь
(сопрограммы представлены в виде прямоугольников)

`CoroutineContext`, который в дальнейшем мы будем называть просто *контекстом*, — более широкая концепция. Это неизменяемое объединение множества контекстных элементов. Далее для обозначения *элемента контекста* мы будем использовать термин "элемент".

Это теория. На практике вы чаще всего будете использовать специальный элемент контекста для управления тем, какой поток или пул потоков будет выполнять сопрограммы. Например, представьте, что вам нужно выполнить вычисления с высокой нагрузкой на ЦП в функции `launch`, не блокируя при этом основной поток. Вот где библиотека сопрограмм действительно удобна, потому что пулы потоков для наиболее распространенных вариантов применений доступны "из коробки". В случае со счетными задачами не нужно определять собственный пул потоков. Все, что нужно сделать, — это использовать специальный элемент контекста `Dispatchers.Default`:

```
fun main() = runBlocking<Unit> {
    launch(Dispatchers.Default) {
        println("I'm executing in ${Thread.currentThread().name}")
    }
}
```

Вывод теперь выглядит так:

```
I'm executing in DefaultDispatcher-worker-2 @coroutine#2
```

`Dispatchers.Main` — это элемент контекста. Как будет показано позже, различные элементы контекста можно комбинировать с помощью операторов для дополнительной настройки поведения сопрограмм.

Как следует из названия, цель `Dispatcher` — отправка сопрограмм в конкретный поток или пул потоков. По умолчанию "из коробки" доступны четыре диспетчера — `Main`, `Default`, `IO` и `Unconfined`.

`Dispatchers.Main`

Использует основной или UI-поток используемой вами платформы.

`Dispatchers.Default`

Предназначен для счетных задач и поддерживается пулом из четырех потоков по умолчанию.

`Dispatchers.IO`

Предназначен для задач, ограниченных возможностями ввода-вывода, и поддерживается пулом из 64 потоков по умолчанию

`Dispatchers.Unconfined`

Его не следует применять при изучении сопрограмм, и он вряд ли вам понадобится. В основном он используется в библиотеке сопрограмм.

Просто изменив диспетчер, можно контролировать, в каком потоке или пуле потоков будет выполняться сопрограмма. Элемент контекста `Dispatcher.Default` представляет собой подкласс `CoroutineDispatcher`, но существуют и другие элементы.

Предоставляя контекст диспетчера, можно легко указать, где выполняется логический поток. Таким образом, разработчик несет ответственность за предоставление контекста функции запуска сопрограмм.

Говоря языком сопрограмм, сопрограмма всегда выполняется в контексте. *Этот* контекст предоставляется областью видимости сопрограммы и отличается от контекста, который предоставляем мы. Во избежание путаницы будем называть контекст, который мы предоставляем функции запуска сопрограмм, *предоставляемым контекстом*.

Разница едва заметна. Помните объект `Job`? Экземпляр `Job` — это дескриптор жизненного цикла сопрограммы, а также часть контекста сопрограмм. У каждой сопрограммы есть экземпляр `Job`, который ее представляет и является частью контекста сопрограмм.

Пришло время раскрыть секрет, как создаются эти контексты. Посмотрите на пример 7.1, который немного отличается от предыдущего.

Пример 7.1. Пример с диспетчерами

```
fun main() = runBlocking<Unit>(Dispatchers.Main) {
    launch(Dispatchers.Default) {
        val threadName = Thread.currentThread().name
        println("I'm executing in $threadName")
    }
}
```

Этот блок кода создает две сопрограммы с соответствующим экземпляром `Job`: функция `runBlocking` запускает первую сопрограмму, а вторую запускает функция `launch`.

У сопрограммы, созданной функцией `runBlocking`, есть собственный контекст. Поскольку это корневая сопрограмма, запущенная в области видимости, мы называем

этот контекст *контекстом области видимости*. Контекст области видимости включает в себя контекст сопрограммы (рис. 7.9).

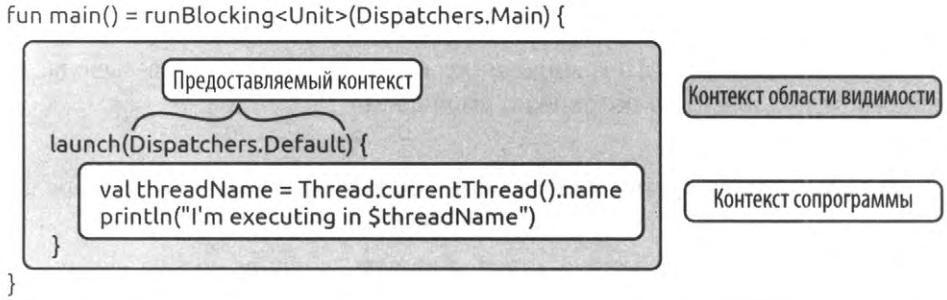


Рис. 7.9. Контексты

Вы уже видели, что `launch` — это функция-расширение `CoroutineScope` (которая содержит контекст), и она может получать контекст в качестве первого параметра. Итак, в нашем распоряжении есть два контекста в этой функции, как показано в примере 7.1: один из типа получателя (контекст области видимости), а второй — из параметра контекста (предоставляемый контекст).

Что делает функция `launch` в своей реализации перед вызовом предоставленной функции? Она объединяет два контекста, чтобы элементы из параметра контекста имели приоритет над другими элементами из области видимости. В результате этой операции слияния мы получаем родительский контекст. На данный момент `Job` сопрограммы еще не создан.

Наконец, мы создаем новый экземпляр задания `Job` в качестве потомка `Job` из родительского контекста. После этого он добавляется в родительский контекст, заменяя экземпляр `Job` из родительского контекста для получения контекста сопрограмм.

Эти связи и взаимодействия показаны на рис. 7.10, где контекст обозначен прямоугольником, содержащим другие элементы контекста.

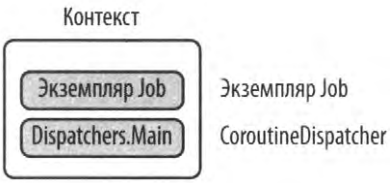


Рис. 7.10. Представление контекста

На рис. 7.10 показан контекст, содержащий экземпляр `Job` и диспетчер `Dispatchers.Main`. Учитывая это представление, на рис. 7.11 показан способ представления контекста из примера 7.1.

Все, что вы указываете в предоставляемом контексте методу `launch`, имеет приоритет над контекстом области видимости. Это приводит к *родительскому контексту*, наследующему элементы из контекста области видимости, которых не было в предоставляемом контексте (в данном случае `Job`). Затем создается новый экземпляр

Job (с точкой в правом верхнем углу) в качестве потомка Job родителя, который в данном случае также является Job контекста области видимости. Результирующий контекст сопрограммы состоит из элементов родительского контекста, за исключением Job (который является потомком Job в родительском контексте).

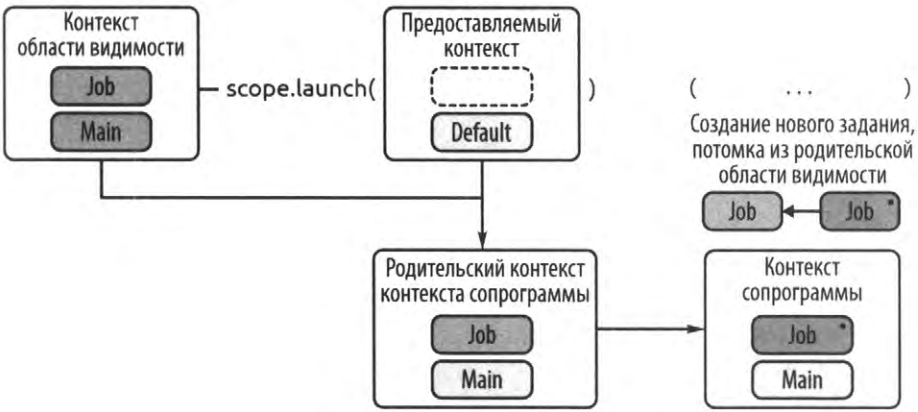


Рис. 7.11. Детали контекста

Данный *контекст сопрограмм* — это контекст, в котором будет выполняться лямбда-выражение, которую мы предоставляем для запуска.

Структурированный параллелизм возможен, потому что Job в контексте сопрограммы — это потомок Job из родительского контекста. Если область видимости по какой-либо причине отменяется, любая работающая дочерняя сопрограмма автоматически отменяется[6].

Что еще более важно, контекст сопрограмм наследует контекстные элементы из контекста области видимости, которые не переопределяются контекстом, предоставленным в качестве параметра для запуска; в этом отношении метод async ведет себя точно так же.

Функции, поддерживающие возможность приостановки

Мы рассмотрели, как запустить сопрограмму, используя функции launch и async, и коснулись вопросов блокировки и ее отсутствия. Сопрограммы Kotlin предлагают еще кое-что, чтобы продемонстрировать, насколько мощными они могут быть: *функции, поддерживающие возможность приостановки.*

---

[6] Возможно, вы заметили, что ничто не мешает вам передать экземпляр Job в "предоставляемом контексте". Что тогда произойдет? Следуя объясненной логике, этот экземпляр становится родителем Job контекста сопрограмм (например, вновь созданной сопрограммы). Таким образом, область видимости больше не является родителем сопрограммы; связь "родитель — потомок" разорвана. По данной причине делать это настоятельно не рекомендуется, за исключением особых случаев, которые будут объяснены в следующей главе.

Представим, что мы последовательно вызываем две задачи. Первая задача завершается до того, как вторая сможет приступить к выполнению (рис. 7.12).

Рис. 7.12. Последовательный вызов двух задач

При выполнении задачи *A* базовый поток не может продолжить выполнение других задач; тогда говорят, что задача *A* — это *блокирующий вызов*.

Однако задача *A*, тратящая разумное количество времени на ожидание более продолжительного задания (например, запроса по протоколу HTTP), в конечном счете блокирует базовый поток, делая ожидающую задачу *B* бесполезной.

Таким образом, задача *B* ожидает завершения задачи *A*. Экономный разработчик может рассматривать этот сценарий как пустую трату ресурсов потока, поскольку поток может (и должен) приступить к выполнению другой задачи, пока задача *A* ожидает результата сетевого вызова.

Используя функции, поддерживающие возможность приостановки, можно разделить задачи на фрагменты, которые можно *приостановить*. В нашем примере задачу *A* можно приостановить, когда она выполняет удаленный вызов, оставляя основной поток свободным для выполнения другой задачи (или только ее части). Когда задача *A* получает результат своего удаленного вызова, ее можно возобновить в более поздний момент времени (рис. 7.13).

Рис. 7.13. Сэкономленное время отображается в конце

Как видите, две задачи выполняются быстрее, чем в предыдущем случае. В результате такого чередования фрагментов задач базовый поток всегда занят выполнением задачи. Таким образом, механизм приостановки требует меньшего количества потоков для получения той же общей пропускной способности, а это весьма важно, когда у каждого потока есть собственный стек, занимающий минимум 64 Кбайт памяти. Обычно поток занимает 1 Мбайт ОЗУ.

С помощью механизма приостановки можно экономить, используя большее количество тех же ресурсов.

Функции, поддерживающие возможность приостановки, "под капотом"

На данный момент мы ввели новую концепцию: факт, согласно которому задачу можно *приостановить*. Задача может "приостановить" выполнение, не блокируя

основной поток. Хотя это может показаться волшебством, важно понимать, что все сводится к низкоуровневым конструкциям, которые мы объясним в этом разделе.

Задачу, или, точнее, сопрограмму, можно приостановить, если она использует хотя бы одну *функцию, поддерживающую возможность приостановки*. Такую функцию легко узнать, т. к. она объявляется с модификатором suspend.

Когда компилятор Kotlin встречает эту функцию, она компилируется в обычную функцию с дополнительным параметром типа Continuation<T>, который представляет собой просто интерфейс, как показано в примере 7.2.

Пример 7.2. Интерфейс Continuation<T>

```
public interface Continuation<in T> {
    /**
     * Контекст сопрограммы, соответствующей этому интерфейсу.
     */
    public val context: CoroutineContext

    /**
     * Возобновляет выполнение соответствующей сопрограммы, передавая успешный
       или неудачный [результат] в качестве возвращаемого значения
       последней точки приостановки.
     */
    public fun resumeWith(result: Result<T>)
}
```

Предположим, что мы определяем эту функцию следующим образом:

```
suspend fun backgroundWork(): Int {
    // возвращает значение типа Int
}
```

Во время компиляции она преобразуется в обычную функцию (без модификатора suspend) с дополнительным аргументом Continuation:

```
fun backgroundWork(callback: Continuation<Int>): Int {
    // возвращает значение типа Int
}
```

Функции, поддерживающие возможность приостановки, компилируются в обычные функции, принимающие дополнительный аргумент Continuation. Это реализация Continuation Passing Style (CPS) — стиля программирования, в котором поток управления передается в форме объекта Continuation.

Объект Continuation содержит весь код, который должен быть выполнен в теле функции backgroundWork.

Что на самом деле генерирует компилятор Kotlin для этого объекта?

Из соображений эффективности он генерирует конечный автомат[7]. Реализация конечного автомата заключается в выделении памяти для как можно меньшего количества объектов, поскольку сопрограммы легковесны и их количество в режиме выполнения может исчисляться тысячами.

В этом автомате каждое состояние соответствует *точке приостановки* в теле функции, поддерживающей возможность приостановки. Рассмотрим пример. Представьте, что в проекте Android мы используем уровень презентатора для выполнения некоторых длительных процессов, окружающих ввод-вывод и обработку графики, где в следующем блоке кода имеется две точки приостановки с самоуправляемой сопрограммой, запущенной из `viewModelScope`[8]:

```
suspend fun renderImage() {
    val path: String = getPath()
    val image = fetchImage(path)   // первая точка приостановки (fetchImage
                // - это функция, поддерживающая возможность приостановки)
    val clipped = clipImage(image) // вторая точка приостановки (clipImage
                // - это функция, поддерживающая возможность приостановки
    postProcess(clipped)
}
/** Вот пример использования функции, поддерживающей возможность
    приостановки [renderImage] */
fun onStart() {
    viewModelScope.launch(Dispatchers.IO) {
        renderImage()
    }
}
```

Компилятор генерирует анонимный класс, реализующий интерфейс `Continuation`. Чтобы вы представляли, что генерируется на самом деле, мы предоставим псевдокод того, что генерируется для функции `renderImage`. У класса есть поле `state`, содержащее текущее состояние конечного автомата, а также поля для каждой переменной, которые совместно используются состояниями:

```
object : Continuation<Unit> {
    // состояние
    private var state = 0
```

---

[7] На самом деле, когда функция, поддерживающая возможность приостановки, вызывает только одну функцию, допускающую приостановку, в качестве хвостового рекурсивного вызова, конечный автомат не требуется.

[8] `viewModelScope` проистекает из реализации библиотеки AndroidX, `ViewModel`. `ViewModelScope` ограничена жизненным циклом `ViewModel`. Подробнее об этом *см. в следующей главе*.

```
    // поля
    private var path: String? = null
    private var image: Image? = null

    fun resumeWith(result: Any) {
        when (state) {
            0 -> {
                path = getPath()
                state = 1
                // Передаем этот конечный автомат в качестве объекта Continuation
                val firstResult = fetchImage(path, this)
                if (firstResult == COROUTINE_SUSPENDED) return
                // Если мы не получаем COROUTINE_SUSPENDED, то получаем фактический
                // экземпляр Image, и выполнение должно перейти к следующему состоянию
                resumeWith(firstResult)
            }
            1 -> {
                image = result as Image
                state = 2
                val secondResult = clipImage(image, this)
                if (secondResult == COROUTINE_SUSPENDED) return
                    resumeWith(secondResult)
            }
            2 -> {
                val clipped = result as Image
                postProcess(clipped)
            }
        else -> throw IllegalStateException()
        }
    }
}
```

Этот конечный автомат инициализируется строкой `private var state = 0`. Следовательно, когда сопрограмма, запускаемая методом `launch`, вызывает функцию `renderImage`, выполнение "перескакивает" к первой ветке (0). Мы извлекаем путь, задаем для следующего состояния значение 1, а затем вызываем `fetchImage` — первую функцию, поддерживающую возможность приостановки, в теле `renderImage`.

На данном этапе возможны два сценария:

1. `fetchImage` требуется время для возврата экземпляра `Image`, и она незамедлительно возвращает значение `COROUTINE_SUSPENDED`. Возвращая это конкретное значение, функция, по сути, говорит: "Мне нужно больше времени, чтобы вернуть фактическое значение, поэтому дайте мне объект конечного автомата, и я буду ис-

пользовать его, когда получу результат". Когда у fetchImage, наконец, появляется экземпляр Image, она вызывает stateMachine.resumeWith(image). Поскольку в этот момент состояние равно 1, выполнение "перескакивает" на вторую ветку оператора when.

2. fetchImage незамедлительно возвращает экземпляр Image. В этом случае выполнение переходит к следующему состоянию (через resumeWith(image)).

Остальная часть выполнения следует той же схеме, пока код последнего состояния не вызовет функцию postProcess.

Данное объяснение не является точным состоянием конечного автомата, генерируемого в байт-коде; скорее, это псевдокод его репрезентативной логики для передачи основной идеи. Для повседневного использования менее важно знать детали реализации реального конечного автомата, сгенерированного в байт-коде Kotlin, нежели понимать, что происходит "под капотом".

На концептуальном уровне при вызове функции, поддерживающей возможность приостановки, функция обратного вызова (Continuation) создается вместе с генерируемыми структурами, поэтому остальная часть кода после такой функции будет вызываться, только когда функция, поддерживающая возможность приостановки, возвращает управление. Затрачивая меньше времени на шаблонный код, можно сосредоточиться на бизнес-логике и высокоуровневых концепциях.

До сих пор мы анализировали, как компилятор Kotlin реструктурирует код "под капотом", таким образом, чтобы нам не надо было писать функции обратного вызова самостоятельно. Конечно, не нужно полностью знать генерацию кода конечного автомата, чтобы использовать функции, поддерживающие возможность приостановки. Тем не менее эта концепция важна для понимания! Поэтому нет ничего лучше практики!

Использование сопрограмм и функций, поддерживающих возможность приостановки: практический пример

Представьте, что в приложении для Android вы хотите загрузить профиль пользователя с идентификатором. При переходе к профилю, возможно, имеет смысл получить данные пользователя на основе идентификатора в методе fetchAndLoadProfile.

Для этого можно использовать сопрограммы, применяя знания из предыдущего раздела. Пока предположим, что где-то в приложении (обычно в MVC это контроллер или ViewModel в MVVM) у вас есть CoroutineScope. У CoroutineScope есть диспетчер Dispatchers.Main в CoroutineContext. В таком случае мы говорим, что эта область

видимости отправляет сопрограммы в основной поток, что идентично поведению по умолчанию. В следующих главах мы приведем подробные объяснения и примеры областей видимости сопрограмм, а также покажем, как получить к ним доступ и при необходимости создавать их самостоятельно.

Тот факт, что область видимости по умолчанию — это основной поток, никоим образом не является ограничивающим, поскольку вы можете создавать сопрограммы с любым `CoroutineDispatcher`, который вам нужен в этой области видимости. Это иллюстрирует следующая реализация метода `fetchAndLoadProfile`:

```
fun fetchAndLoadProfile(id: String) {
    scope.launch {                                          ❶
        val profileDeferred = async(Dispatchers.Default) {  ❷
            fetchProfile(id)
        }
        val profile = profileDeferred.await()               ❸
        loadProfile(profile)                                ❹
    }
}
```

Все выполняется в четыре этапа.

❶ Начнем с функции `launch`. Нам нужно, чтобы функция `fetchAndLoadProfile` возвращала управление сразу же, и мы могли последовательно работать в основном потоке. Поскольку областью видимости по умолчанию является основной поток, `launch` без дополнительного контекста наследует контекст области видимости, поэтому выполняется в основном потоке.

❷ Используя функцию `async` и `Dispatchers.Default`, мы вызываем функцию `fetchProfile`, которая является блокирующим вызовом. Напоминаем, что использование `Dispatchers.Default` приводит к выполнению функции `fetchProfile` в пуле потоков. Мы сразу же получаем `Deferred<Profile>`, которому даем имя `profileDeferred`. На данный момент в одном из потоков пула потоков выполняется текущая фоновая работа. Это сигнатура функции `fetchProfile`: `fun fetchProfile(id: String): Profile { // impl }`. Это блокирующий вызов, который может выполнять запрос к базе данных на удаленном сервере.

❸ Нельзя сразу использовать `profileDeferred` для загрузки профиля — нужно дождаться результата фонового запроса. Для этого используется функция `profileDeferred.await()`, которая сгенерирует и вернет экземпляр `Profile`.

❹ Наконец, получив профиль, можно вызвать функцию `loadProfile`. Поскольку внешний запуск наследует свой контекст от родительской области видимости, эта функция вызывается в основном потоке. Мы предполагаем, что это ожидаемо, поскольку большинство операций, связанных с пользовательским интерфейсом, должны выполняться в основном потоке.

Всякий раз при вызове метода `fetchAndLoadProfile` фоновая обработка выполняется за пределами UI-потока, чтобы получить профиль. Как только профиль становится доступным, пользовательский интерфейс обновляется. Можно вызывать метод

fetchAndLoadProfile из любого потока — это не изменит того факта, что функция loadProfile в конечном счете вызывается в UI-потоке.

Неплохо, но можно и лучше.

Обратите внимание, как этот код читается сверху вниз, без косвенности или функций обратного вызова. Вы можете возразить, что имя "profileDeferred" и вызовы await() кажутся неуклюжими, что может стать еще очевиднее, когда вы получаете профиль, ждете его, а затем загружаете. Здесь в игру вступают функции, поддерживающие возможность приостановки.

Эти функции лежат в основе фреймворка сопрограмм.

На концептуальном уровне функция, поддерживающая возможность приостановки, не может вернуть управление сразу же. Если этого не происходит незамедлительно, она приостанавливает сопрограмму, вызвавшую эту функцию, пока выполняется вычисление. Внутреннее вычисление *не должно блокировать* вызывающий поток. Позже, когда вычисление завершается, выполнение сопрограммы возобновляется.

Функцию, поддерживающую возможность приостановки, можно вызвать только из сопрограммы или другой такой же функции.

Говоря "приостановить сопрограмму", мы подразумеваем, что выполнение сопрограммы остановлено. Например:

```
suspend fun backgroundWork(): Int {
    // возвращаем значение типа Int
}
```

Во-первых, функция, поддерживающая возможность приостановки, — это необычная функция; у нее есть собственное ключевое слово suspend. У нее может быть тип возвращаемого значения, но обратите внимание, что в этом случае она не возвращает Deferred<Int> — только значение типа Int.

Во-вторых, ее можно вызвать только из сопрограммы или другой функции, поддерживающей возможность приостановки.

Вернемся к предыдущему примеру: получение и ожидание профиля выполнялись с помощью блока async. На концептуальном уровне это и есть цель функции, поддерживающей возможность приостановки. Мы позаимствуем то же имя, что и у блокирующей функции fetchProfile, и перепишем код:

```
suspend fun fetchProfile(id: String): Profile {
    // пока мы не показываем реализацию
}
```

Два основных отличия от исходного блока async — это модификатор suspend и тип возвращаемого значения.

Это позволяет упростить функцию fetchAndLoadProfile:

```
fun fetchAndLoadProfile(id: String) {
    scope.launch {
```

```
        val profile = fetchProfile(id) // Приостановка
        loadProfile(profile)
    }
}
```

Теперь, когда `fetchProfile` является функцией, поддерживающей возможность приостановки, сопрограмма, запускаемая методом `launch`, приостанавливается при вызове этой функции. *"Приостанавливается"* означает, что выполнение сопрограммы остановлено и следующая строка не выполняется. Она будет оставаться в таком состоянии до тех пор, пока не будет получен профиль, после чего работа сопрограммы возобновится. Потом выполняется следующая строка (с `loadProfile`).

Обратите внимание, что это похоже на процедурный код. Представьте, как бы вы реализовали сложную асинхронную логику, где каждый шаг требует результата предыдущего шага. Вы бы вызывали функции, поддерживающие возможность приостановки, одну за другой в классическом процедурном стиле. Код, который легко понять, легче в сопровождении. Это один из наиболее полезных аспектов таких функций.

В качестве бонуса IntelliJ IDEA и Android Studio помогут вам мгновенно обнаружить вызовы, допускающие приостановку. На рис. 7.14 виден символ на полях, указывающий на такой вызов.

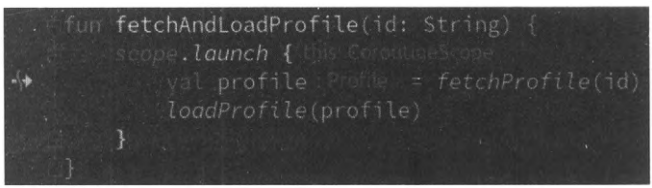


Рис. 7.14. Вызов, допускающий приостановку

Если вы видите на полях этот символ, вы знаете, что сопрограмма может временно приостановиться на этой строке.

Не ошибитесь с модификатором *suspend*

Как бы впечатляюще это ни выглядело, добавление модификатора `suspend` к обычной функции не превращает ее волшебным образом в неблокирующую функцию. Это еще не все. Вот пример с функцией `fetchProfile`:

```
suspend fun fetchProfile(id: String) = withContext(Dispatchers.Default) {
    // та же реализация, что и у исходной функции fetchProfile,
    // которая возвращает экземпляр Profile
}
```

Функция `fetchProfile(...)` использует функцию `withContext` из фреймворка сопрограмм, которая принимает `CoroutineContext` в качестве параметра. В данном случае

мы предоставляем Dispatchers.Default в качестве контекста. Почти каждый раз при использовании withContext мы будем предоставлять только диспетчер.

Поток, который будет выполнять тело функции withContext, определяется предоставленным диспетчером. Например, при использовании Dispatchers.Default это будет один из потоков пула, предназначенного для счетных задач. В случае с Dispatchers.Main это будет основной поток.

Почему и как функция fetchProfile приостанавливает выполнение? Это деталь реализации withContext и сопрограмм в целом.

Самая важная концепция, которую нужно запомнить, проста: сопрограмма, вызывающая функцию, поддерживающую возможность приостановки, *может* приостановить выполнение. На языке сопрограмм это значит, что она достигает точки приостановки.

Почему мы сказали "*может* приостановить"? Представьте, что в реализации функции fetchProfile вы проверяете, есть ли у вас связанный профиль в кэше. Если у вас есть данные в кэше, вы можете сразу же их вернуть. Тогда приостанавливать выполнение внешней сопрограммы не нужно[9].

Существует несколько способов создания функции, поддерживающей возможность приостановки. Использование функции withContext — лишь один из них, хотя, вероятно, самый распространенный.

Резюме

- Сопрограммы всегда запускаются из CoroutineScope. Говоря языком структурированного параллелизма, CoroutineScope — это родитель, а сами сопрограммы — его потомки. CoroutineScope может быть потомком существующего CoroutineScope. Смотрите следующую главу, где рассказывается, как получить или создать CoroutineScope.
- CoroutineScope можно рассматривать как корневую сопрограмму. На самом деле, все, где есть Job, технически можно считать сопрограммой. Единственная разница заключается в предполагаемом использовании. Область видимости предназначена для "охвата" дочерних сопрограмм. Как было показано в начале этой главы, отмена области видимости приводит к отмене всех ее дочерних сопрограмм.
- launch — это функция запуска сопрограмм, возвращающая экземпляр Job. Она работает по принципу "выстрелил и забыл".
- async — это функция запуска сопрограмм, которая может возвращать значения, очень похожие на Promise и Future. Она возвращает экземпляр Deferred<T>, представляющий собой специализированный Job.

[9] Мы покажем, как это сделать, в *главе 8*.

- `Job` — это дескриптор жизненного цикла сопрограммы.
- Контекст только что созданной сопрограммы, запускаемой функциями `launch` или `async`, контекст сопрограмм наследуется от контекста области видимости и от контекста, передаваемого в качестве параметра (предоставляемого контекста) — последний имеет приоритет над первым. Один элемент контекста всегда создается заново — это `Job` сопрограммы.

  Например:

```
launch(Dispatchers.Main) {
    async {
        // наследует контекст родителя, поэтому отправляется в основной поток
    }
}
```

- Функция, поддерживающая возможность приостановки, обозначает функцию, которая не может вернуть управление сразу же. Используя функцию `withContext` и соответствующий `Dispatcher`, любую блокирующую функцию можно превратить в неблокирующую функцию, допускающую приостановку.
- Обычно сопрограмма состоит из нескольких вызовов функций, поддерживающих возможность приостановки. Каждый раз, когда вызывается такая функция, достигается точка приостановки. Выполнение сопрограммы останавливается в каждой из этих точек до тех пор, пока не будет возобновлено[10].

И последнее: *область видимости* и *контекст* — новые понятия. Это всего лишь части механизма сопрограмм. В следующей главе будут рассмотрены другие темы, такие как *обработка исключений* и *согласованная отмена*.

[10] Механизм сопрограммы возобновляет сопрограмму, когда функция, вызвавшая ее приостановку, завершает работу.

ГЛАВА 8

Структурированный параллелизм и сопрограммы

В предыдущей главе мы представили новую парадигму асинхронного программирования — сопрограммы. При использовании сопрограмм важно знать, как правильно применять функции, поддерживающие возможность приостановки; мы рассмотрим данную тему в этой главе. Поскольку большинству программ приходится иметь дело с обработкой и отменой исключений, мы рассмотрим и эти вопросы: вы увидите, что у сопрограмм есть собственный набор правил, о которых следует знать.

В первом разделе рассказывается об идиоматическом использовании функций, поддерживающих возможность приостановки.

В качестве примера мы возьмем туристическое приложение, чтобы сравнить две реализации: одну — на базе потоков, вторую — на базе функций, допускающих приостановку, — и сопрограмм. Вы увидите, как это сравнение подчеркивает мощь сопрограмм в некоторых ситуациях.

Как и в большинстве мобильных приложений, в примере с походом требуется *механизм отмены*. Мы расскажем все, что вам нужно знать об отмене с помощью сопрограмм. Чтобы подготовиться к большинству ситуаций, мы рассмотрим *параллельную декомпозицию* и *супервизию*.

Используя эти концепции, при необходимости можно реализовать сложную параллельную логику.

В конце главы объясняется обработка исключений с помощью сопрограмм.

Функции, поддерживающие возможность приостановки

Представьте, что мы разрабатываем приложение, которое поможет пользователям планировать, отслеживать, рисовать и делиться информацией о походах. Наши пользователи должны иметь возможность перейти к любому из походов, в который они уже ходили или идут сейчас. Перед тем, как отправиться в поход, пригодятся базовые статистические данные, например:

- общее расстояние;

- продолжительность последнего похода по времени и расстояние;
- текущая погода на выбранном маршруте;
- любимые маршруты.

Такое приложение потребует взаимодействий между клиентом и сервером (серверами) для получения метеорологических данных и информации о пользователях. Как хранить данные для такого приложения?

Мы можем хранить их локально для последующего использования или на удаленных серверах (это называется *стратегиями долговременного хранения*). Более длительные задачи, особенно сетевые взаимодействия или задачи ввода-вывода, можно формировать с помощью фоновых заданий, таких как чтение из базы данных, локального файла или сообщения в формате protobuf; или запрос к удаленному серверу. По сути, чтение данных с хост-устройства всегда будет осуществляться быстрее, нежели чтение тех же данных из сети.

Таким образом, извлекаемые данные могут поступать с разной скоростью, в зависимости от характера запроса. Рабочая логика должна быть достаточно устойчивой и гибкой, чтобы поддерживать и преодолевать эту ситуацию, и достаточно жесткой, чтобы справляться с обстоятельствами, находящимися за пределами нашего контроля или даже осознания.

Настройка места действия

Нам нужно создать функциональность, которая позволит пользователям получать информацию о своих любимых походах наряду с текущими погодными условиями по каждому из них.

Мы уже предоставили библиотечный код приложения, описанного в начале главы. Ниже приведен набор доступных нам классов и функций:

```
data class Hike(
   val name: String,
   val miles: Float,
   val ascentInFeet: Int)

class Weather // реализация удалена для краткости

data class HikeData(val hike: Hike, val weather: Weather?)
```

`Weather` не является классом данных Kotlin, потому что нам нужно имя типа для атрибута погоды `HikeData` (если бы мы объявили `Weather` как класс данных без предоставления атрибутов, код бы не скомпилировался).

В данном примере `Hike` — это только:

1. Имя.
2. Общее количество миль.
3. Общая протяженность восхождения в футах.

HikeData связывает объект Hike с экземпляром Weather, *допускающим значение null* (если мы по какой-то причине не смогли получить данные о погоде).

Нам также предоставляются методы для получения списка Hike с учетом идентификатора пользователя наряду с данными о погоде:

```
fun fetchHikesForUser(userId: String): List<Hike> {
    // реализация удалена для краткости
}

fun fetchWeather(hike: Hike): Weather {
    // реализация удалена для краткости
}
```

Эти две функции могут быть длительными операциями, такими как запросы к базе данных или API. Чтобы избежать блокировки UI-потока при получении списка походов или текущей информации о погоде, мы будем использовать функции, поддерживающие возможность приостановки.

Мы считаем, что лучший способ понять, как использовать такие функции, — это сравнить "традиционный" подход с использованием потоков и Handler и реализацию с использованием функций, поддерживающих возможность приостановки, и сопрограмм.

Сначала мы продемонстрируем, что в некоторых ситуациях у традиционного подхода есть свои ограничения, и преодолеть их непросто. Затем мы покажем, как использование функций, поддерживающих возможность приостановки, и сопрограмм меняет способ реализации асинхронной логики и как решить все проблемы, с которыми мы столкнулись при использовании традиционного подхода.

Начнем с реализации на базе потоков.

Традиционный подход с использованием *java.util.concurrent.ExecutorService*

Функции fetchHikesForUser и fetchWeather следует вызывать из фонового потока. В Android это можно сделать разными способами. Конечно, в Java есть традиционная библиотека Thread и фреймворк Executor. В стандартной библиотеке Android есть (теперь устаревшие) классы AsyncTask, HandlerThread, а также класс ThreadPoolExecutor.

Среди всех этих возможностей нам нужно выбрать лучшую реализацию с точки зрения выразительности, удобочитаемости и контроля. По этим причинам мы решили использовать фреймворк Executor.

Предположим, что в ViewModel мы используем один из фабричных методов для ExecutorService из класса Executors, чтобы вернуть ThreadPoolExecutor для выполнения асинхронных операций с использованием традиционной модели на базе потоков.

Ниже мы выбрали пул *занимания работ*. По сравнению с простым пулом потоков с блокирующей очередью пул занимания работ может уменьшить конкуренцию, сохраняя при этом активное целевое количество потоков. Идея заключается в поддержке достаточного количества очередей работ, чтобы одна из задач переполненного рабочего процесса[1] могла быть занята другим, менее переполненным рабочим потоком:

```
class HikesViewModel : ViewModel() {
    private val ioThreadPool: ExecutorService =
        Executors.newWorkStealingPool(10)

    fun fetchHikesAsync(userId: String) {
        ioThreadPool.submit {
            val hikes = fetchHikesForUser(userId)
            onHikesFetched(hikes)
        }
    }

    private fun onHikesFetched(hikes: List<Hike>) {
        // Продолжаем с оставшейся частью логики компонента ViewModel.
        // Осторожно, этот код выполняется из фонового потока.
    }
}
```

При выполнении операций ввода-вывода разумно иметь 10 потоков даже на устройствах под управлением Android. В случае с `Executors.newWorkStealingPool` фактическое количество потоков динамически увеличивается и сжимается в зависимости от нагрузки. Заметьте, однако, что пул занимания работ не дает никаких гарантий относительно порядка выполнения задач.

Мы также могли бы использовать примитивный Android-класс `ThreadPoolExecutor`. Точнее, можно было бы создать пул потоков:

```
private val ioThreadPool: ExecutorService =
    ThreadPoolExecutor(
        4,      // начальный размер пула
        10,     // максимальный размер пула
        1L,
        TimeUnit.SECONDS,
        LinkedBlockingQueue()
    )
```

[1] При выполнении счетных задач, рабочий поток ограничен возможностями ядра процессора.

Тогда вариант использования выглядел бы точно так же. Даже при наличии незначительных отличий от первоначально созданного нами пула занимания работ, здесь важно отметить, как отправлять задачи в пул потоков.

Использование пула потоков только для функции `fetchHikesForUser` может оказаться излишним, особенно если она не вызывается параллельно для разных пользователей. Рассмотрим остальную часть реализации, в которой используется объект `ExecutorService` для более сложной параллельной работы, как показано в следующем коде:

```
class HikesViewModel : ViewModel() {
    // другие атрибуты
    private val hikeDataList = mutableListOf<HikeData>()
    private val hikeLiveData = MutableLiveData<List<HikeData>>()

    fun fetchHikesAsync(userId: String) { // содержимое скрыто }

    private fun onHikesFetched(hikes: List<Hike>) {
        hikes.forEach { hike ->
            ioThreadPool.submit {
                val weather = fetchWeather(hike)          ❶
                val hikeData = HikeData(hike, weather)    ❷
                hikeDataList.add(hikeData)                ❸
                hikeLiveData.postValue(hikeDataList)      ❹
            }
        }
    }
}
```

Для каждого похода передается новая задача, которая:

❶ Получает информацию о погоде.

❷ Сохраняет объекты `Hike` и `Weather` в контейнере `HikeData`.

❸ Добавляет экземпляр `HikeData` во внутренний список.

❹ Уведомляет представление об изменении `hikeDatalist`, что передает недавно обновленное состояние данных списка.

Мы явно оставили распространенную ошибку в предыдущем коде. Заметили ее? Хотя он и так работает нормально, представьте, что мы добавляем общедоступный метод, чтобы добавить новый поход:

```
fun addHike(hike: Hike) {
    hikeDataList.add(HikeData(hike, null))
    // затем извлекаем Weather и уведомляем представление с помощью hikeLiveData
}
```

На этапе ❸ в методе onHikesFetched мы добавили новый элемент в hikeDataList из одного из фоновых потоков ioThreadPool. Что могло пойти не так с настолько безобидным методом?

Можно попытаться вызвать метод addHike из основного потока, в то время как hikeDataList изменяется фоновым потоком.

Поддержка потока, из которого будет вызываться общедоступный метод addHike, ничем не обеспечена. В Kotlin изменяемый список в JVM поддерживается классом ArrayList. Однако он не является потокобезопасным. На самом деле это не единственная наша ошибка. hikeDataList публикуется некорректно — нет гарантии, что на этапе ❹ фоновый поток увидит обновленное значение hikeDataList. Здесь отсутствует связь *happens before*[2] из модели памяти Java — фоновый поток может не увидеть актуальное состояние hikeDataList, даже если основной поток заранее поместил новый элемент в список.

Следовательно, итератор в цепочке onHikesFetched выбросит исключение ConcurrentModificationException, когда поймет, что коллекция "волшебным образом" была изменена. В данном случае заполнение hikeDataList из фонового потока небезопасно (рис. 8.1).

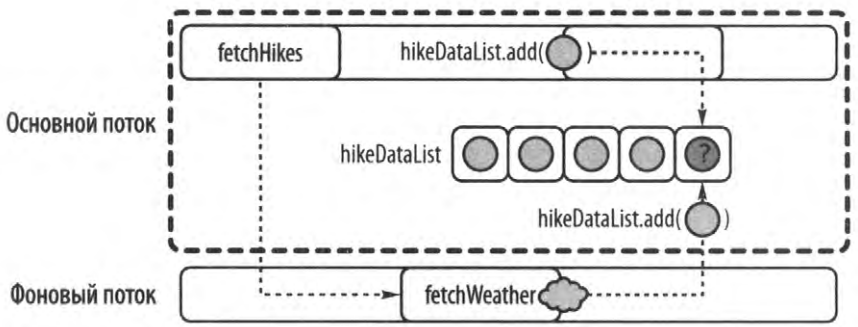


Рис. 8.1. Метод addHike добавляется к hikeDataList, которая уже модифицируется в фоновом потоке

Пристрастие к этому паттерну, даже если он и безопасен, увеличивает вероятность того, что привычка возьмет верх над чувствительностью и что в течение одного и того же дня, недели или месяца эта ошибка будет повторяться при менее безопасных обстоятельствах. Посмотрите на других членов команды, имеющих доступ к редактированию этого кода, и увидите, что вы быстро теряете контроль.

Потокобезопасность имеет значение каждый раз, когда несколько потоков пытаются одновременно получить доступ к одному и тому же ресурсу, и это трудно сделать правильно. Вот почему *использование основного потока*[3] *по умолчанию* считается хорошей практикой.

---

[2] См. Гетц Б. и др. Java Concurrency на практике. — СПб.: Питер, 2020. — Раздел 16.2.2.

[3] Мы упоминали об этом в *главе 5*. В данном случае это означает, что мы добавляем новый элемент в hikeDataList из основного потока.

Итак, как же это сделать? Можно ли заставить фоновый поток сказать основному потоку "добавлять этот элемент в этот список всякий раз, когда это возможно, а затем уведомлять представление с помощью обновленного списка HikeData"? Для этой цели можно использовать удобные классы HandlerThread и Handler.

Вспомним, что такое *HandlerThread*

HandlerThread — это поток, к которому привязан "цикл сообщений". Это реализация паттерна проектирования "производитель — потребитель", где HandlerThread — это потребитель. Обработчик находится между фактической очередью сообщений и другими потоками, которые могут отправлять новые сообщения. "Под капотом" цикл, потребляющий очередь сообщений, создается с помощью класса Looper. HandlerThread завершает работу, когда вы вызываете его методы quit или quickSafely. Если перефразировать документацию Android, можно сказать, что метод quit вызывает завершение цикла обработчика без обработки каких-либо сообщений в очереди. Метод quitSafely вызывает завершение работы цикла обработчика, как только будут обработаны все оставшиеся сообщения в очереди, которые уже должны быть доставлены.

Будьте очень осторожны, не забывая останавливать HandlerThread. Например, представьте, что вы запускаете его в жизненном цикле активности (скажем, в методе фрагмента, onCreate). Если повернуть устройство, то активность будет удалена, а затем создана снова. После этого создается и запускается новый экземпляр HandlerThread, в то время как старый все еще работает, что приводит к серьезной утечке памяти (рис. 8.2).

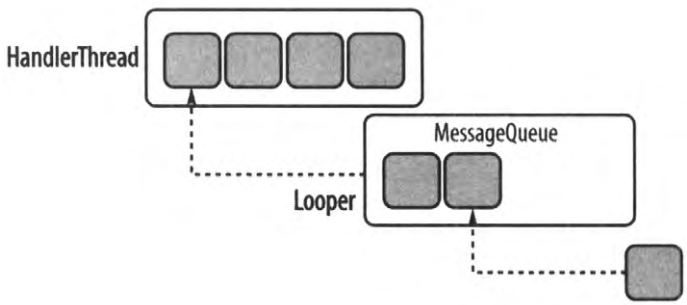


Рис. 8.2. HandlerThread потребляет задачи, поступающие из очереди MessageQueue

В Android HandlerThread — это основной поток. Поскольку создание обработчика для передачи сообщений в основной поток — очень распространенный случай, в классе Looper существует статический метод для получения ссылки на экземпляр Looper основного потока. Используя обработчик, можно передать объект, реализующий интерфейс Runnable, чтобы выполнить его в потоке, к которому присоединен экземпляр Looper, ассоциируемый с обработчиком. Вот как выглядит соответствующая сигнатура в Java:

```
public final boolean post(@NonNull Runnable r) { ... }
```

Поскольку у объекта, реализующего интерфейс Runnable, только один абстрактный метод run, его можно изящно и синтаксически "подсластить" с помощью лямбда-выражения, как показано в следующем коде:

```
// Прямая трансляция в Kotlin (хотя и не идиоматично)
handler.post(object: Runnable {
        override fun run() {
            // содержимое метода run
        }
    }
)

// .. его можно изящно упростить и получить:
handler.post {
    // содержимое метода run
}
```

На практике нужно сделать следующее:

```
val handler: Handler = Handler(Looper.getMainLooper())
```

Затем можно использовать обработчик цикла из предыдущего примера, как показано в следующем коде:

```
class HikesViewModel : ViewModel() {
    private val ioThreadPool: ExecutorService = Executors.newWorkStealing
Pool(10)
    private val hikeDataList = mutableListOf<HikeData>()
    private val hikeLiveData = MutableLiveData<List<HikeData>>()
    private val handler: Handler = Handler(Looper.getMainLooper())

    private fun onHikesFetched(hikes: List<Hike>) {
        hikes.forEach { hike ->
            ioThreadPool.submit {
                val weather = fetchWeather(hike)
                val hikeData = HikeData(hike, weather)

                // Здесь мы размещаем объект, реализующий интерфейс Runnable
                handler.post {
                    hikeDataList.add(hikeData)          ❶
                    hikeLiveData.value = hikeDataList   ❷
                }
            }
        }
    }
```

```
    // другие методы удалены для краткости
}
```

На этот раз мы передаем объект, реализующий интерфейс `Runnable`, в основной поток, в котором:

❶ Новый экземпляр `HideData` добавляется в `hikeDataList`.

❷ Мы присваиваем `hikeLiveData` `hikeDataList` в качестве обновленного значения. Обратите внимание, что здесь можно использовать удобочитаемый и интуитивно понятный оператор присваивания: `hikeLiveData.value = ...` — это лучше, чем `hikeLiveData.postValue(..)`. Это связано с тем, что объект, реализующий интерфейс `Runnable`, будет выполняться из основного потока — метод `postValue` полезен только при обновлении значения `LiveData` из фонового потока.

При этом все методы доступа для `hikeDataList` ограничены основным потоком (рис. 8.3), так устраняются все возможные проблемы параллелизма.

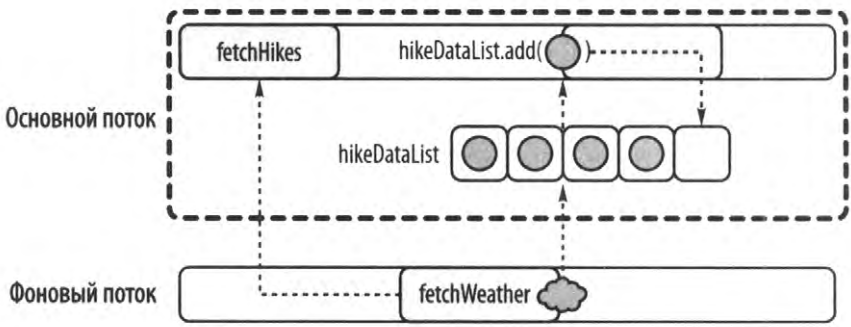


Рис. 8.3. Основной поток может получить доступ только к `hikeDataList`

Вот и все, что касается "традиционного" подхода. Чтобы сделать практически то же самое, можно было использовать другие библиотеки, например *RxJava/RxKotlin* и *Arrow*. Логика состоит из нескольких этапов. Вы запускаете первый этап, предоставляя ему функцию обратного вызова, содержащую набор инструкций для запуска после завершения фонового задания. Каждый этап связан со следующим кодом, находящимся в функциях обратного вызова. Мы обсуждали это в *главе 6* и надеемся, что выявили некоторые потенциальные ловушки и предоставили вам инструменты, которые помогут их избежать.

Интересно, что сложность функции обратного вызова в этом примере не кажется проблемой: все делается с помощью двух методов — `Handler` и `ExecutorService`. Однако в следующем сценарии возникает коварная ситуация.

Пользователь переходит к списку походов, после чего вызывается метод `fetchHikesAsync`. Пользователь только что установил приложение на новое устройство; таким образом, история не кэшируется, поэтому приложению необходим доступ к удаленным API для получения последних данных из удаленной службы.

Предположим, что беспроводная сеть работает медленно, но не настолько, чтобы вызывать ошибки времени ожидания ввода-вывода. Представление по-прежнему показывает, что список обновляется, и пользователь может подумать, будто на самом деле произошла скрытая ошибка, и повторит попытку выборки данных (что можно сделать, используя некий пользовательский интерфейс обновления, например `SwipeRefreshLayout`, явную кнопку обновления или даже просто с помощью навигации для повторного входа в пользовательский интерфейс, предполагая, что выборка будет вызываться неявно).

К сожалению, ничто в нашей реализации не предполагает этого. При вызове метода `fetchHikesAsync` запускается рабочий процесс, который нельзя остановить. Представьте себе худший вариант: каждый раз, когда пользователь переходит назад и снова входит в представление списка походов, запускается новый рабочий процесс. Это явно неудачный дизайн.

Одним из возможных решений может стать механизм отмены. Мы могли бы реализовать его, гарантируя, что каждый новый вызов `fetchHikesAsync` отменяет все предыдущие выполняющиеся или ожидающие вызовы. Кроме того, можно отказаться от новых вызовов `fetchHikesAsync`, пока предыдущий вызов все еще выполняется. Чтобы реализовать это в данном контексте, нужно все тщательно обдумать.

Механизм отмены здесь не такой простой, как в других потоках, потому нужно убедиться, что *каждый* фоновый поток действительно останавливает выполнение.

Как вы знаете из предыдущей главы, сопрограммы и функции, поддерживающие возможность приостановки, могут отлично подойти в подобных обстоятельствах. Мы выбрали это приложение для походов, потому что у нас есть прекрасная возможность использовать функции, допускающие приостановку.

Использование приостанавливаемых функций и сопрограмм

Напоминаем, что сейчас мы реализуем ту же самую логику; но на этот раз будем использовать функции, поддерживающие возможность приостановки, и сопрограммы.

Мы объявляем функцию, поддерживающую возможность приостановки, когда функция может не сразу вернуть управление. Следовательно, любую блокирующую функцию можно переписать как функцию, допускающую приостановку.

Функция `fetchHikesForUser` — хороший пример, поскольку она блокирует вызывающий поток до тех пор, пока не вернет список экземпляров `Hike`. Поэтому ее можно выразить как функцию, поддерживающую возможность приостановки, как показано в следующем коде:

```
suspend fun hikesForUser(userId: String): List<Hike> {
    return withContext(Dispatchers.IO) {
        fetchHikesForUser(userId)
    }
}
```

Нам пришлось выбрать другое имя для функции. В этом примере блокирующие вызовы по соглашению оформлены префиксом "fetch".

Точно так же, как показано в примере 8.1, можно объявить эквивалент для функции `fetchWeather`.

Пример 8.1. `fetchWeather` как функция, поддерживающая возможность приостановки

```
suspend fun weatherForHike(hike: Hike): Weather {
    return withContext(Dispatchers.IO) {
        fetchWeather(hike)
    }
}
```

Эти функции представляют собой обертки соответствующих блокирующих аналогов. При вызове из сопрограммы `Dispatcher`, передаваемый функции `withContex`, определяет, в каком пуле потоков выполняется блокирующий вызов. Здесь `Dispatchers.IO` идеально подходит и очень похож на пул занимания работ, который мы видели ранее.

После того как мы обернули блокирующие вызовы в блоки, такие как функция `weatherForHike`, можно использовать эти функции в сопрограммах, в чем вы скоро убедитесь.

На самом деле существует соглашение, призванное облегчить всем жизнь: *функция, поддерживающая возможность приостановки. никогда не блокирует вызывающий поток*. В случае с `weatherForHike` так и оно есть, поскольку независимо от того, какой поток вызывает эта функция из сопрограммы, оператор `withContext(Dispatchers.IO)` вызывает выполнение для перехода к другому потоку[4].

Все, что мы сделали с помощью обратного вызова, теперь можно уместить в один общедоступный метод `update`, который выглядит как процедурный код. Это допустимо благодаря функциям, поддерживающим возможность приостановки, как показано в примере 8.2.

Пример 8.2. Использование функций, поддерживающих возможность приостановки, в компоненте ViewModel

```
class HikesViewModel : ViewModel() {
    private val hikeDataList = mutableListOf<HikeData>()
    private val hikeLiveData = MutableLiveData<List<HikeData>>()

    fun update() {
        viewModelScope.launch {                                    ❶
```

[4] Если только `Dispatchers.IO` не страдает от застревания потоков, что крайне маловероятно.

```
            /* Этап 1: получаем список походов */
            val hikes = hikesForUser("userId")                    ❷

            /* Этап 2: для каждого похода мы получаем прогноз погоды,
               оборачиваем в контейнер, обновляем hikeDataList,
               а затем уведомляем слушателей представления, обновив LiveData. */
            hikes.forEach { hike ->                               ❸
                launch {
                    val weather = weatherForHike(hike)            ❹
                    val hikeData = HikeData(hike, weather)
                    hikeDataList.add(hikeData)
                    hikeLiveData.value = hikeDataList
                }
            }
        }
    }
}
```

Теперь рассмотрим этот пример подробно.

❶ Когда мы вызываем функцию `update`, она незамедлительно запускает сопрограмму, используя функцию `launch`. Как известно, сопрограмма никогда не запускается на ровном месте. Как было показано в *главе 7*, она всегда должна запускаться в области видимости сопрограмм. Здесь мы используем `viewModelScope`.

Откуда эта область видимости? Команда Android Jetpack из компании Google знает, что для использования Kotlin и сопрограмм требуется область видимости. Чтобы облегчить всем жизнь, члены команды осуществляют поддержку Android KTX, представляющему собой набор расширений Kotlin для платформы Android и других API. Цель состоит в том, чтобы использовать идиомы Kotlin, сохраняя при этом изящную интеграцию с фреймворком Android. Они используют функции-расширения, лямбда-выражения, значения параметров по умолчанию и сопрограммы. Android KTX состоит из нескольких библиотек. В этом примере мы использовали библиотеку *lifecycle-viewmodel-ktx*. Чтобы воспользоваться ею в своем приложении, нужно добавить в зависимости, перечисленные в `build.gradle` (используйте более новую версию, если она доступна), следующий код: `implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.2.0"`.

❷ Строка `val hikes = hikesForUser("userId")` — это первая точка приостановки. Сопрограмма, запускаемая функцией `launch`, останавливается до тех пор, пока функция `hikesForUser` не вернет управление.

❸ У нас есть список экземпляров `Hike`. Теперь можно *параллельно* получать данные о погоде для каждого из них. Мы можем использовать цикл и запускать новую сопрограмму для каждого похода с помощью метода `launch`.

❹ `val weather = weatherForHike(hike)` — еще одна точка приостановки. Каждая сопрограмма, запущенная в цикле `for`, достигнет этой точки.

Подробнее рассмотрим сопрограмму, запускаемую для каждого экземпляра `Hike`, в следующем коде:

```
launch {
    val weather = weatherForHike(hike)
    val hikeData = HikeData(hike, weather)
    hikeDataList.add(hikeData)
    hikeLiveData.value = hikeDataList
}
```

Поскольку родительская область видимости (`viewModelScope`) по умолчанию соответствует основному потоку, каждая отдельная строка в блоке `launch` выполняется в основном потоке, за исключением содержимого функции `weatherForHike`, которая использует `Dispatchers.IO` (см. пример 8.1). Присваивание значения переменной `weather` выполняется в основном потоке. Таким образом, использование `hikeDataList` ограничено основным потоком — проблем с потокобезопасностью не возникает. Что касается `hikeLiveData`, то можно использовать метод записи `value` (а так как мы в Kotlin, это означает оператор присваивания) вместо `postValue`, поскольку мы знаем, что вызываем его из основного потока.

При использовании области видимости всегда нужно осознавать, как она управляет сопрограммами, и в особенности знать, какой диспетчер она использует. В следующем коде показано, как она объявляется в исходном коде библиотеки:

```
val ViewModel.viewModelScope: CoroutineScope
    get() {
        val scope: CoroutineScope? = this.getTag(JOB_KEY)
        if (scope != null) {
            return scope
        }
        return setTagIfAbsent(
        JOB_KEY,
        CloseableCoroutineScope(
            SupervisorJob() + Dispatchers.Main.immediate))
    }
```

Как видно из этого примера, `viewModelScope` объявляется как свойство-расширение класса `ViewModel`. Даже если этот класс не имеет абсолютно никакого понятия о `CoroutineScope`, объявляя его таким образом, мы активируем синтаксис из нашего примера. После этого мы обращаемся ко внутреннему хранилищу, чтобы прове-

рить, была ли создана область видимости. Если нет, создается новая область с помощью `CloseableCoroutine.Scope(..)`[5]. Не обращайте внимания на `SupervisorJob` — мы объясним его роль позже, когда будем обсуждать отмену. Особую важность здесь представляет `Dispatchers.Main.immediate` (вариант `Dispatcher.Main`), который выполняет сопрограммы сразу же, когда они запускаются из основного потока. Следовательно, эта область видимости по умолчанию соответствует основному потоку. Это важная информация, которую нужно знать, чтобы двигаться дальше.

Сравнение приостанавливаемых и традиционной многопоточности: итоги

Благодаря функциям, поддерживающим возможность приостановки, асинхронную логику можно писать, как процедурный код. Поскольку компилятор Kotlin генерирует все необходимые функции обратного вызова и шаблонный код "под капотом", код, который вы пишете, используя механизм отмены, может быть гораздо короче[6].

Например, области видимости сопрограмм, которая использует `Dispatchers.Main`, не нужен `Handler` или другие примитивные типы, поддерживающие возможность взаимодействий сопрограмм, чтобы передавать данные, поступающие в фоновый поток и обратно, в основной поток, как это все еще случается в чисто многопоточных окружениях (без сопрограмм). В действительности все проблемы, которые у нас были при использовании подхода на основе потоков, теперь прекрасно решаются с помощью сопрограмм, включая механизм отмены.

Код, использующий сопрограммы и функции, поддерживающие возможность приостановки, также может быть более удобочитаемым, поскольку здесь может быть гораздо меньше неявных или косвенных инструкций (таких как вложенные вызовы или экземпляры SAM, как описано в *главе 6*). Более того, IntelliJ и Android Studio выделяют эти вызовы специальным значком на полях.

В этом разделе мы лишь поверхностно коснулись темы отмены. В следующем разделе рассказывается все, что нужно знать об отмене с помощью сопрограмм.

Отмена

Отмена задач является важной частью приложения Android. Когда пользователь впервые переходит к представлению, отображающему список походов вместе со статистикой и погодой, из компонента ViewModel запускается приличное количе-

---

[5] Это просто подкласс обычной области видимости, который вызывает метод `coroutineContext.cancel()` в методе `close()`.

[6] Обратите внимание, что материал о подходе, в котором используются функции, поддерживающие возможность приостановки, короче (три с половиной страницы по сравнению с семью страницами, посвященными традиционному подходу), вероятно, потому что функции, поддерживающие возможность приостановки, — это более простое (и легко объяснимое) решение.

ство сопрограмм. Если по какой-то причине пользователь решит выйти из представления, то, вероятно, запущенные представлением задачи выполняются впустую. Если, конечно, пользователь не вернется обратно к представлению, но делать такие предположения небезопасно. Чтобы не тратить ресурсы впустую, в этом сценарии рекомендуется отменить все текущие задачи, связанные с представлениями, которые больше не нужны. Это хороший пример, который можно реализовать самостоятельно, как часть архитектуры приложения. Есть еще один вид отмены: который происходит, когда случается что-то плохое. Итак, здесь мы будем различать два вида отмены.

Спроектированная отмена.

Например, задача, которая отменяется после того, как пользователь нажимает кнопку **Отмена** в собственном или произвольном пользовательском интерфейсе.

Отмена в случае сбоя.

Например, отмена, вызванная исключениями: либо преднамеренно (исключение было выброшено), либо неожиданно (исключение не было обработано).

Помните о них — вы увидите, что сопрограммы обрабатывают их по-разному.

Жизненный цикл сопрограмм

Чтобы понять, как работает отмена, нужно знать, что у сопрограммы есть жизненный цикл, который показан на рис. 8.4.

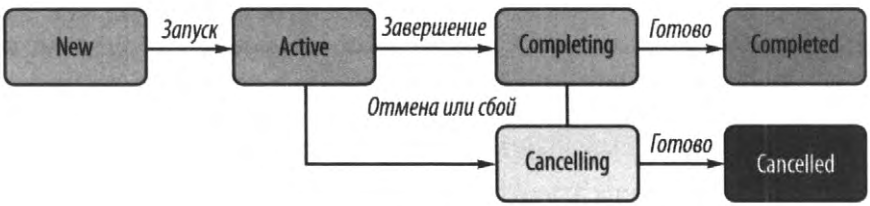


Рис. 8.4. Жизненный цикл сопрограммы

Когда мы создаем сопрограмму, например используя функцию `launch {..}` без дополнительного контекста или аргументов, она создается в состоянии `Active`. Это означает, что она запускается сразу после вызова метода `launch`, т. е. *немедленно*. В некоторых ситуациях у вас может возникнуть желание запустить сопрограмму *отложенным образом*. Это означает, что она ничего не сделает, пока вы не запустите ее вручную. Для этого функции `launch` и `async` могут принимать именованный аргумент "start" типа `CoroutineStart`. Значение по умолчанию — `CoroutineStart.DEFAULT` (немедленный запуск), но можно использовать и `CoroutineStart.LAZY`, как в следующем коде:

```
val job = scope.launch(start = CoroutineStart.LAZY) { ... }
// делаем что-то
job.start()
```

Не забудьте вызвать функцию job.start(), потому что при отложенном запуске сопрограмму необходимо запускать явно[7]! Не нужно делать это по умолчанию, т. к. сопрограмма создается в состоянии Active.

Когда сопрограмма завершает работу, она остается в состоянии Completing до тех пор, пока все ее потомки не достигнут состояния Completed (*см. главу 7*). Только после этого она переходит в состояние Completed. "Взломаем" исходный код и посмотрим:

```
viewModelScope.launch {
    launch {
        fetchData() // на это может уйти какое-то время
    }
    launch {
        fetchOtherData()
    }
}
```

viewModelScope.launch завершает работу почти мгновенно: он запускает только две дочерние сопрограммы и больше ничего не делает. Он быстро достигает состояния Completing и переходит в состояние Completed только после завершения дочерних сопрограмм.

Состояние *Canceled*

Находясь в состоянии Active или Completing, если выбрасывается исключение или логика вызывает метод cancel(), сопрограмма переходит в состояние Canceling. Тогда, при необходимости, выполняется очистка. Сопрограмма остается в этом состоянии до тех пор, пока очистка не завершится. Только после этого сопрограмма перейдет в состояние Canceled.

Состояния находятся в Job

"Под капотом" все эти состояния жизненного цикла находятся в Job сопрограммы. У Job нет свойства с именем "state" (значения которого могут варьироваться от NEW до COMPLETED). Состояние представлено тремя логическими значениями (флагами): isActive, isCancelled и isCompleted. Каждое состояние представлено комбинацией этих флагов, как показано в табл. 8.1.

Таблица 8.1. Состояния Job

Состояние	isActive	isCompleted	isCancelled
New (необязательное начальное состояние)	false	false	false
Active (начальное состояние по умолчанию)	true	false	false

[7] При отложенном запуске сопрограмма находится в состоянии New. Только после вызова job.start() она перемещается в состояние Active. Вызов job.join() также запускает сопрограмму.

Таблица 8.1 (окончание)

Состояние	isActive	isCompleted	isCancelled
Completing (переходное состояние)	true	False	false
Cancelling (переходное состояние)	false	false	true
Cancelled (финальное состояние)	false	true	true
Completed (финальное состояние)	false	true	false

Как видите, используя только эти логические значения, невозможно отличить состояние Completing от состояния Active. В большинстве случаев вас будет волновать значение конкретного флага, а не само состояние. Например, в случае с isActive вы фактически проверяете состояния Active и Completing одновременно. Подробности об этом *см. в следующем разделе.*

Отмена сопрограммы

Рассмотрим пример, в котором у нас есть сопрограмма, просто выводящая в консоль фразу "job: I'm working.." два раза в секунду. Родительская сопрограмма немного ждет, прежде чем отменить ее:

```
val startTime = System.currentTimeMillis()
val job = launch(Dispatchers.Default) {
    var nextPrintTime = startTime
    while (true) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
delay(1200)
println("main: I'm going to cancel this job")
job.cancel()
println("main: Done")
```

Видно, что у экземпляра Job, возвращаемого функцией launch, есть метод cancel(). Его название предполагает, что он отменяет работающую сопрограмму. Кстати, у экземпляра Deferred, который возвращает функция async, тоже есть этот метод, поскольку экземпляр Deferred — это специализированный экземпляр Job.

Вернемся к нашему примеру. Возможно, вы ожидаете, что этот небольшой фрагмент кода трижды выведет фразу "job: I'm working..".

На самом деле вывод будет выглядеть так:

```
job: I'm working..
job: I'm working..
job: I'm working..
main: I'm going to cancel this job
main: Done
job: I'm working..
job: I'm working..
```

Таким образом, дочерняя сопрограмма все еще работает, несмотря на отмену со стороны родителя, потому что дочерняя сопрограмма не действует согласованно. Есть несколько способов изменить это. Первый способ — периодически проверять статус отмены сопрограммы, используя `isActive`, как показано в следующем коде:

```
val job = launch(Dispatchers.Default) {
    var nextPrintTime = startTime
    while (isActive) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
```

Мы можем вызывать `isActive` таким образом, потому что это свойство-расширение `CoroutineScope`, как показано в следующем коде:

```
/**
* Возвращает true, если текущий Job все еще активен
  (он не завершен и еще не отменен).
*/
val CoroutineScope.isActive: Boolean (source)
```

Теперь, когда имеет место согласованная отмена, результат выглядит так:

```
job: I'm working..
job: I'm working..
job: I'm working..
main: I'm going to cancel this job
main: Done
```

Использование `isActive` — это просто чтение логического значения. Определение того, следует ли останавливать задание, а также настройка и выполнение этой логики лежат на вас.

Вместо `isActive` можно использовать `ensureActive`. Разница между ними заключается в том, что `ensureActive` немедленно выбрасывает исключение `CancellationException`, если задание больше не активно.

Таким образом, ensureActive является заменой следующего кода:

```
if (!isActive) {
    throw CancellationException()
}
```

Как и в случае с Thread.yield(), существует третья возможность — функция yield(). Помимо проверки состояния отмены задания основной поток освобождается и становится доступным для других сопрограмм. Это особенно полезно при выполнении ресурсоемких вычислений в сопрограмме с использованием Dispatchers.Default (или чего-то похожего). Размещая функцию yield() в стратегических местах, можно избежать исчерпания пула потоков. Другими словами, вы вряд ли хотите, чтобы сопрограмма была слишком "себялюбивой", и чтобы ядро было занято конкретными контекстуальными обязанностями в течение длительного периода времени, если эти ресурсы лучше было бы использовать в другом процессе. Чтобы быть более согласованной, "прожорливая" сопрограмма, ограниченная возможностями процессора, должна время от времени использовать функцию yield(), давая другим сопрограммам возможность запускаться.

Эти способы прерывания работы сопрограммы идеально подходят, когда отмена происходит в коде. А как быть, если вы просто делегировали часть работы сторонней библиотеке, например HTTP-клиенту?

Отмена задачи, делегированной сторонней библиотеке

OkHttp — широко распространенный HTTP-клиент для Android. Если вы не знакомы с этой библиотекой, то ниже приведен фрагмент из официальной документации для выполнения синхронного запроса с методом GET:

```
fun run() {
    val request = Request.Builder()
        .url("https://publicobject.com/helloworld.txt")
        .build()

    client.newCall(request).execute().use { response ->
      if (!response.isSuccessful)
          throw IOException("Unexpected code $response")

      for ((name, value) in response.headers) {
        println("$name: $value")
      }

      println(response.body?.string())
    }
}
```

Это довольно простой пример. `client.newCall(request)` возвращает экземпляр `Call`. Вы ставите в очередь экземпляр `Callback`, в то время как код продолжает невозмутимо выполняться. Возможна ли отмена? Да. `Call` можно отменить вручную с помощью `call.cancel()`.

Если говорить об использовании сопрограмм, то предыдущий пример — это разновидность кода, который можно писать в сопрограмме. Было бы идеально, если бы эта отмена производилась автоматически при отмене сопрограммы, в которой выполняется HTTP-запрос. В противном случае пришлось бы написать:

```
if (!isActive) {
    call.cancel()
    return
}
```

Очевидное предостережение: это засоряет код, не говоря уже о том, что вы можете забыть добавить эту проверку или разместить ее не в том месте. Должно быть решение получше.

К счастью, фреймворк сопрограмм поставляется с функциями, специально предназначенными для превращения функции, ожидающей функцию обратного вызова, в функцию, поддерживающую возможность приостановки. Они бывают нескольких видов, включая `suspendCancellableCoroutine`. Последняя предназначена для создания функции, допускающей приостановку, которая *действует согласованно, чтобы ее можно было отменить*.

В следующем коде показано, как создать такую функцию в качестве функции-расширения `Call`, которую можно отменить и которая приостанавливает выполнение до тех пор, пока вы не получите ответ на HTTP-запрос или не будет выброшено исключение:

```
suspend fun Call.await() = suspendCancellableCoroutine<ResponseBody?> {
    continuation ->

    continuation.invokeOnCancellation {
        cancel()
    }

    enqueue(object : Callback {
        override fun onResponse(call: Call, response: Response) {
            continuation.resume(response.body)
        }

        override fun onFailure(call: Call, e: IOException) {
            continuation.resumeWithException(e)
        }
    })
}
```

Если вы никогда не видели такой код, то вполне естественно, что вас пугает его отталкивающая сложность. Хорошая новость заключается в том, что данная функция полностью универсальна — ее нужно написать только один раз. Она может находиться в пакете проекта "util", если хотите, или в пакете parallelism; или просто запомните основы и используйте некую ее версию при выполнении подобных преобразований.

Прежде чем показать преимущества этого вспомогательного метода, нужно дать подробное объяснение.

В *главе 7* мы объяснили, как компилятор Kotlin генерирует экземпляр `Continuation` для каждой функции, поддерживающей возможность приостановки. Функция `suspendCancellableCoroutine` позволяет использовать этот экземпляр. Она принимает лямбда-выражение с `CancellableContinuation` в качестве приемника, как показано в следующем коде:

```
public suspend inline fun <T> suspendCancellableCoroutine(
    crossinline block: (CancellableContinuation<T>) -> Unit
): T
```

`CancellableContinuation` — это объект `Continuation`, который можно отменять. Можно зарегистрировать функцию обратного вызова, которая будет вызываться при отмене, используя `invokeOnCancellation { .. }`. В данном случае все, что нам нужно, — это отменить `Call`. Поскольку мы находимся в функции-расширении `Call`, добавляем следующий код:

```
continuation.invokeOnCancellation {
    cancel() // Call.cancel()
}
```

После того, как мы указали, что должно произойти после отмены функции, поддерживающей возможность приостановки, мы выполняем HTTP-запрос, вызывая `Call.enqueue()` и предоставляя экземпляр `Callback`. Функция "возобновляет выполнение" или "прекращает приостановку" при возобновлении соответствующего объекта `Continuation` — с помощью функции `resume` или исключения `resumeWithException`.

Когда вы получите результат HTTP-запроса, будет вызвана либо функция `onResponse`, либо функция `onFailure` предоставленного вами экземпляра `Callback`. Если вызывается функция `onResponse`, то это оптимистичный вариант. Вы получили ответ и теперь должны возобновить работу с результатом на ваш выбор. Как показано на рис. 8.5, мы выбрали тело HTTP-ответа. При пессимистичном варианте вызывается функция `onFailure`, а API `OkHttp` предоставляет вам экземпляр `IOException`.

Важно возобновить работу с этим исключением, используя `resumeWithException`. Таким образом, фреймворк сопрограмм узнает о сбое функции, допускающей приостановку, и будет передавать это событие по всей иерархии сопрограмм.

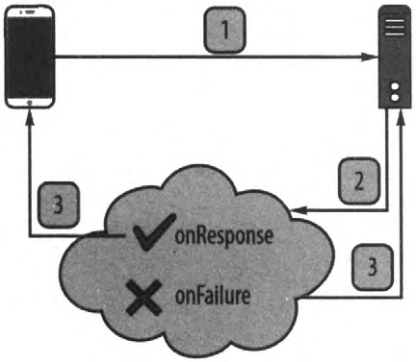


Рис. 8.5. (1) Сначала устройство отправляет HTTP-запрос на сервер. (2) Тип возвращаемого ответа будет определять, что произойдет дальше. (3) Если запрос выполнен успешно, вызывается функция onResponse. В противном случае вызывается функция onFailure

Теперь самое интересное: продемонстрируем, как использовать это в сопрограмме.

```
fun main() = runBlocking {
    val job = launch {                                    ❶
        val response = performHttpRequest()               ❷
        println("Got response ${response?.string()}")
    }
    delay(200)                                            ❸
    job.cancelAndJoin()                                   ❹
    println("Done")
}

val okHttpClient = OkHttpClient()
val request = Request.Builder().url(
    "http://publicobject.com/helloworld.txt"
).build()

suspend fun performHttpRequest(): ResponseBody? {
    return withContext(Dispatchers.IO) {
        val call = okHttpClient.newCall(request)
        call.await()
    }
}
```

❶ Начинаем с запуска сопрограммы с помощью функции launch.

❷ В сопрограмме, возвращаемой этой функцией, мы вызываем функцию performHttpRequest(), которая использует Dispatchers.IO. Она создает новый эк-

экземпляр Call, а затем вызывает функцию await(). В этот момент выполняется HTTP-запрос.

❸ Параллельно, пока в каком-то потоке Dispatchers.IO выполняется этап ❷, основной поток продолжает выполнение основного метода и сразу сталкивается с задержкой: delay(200). Сопрограмма, работающая в основном потоке, приостанавливается на 200 мс.

❹ По прошествии 200 мс мы вызываем метод job.cancelAndJoin(), который является удобным методом для job.cancel(), а затем job.join(). Следовательно, если HTTP-запрос занимает больше 200 мс, то сопрограмма, запускаемая функцией launch, все еще находится в состоянии Active. Функция performHttpRequest() еще не вернула управление. При вызове job.cancel() мы отменяем сопрограмму. Благодаря структурированному параллелизму сопрограмма знает обо всех своих дочерних элементах. Отмена передается по всей иерархии. Объект Continuation функции performHttpRequest() отменяется, как и HTTP-запрос. Если HTTP-запрос занимает менее 200 мс, то job.cancelAndJoin() не имеет никакого эффекта.

Неважно, насколько глубоко в иерархии сопрограмм выполняется HTTP-запрос. Если используется предопределенный Call.await(), то в случае отмены родительской сопрограммы запускается отмена Call.

Сопрограммы, которые действуют согласованно, чтобы их можно было отменить

Вы только что видели различные способы сделать так, чтобы сопрограмму можно было отменить. На самом деле у фреймворка сопрограмм есть соглашение: корректно работающая сопрограмма, которую можно отменить, при отмене выбрасывает исключение CancellationException. Почему? Рассмотрим следующую функцию, поддерживающую возможность приостановки:

```
suspend fun wasteCpu() = withContext(Dispatchers.Default) {
    var nextPrintTime = System.currentTimeMillis()
    while (isActive) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
```

Ее и правда можно отменить благодаря проверке с помощью isActive. Представьте, что вам нужно выполнить очистку, когда эта функция отменена. Вы знаете, что эта функция отменяется, если isActive == false, поэтому в конце можно добавить блок очистки, как показано здесь:

```
suspend fun wasteCpu() = withContext(Dispatchers.Default) {
    var nextPrintTime = System.currentTimeMillis()
```

```
    while (isActive) {
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }

    // очистка
    if (!isActive) { .. }
}
```

Иногда требуется, чтобы логика очистки находилась за пределами отмененной функции; например, если эта функция из внешней зависимости. Поэтому нужно найти способ уведомить стек вызовов об отмене данной функции. Исключения идеально подходят для этого. Вот почему сопрограммы следуют соглашению о выбросе исключения `CancellationException`. На самом деле *все* функции, поддерживающие возможность приостановки, из пакета *kotlinx.coroutiness* можно отменять, и при отмене выбрасывается исключение `CancellationException`. `withContext` — одна из таких функций, поэтому вы можете отреагировать на отмену `wasteCpu` выше в стеке вызовов, как показано в следующем коде:

```
fun main() = runBlocking {
    val job = launch {
        try {
            wasteCpu()
        } catch (e: CancellationException) {
            // обработка отмены
        }
    }
    delay(200)
    job.cancelAndJoin()
    println("Done")
}
```

При выполнении этого кода вы обнаружите перехват исключения `CancellationException`. Хотя мы никогда явно не выбрасывали исключение `CancellationException` из функции `wasteCpu()`, функция `withContext` сделала это за нас.

Выбрасывая это исключение только в случае отмены, фреймворк сопрограмм способен различать простую отмену и сбой сопрограммы. В последнем случае будет возбуждено исключение, не являющееся подтипом `CancellationException`.

Если вы хотите изучить отмену сопрограмм, то можете *определить имена* своих сопрограмм и включить отладку сопрограмм в интегрированной среде разработки, добавив параметр `Dkotlinx.coroutines.debug` виртуальной

машины. Просто добавьте элемент контекста CoroutineName: val job = launch(CoroutineName("wasteCpu")) {..}. Таким образом, при перехвате исключения CancellationException трассировка стека будет гораздо более явной, и начинаться она будет со следующей строки:

```
kotlinx.coroutines.JobCancellationException:
StandaloneCoroutine was cancelled; job="waste-
Cpu#2":StandaloneCoroutine{Cancelling}@53bd815b
```

В предыдущем примере, если вы замените функцию WasteCpu() на PerformHttpRequest() — функцию, поддерживающую возможность приостановки, которую мы создали ранее с помощью suspendCancellableCoroutine, — вы обнаружите, что было перехвачено исключение CancellationException. Таким образом, функция, созданная с помощью suspendCancellableCoroutine, также выбрасывает исключение CancellationException при отмене.

Функцию *delay()* можно отменить

Помните функцию delay()? Ее сигнатура показана в следующем коде:

```
public suspend fun delay(timeMillis: Long) {
    if (timeMillis <= 0) return // задержки нет
    return suspendCancellableCoroutine sc@ { .. }
}
```

И снова suspendCancellableCoroutine! Это означает, что везде, где вы используете функцию delay(), вы предоставляете сопрограмме или функции, допускающей приостановку, возможность отмены. На основании этого можно было бы переписать функцию wasteCpu() следующим образом:

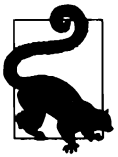
```
private suspend fun wasteCpu() = withContext(Dispatchers.Default) {
    var nextPrintTime = System.currentTimeMillis()
    while (true) {                    ❶
        delay(10)                     ❷
        if (System.currentTimeMillis() >= nextPrintTime) {
            println("job: I'm working..")
            nextPrintTime += 500
        }
    }
}
```

Обратите внимание:

❶ Мы убрали проверку с isActive.

❷ Затем мы добавили простую функцию delay() с достаточно небольшим временем приостановки (поэтому это поведение похоже на предыдущую реализацию).

Новая версия функции wasCpu() оказывается отменяемой, как и исходная функция, и при отмене выбрасывает исключение CancellationException, потому что бóльшую часть времени эта функция проводит в функции delay().



Подводя итоги, нужно сказать, что вы должны стремиться к тому, чтобы функции, поддерживающие возможность приостановки, можно было отменять. Такая функция может состоять из нескольких функций, допускающих приостановку. Все они должны быть отменяемыми. Например, если вам нужно выполнить вычисления с высокой нагрузкой на ЦП, в стратегически важных местах следует использовать функцию yield() или sureActive(). Например:

```
suspend fun compute() = withContext(Dispatchers.Default) {
    blockingCall() // обычный блокирующий вызов;
                   // надеемся, что блокировка не продержится
                   // слишком долго
    yield()                 // даем возможность отмены
    anotherBlockingCall()  // потому что, почему бы и нет?
}
```

Обработка отмены

В предыдущем разделе вы узнали, что можно реагировать на отмену с помощью оператора try/catch. Однако представьте, что в коде, обрабатывающем отмену, нужно вызвать другие функции, поддерживающие возможность приостановки. У вас может возникнуть соблазн реализовать стратегию, показанную в следующем коде:

```
launch {
    try {
        suspendCall()
    } catch (e: CancellationException) {
       // обработка отмены
       anotherSuspendCall()
    }
}
```

К сожалению, предыдущий код не компилируется. Почему? Потому что *выполнение отмененной сопрограммы нельзя приостановить*. Это еще одно правило сопрограмм. Решение — использовать withContext(NonCancellable), как показано в следующем коде:

```
launch {
    try {
        suspendCall()
    } catch (e: CancellationException) {
```

```
        // обработка отмены
        withContext(NonCancellable) {
            anotherSuspendCall()
        }
    }
}
```

NonCancellable специально предназначен для функции withContext(), чтобы можно было убедиться, что предоставленный блок кода не будет отменен[8].

Причины отмены

Как уже было показано, есть два вида отмены: *спроектированная отмена* и *отмена в случае сбоя*. Изначально мы говорили, что при выбросе исключения возникает сбой. Это небольшое преувеличение. Вы только что видели, что при добровольной отмене сопрограммы выбрасывается исключение CancellationException. Собственно, в этом и состоит отличие двух этих видов.

При отмене Job.cancel (по замыслу) сопрограмма завершает работу, не затрагивая своего родителя. Если у родителя есть другие дочерние сопрограммы, эта отмена также не затрагивает их. Следующий код иллюстрирует это:

```
fun main() = runBlocking {
    val job = launch {
        val child1 = launch {
            delay(Long.MAX_VALUE)
        }
        val child2 = launch {
            child1.join()
            println("Child 1 is cancelled")

            delay(100)
            println("Child 2 is still alive!")
        }

        println("Cancelling child 1..")
        child1.cancel()
        child2.join()
        println("Parent is not cancelled")
    }
    job.join()
}
```

[8] На самом деле NonCancellable — это специальная реализация Job, которая всегда находится в состоянии Active. Поэтому функции, поддерживающие возможность приостановки, использующие функцию ensureActive() в данном контексте, никогда не отменяются.

Вывод:

```
Cancelling child 1..
Child 1 is cancelled
Child 2 is still alive!
Parent is not cancelled
```

Выполнение child1 откладывается навсегда, пока child2 ждет продолжения работы child1. Родительская сопрограмма быстро отменяет child1, и мы видим, что эта сопрограмма действительно отменяется, т. к. child2 продолжает выполнение. Наконец, фраза Parent is not cancelled является доказательством того, что отмена не повлияла на родительскую сопрограмму (как, кстати, и на child2).

С другой стороны, в случае сбоя (если было выброшено исключение, отличное от CancellationException) по умолчанию родительская сопрограмма отменяется. Если у нее есть другие дочерние сопрограммы, они также отменяются. Попробуем это проиллюстрировать, но предупреждаем — не делайте этого:

```
fun main() = runBlocking {
    val scope = CoroutineScope(coroutineContext + Job())     ❶

    val job = scope.launch {                                 ❷
        launch {
            try {
                delay(Long.MAX_VALUE)                        ❸
            } finally {
                println("Child 1 was cancelled")
            }
        }

        launch {
            delay(1000)                                      ❹
            throw IOException()
        }
    }
    job.join()                                               ❺
}
```

Здесь мы пытаемся создать обстоятельства, при которых дочерняя сопрограмма дает сбой через некоторое время, и мы хотим проверить, приведет ли это к сбою родительской сопрограммы. Затем нужно подтвердить, что все остальные дочерние сопрограммы также должны быть отменены, предполагая, что это переданная нами политика отмены.

На первый взгляд код выглядит нормально.

❶ Мы создаем родительскую область видимости.

❷ Запускаем в этой области новую сопрограмму.

❸ Первая дочерняя сопрограмма ждет бесконечно долго. Если она будет отменена, то должна появиться надпись "Child 1 was cancelled", т. к. из `delay(Long.MAX_VALUE)` было бы выброшено исключение `CancellationException`.

❹ Другая дочерняя сопрограмма выбрасывает исключение `IOException` после задержки продолжительностью в 1 секунду.

❺ Ждем, пока сопрограмма запустится на этапе ❷. Если этого не сделать, то выполнение `runBlocking` завершается, и программа останавливается.

Запустив эту программу, вы и правда увидите фразу "Child 1 was cancelled", хотя сразу после этого программа аварийно завершит работу с неперехваченным исключением `IOException`. Даже если окружить `job.join()` блоком `try/catch`, вы все равно получите сбой.

Чего здесь не хватает, так это возникновения исключения. Оно было выброшено из метода `launch`, который передает исключения вверх по иерархии сопрограмм, пока не достигнет родительской области видимости. Такое поведение нельзя переопределить. Как только область видимости увидит исключение, она отменит себя и все свои дочерние сопрограммы, а затем передаст исключение родительской сопрограмме, которая является областью видимости `runBlocking`.

Важно понимать, что попытка перехватить исключение не изменит того факта, что корневая сопрограмма `runBlocking` будет отменена этим исключением.

В некоторых случаях можно считать это приемлемым сценарием: любое необработанное исключение приводит к сбою программы. Однако в других ситуациях вы, возможно, предпочтете предотвратить сбой `scope` для передачи в основную сопрограмму. Для этого необходимо зарегистрировать `CoroutineExceptionHandler` (CEH):

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, exception ->
        println("Caught original $exception")
    }
    val scope = CoroutineScope(coroutineContext + ceh + Job())

    val job = scope.launch {
        // то же, что и в предыдущем примере
    }
}
```

На концептуальном уровне `CoroutineExceptionHandler` очень напоминает `Thread.UncaughtExceptionHandler`, за исключением того, что он предназначен для сопрограмм. Это элемент контекста, который следует добавить в контекст области видимости или сопрограмм. Область видимости должна создать собственный экземпляр `Job`, т. к. `CoroutineExceptionHandler` вступает в силу только при установке в верхней части иерархии сопрограмм. В предыдущем примере мы добавили CEH в контекст области видимости. Мы вполне могли бы добавить его в контекст первого метода `launch`:

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, exception ->
```

```
        println("Caught original $exception")
    }

    // CEH также может быть частью области видимости
    val scope = CoroutineScope(coroutineContext + Job())

    val job = scope.launch(ceh) {
        // то же, что и в предыдущем примере
    }
}
```

После выполнения этого примера с обработчиком исключений, вывод программы теперь выглядит так:

```
Child 1 was cancelled
Caught original java.io.IOException
```

Аварийного завершения программы больше нет. Из реализации CEH можно повторить предыдущие неудачные операции.

Данный пример демонстрирует, что по умолчанию сбой сопрограммы приводит к тому, что ее родительская сопрограмма отменяет себя и все остальные дочерние сопрограммы. А если такое поведение не соответствует архитектуре приложения? Иногда сбой сопрограммы допустим и не требует отмены всех других сопрограмм, запущенных в той же области видимости. Это называется *супервизией*.

Супервизия

Рассмотрим реальный пример загрузки макета фрагмента. Каждому дочернему представлению может потребоваться полная фоновая обработка. Предполагая, что мы используем область видимости, которая по умолчанию является основным потоком, и дочерние сопрограммы для фоновых задач, сбой одной из этих задач не должен привести к сбою родительской области видимости. В противном случае весь фрагмент перестал бы отвечать пользователю.

Чтобы реализовать данную стратегию отмены, можно использовать `SupervisorJob`, для которого сбой или отмена дочернего элемента не влияет на другие дочерние элементы; и это не влияет на саму область видимости. Обычно `SupervisorJob` используется в качестве замены `Job` при создании `CoroutineScope`. Полученную область называют *областью видимости супервизора*. Она передает отмену только дочерним сопрограммам, как показано в следующем коде:

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, e -> println("Handled $e") }
    val supervisor = SupervisorJob()
    val scope = CoroutineScope(coroutineContext + ceh + supervisor)
    with(scope) {
```

```
        val firstChild = launch {
            println("First child is failing")
            throw AssertionError("First child is cancelled")
        }

        val secondChild = launch {
            firstChild.join()

            delay(10) // экспериментируем с гипотетической отменой
            println("First child is cancelled: ${firstChild.isCancelled},
but second one is still active")
        }

        // ждем, пока вторая дочерняя сопрограмма не завершит работу
        secondChild.join()
    }
}
```

Вывод:

```
First child is failing
Handled java.lang.AssertionError: First child is cancelled
First child is cancelled: true, but second one is still active
```

Обратите внимание, что мы установили CEH в контексте области видимости. Почему? Первая дочерняя сопрограмма выбрасывает исключение, которое так и не перехватывается. Даже если сбой дочерней сопрограммы не влияет на область видимости супервизора, она все равно передает необработанные исключения, которые, как вы знаете, могут привести к сбою программы. Именно в этом и заключается цель CEH: обработка неперехваченных исключений. Интересно, что CEH также можно установить в контексте первой функции launch с тем же результатом, как показано ниже:

```
fun main() = runBlocking {
    val ceh = CoroutineExceptionHandler { _, e -> println("Handled $e") }
    val supervisor = SupervisorJob()
    val scope = CoroutineScope(coroutineContext + supervisor)
    with(scope) {
        val firstChild = launch(ceh) {
            println("First child is failing")
            throw AssertionError("First child is cancelled")
        }

        val secondChild = launch {
            firstChild.join()
```

```
            delay(10)
            println("First child is cancelled: ${firstChild.isCancelled},
but second one is still active")
        }

        // ждем, пока вторая дочерняя сопрограмма не завершит работу
        secondChild.join()
    }
}
```

CEH предназначен для установки на вершине иерархии сопрограмм, поскольку это то место, где можно обрабатывать неперехваченные исключения.

В этом примере CEH устанавливается в дочерней сопрограмме области видимости сопрограмм. Можно установить его во вложенную сопрограмму, как показано ниже:

```
val firstChild = launch {
    println("First child is failing")
    launch(ceh) {
       throw AssertionError("First child is cancelled")
    }
}
```

В этом случае CEH не учитывается, и программа может аварийно завершить работу.

Функция *supervisorScope*

Аналогично функции `coroutineScope`, которая наследует текущий контекст и создает новое задание, функция `supervisorScope` создает `SupervisorJob`. Как и `coroutineScope`, она ждет, пока все дочерние сопрограммы завершат работу. Одно важное отличие от `coroutineScope` заключается в том, что она передает отмену только дочерним сопрограммам и отменяет все дочерние сопрограммы, только если сама дает сбой. Еще одно отличие от функции `coroutineScope` заключается в способе обработки исключений. Мы подробно рассмотрим это в следующем разделе.

Параллельная декомпозиция

Представьте, что функция, поддерживающая возможность приостановки, должна выполнять несколько задач параллельно, прежде чем вернуть результат. Возьмем, к примеру, функцию `weatherForHike` из нашего приложения для пеших прогулок в начале этой главы. Получение данных о погоде может включать несколько API, в зависимости от характера данных. Данные о ветре и температуре можно получать отдельно, из независимых источников данных.

Предположим, что у нас есть функции fetchWind и fetchTemperatures. Тогда функцию weatherForHike можно реализовать следующим образом:

```
private suspend fun weatherForHike(hike: Hike): Weather =
          withContext(Dispatchers.IO)
{
   val deferredWind = async { fetchWind(hike) }
   val deferredTemp = async { fetchTemperatures(hike) }
   val wind = deferredWind.await()
   val temperatures = deferredTemp.await()
   Weather(wind, temperatures) // предполагая, что Weather можно создать таким образом
}
```

В этом примере также можно использовать функцию async, потому что функция withContext предоставляет CoroutineScope — свой последний аргумент, который является лямбда-выражением, поддерживающим возможность приостановки, с CoroutineScope в качестве приемника. Без withContext компиляции бы не было, потому что для функции async не было бы никакой области видимости.

Функция withContext особенно полезна, когда нужно изменить диспетчер в функции, поддерживающей возможность приостановки. Что делать, если его не нужно менять? Функцию weatherForHike вполне можно вызвать из сопрограммы, которая уже отправлена диспетчеру ввода-вывода. Тогда использование withContext(Dispatchers.IO) было бы лишним. В таких ситуациях вместо функции withContext можно использовать функцию coroutineScope или их сочетание. Это функция coroutineScope, и вот как она используется:

```
private suspend fun weatherForHike(hike: Hike): Weather = coroutineScope {
    // Выборка данных по ветру и температуре выполняется параллельно
    val deferredWind = async(Dispatchers.IO) {
        fetchWind(hike)
    }
    val deferredTemp = async(Dispatchers.IO) {
        fetchTemperatures(hike)
    }
   val wind = deferredWind.await()
   val temperatures = deferredTemp.await()
   Weather(wind, temperatures) // предполагая, что Weather можно создать таким образом
}
```

Здесь мы заменили функцию coroutineScope на withContext. Что делает coroutineScope? Прежде всего, взгляните на сигнатуру:

```
public suspend fun <R> coroutineScope(block: suspend CoroutineScope.() -> R): R
```

В официальной документации сказано, что эта функция создает CoroutineScope и вызывает указанный блок suspend с этой областью видимости. Предоставленная об-

ласть наследует coroutineContext из внешней области видимости, но переопределяет Job контекста.

Данная функция предназначена для *параллельной декомпозиции* работы. Когда какая-либо дочерняя сопрограмма в этой области видимости дает сбой, область видимости тоже дает сбой, и все остальные дочерние сопрограммы отменяются (если вам нужно другое поведение, используйте supervisorScope). Как только работа данного блока и всех дочерних сопрограмм будет завершена, функция возвратит управление.

Автоматическая отмена

Для нашего примера, если функция fetchWind дает сбой, то область видимости, предоставленная coroutineScope, также дает сбой, и функция fetchTemperatures впоследствии отменяется. Если fetchTemperatures включает в себя выделение памяти для крупных объектов, то такая отмена может оказаться выгодной.

Функция coroutineScope и в самом деле полезна, когда нужно *выполнить несколько задач параллельно.*

Обработка исключений

Обработка исключений — важная часть архитектуры приложения. Иногда вы просто перехватываете исключения сразу после их возникновения, а в других случаях позволяете им "всплывать", идя вверх по иерархии до тех пор, пока их не обработает выделенный компонент. В этом отношении конструкция try/catch — вероятно, тот способ, который вы использовали до сих пор. Однако у сопрограмм есть одна уловка. Можно было бы начать эту главу с ее описания, но сперва нужно было познакомить вас с *супервизией* и обработчиком CoroutineExceptionHandler.

Необработанные и открытые исключения

Когда дело доходит до распространения исключений, неперехваченные исключения могут обрабатываться механизмом сопрограмм одним из следующих способов.

Необработанные исключения.

Необработанные исключения могут обрабатываться только CoroutineExceptionHandler.

Открытые исключения.

Открытые исключения — это исключения, которые клиентский код может обработать с помощью конструкции try/catch.

Здесь можно выделить две категории функций запуска сопрограмм в зависимости от того, как они обрабатывают неперехваченные исключения:

- необработанные (одна из них — launch);
- открытые (одна из них — async).

Прежде всего, обратите внимание, что мы говорим о неперехваченных исключениях. Если перехватить исключение *до* того, как оно будет обработано функцией запуска сопрограмм, все работает как обычно — вы перехватываете его, поэтому механизм сопрограммы не знает о нем. Ниже показан пример с функцией launch и конструкцией try/catch:

```
scope.launch {
    try {
        regularFunctionWhichCanThrowException()
    } catch (e: Exception) {
        // обработка исключения
    }
}
```

Этот пример работает, как и ожидалось, *если*, как следует из названия, RegularFunctionWhichCanThrowException — это обычная функция, которая напрямую или косвенно не задействует другие функции запуска сопрограмм, в этом случае могут применяться специальные правила (как будет показано далее).

Та же идея применима к функции async, как показано ниже:

```
fun main() = runBlocking {

    val itemCntDeferred = async{
        try {
            getItemCount()
        } catch (e: Exception) {
            // Что-то пошло не так. Предположим, вам все равно, и вы считаете,
            // что должен вернуться 0.
            0
        }
    }

    val count = itemCntDeferred.await()
    println("Item count: $count")
}

fun getItemCount(): Int {
    throw Exception()
    1
}
```

Вывод этой программы, как несложно догадаться, выглядит так:

```
Item count: 0
```

В качестве альтернативы вместо конструкции try/catch можно воспользоваться функцией runCatching. Это позволяет использовать более красивый синтаксис, если вы считаете, что при оптимистичном сценарии исключение не выбрасывается:

```
scope.launch {
    val result = runCatching {
        regularFunctionWhichCanThrowException()
    }

    if (result.isSuccess) {
        // никаких исключений не было
    } else {
        // было выброшено исключение
    }
}
```

"Под капотом" runCatching — не что иное как конструкция try/catch, возвращающая объект Result, который предлагает методы синтаксического сахара, такие как getOrNull() и exceptionOrNull(), как показано далее:

```
/**
 * Вызывает указанную функцию [block] со значением this в качестве приемника
 * и возвращает инкапсулированный результат, если вызов был успешным, перехватывая
 * и инкапсулируя любое выброшенное исключение в качестве сбоя.
 */
public inline fun <T, R> T.runCatching(block: T.() -> R): Result<R> {
    return try {
        Result.success(block())
    } catch (e: Throwable) {
        Result.failure(e)
    }
}
```

Некоторые функции-расширения определены в Result и доступны "из коробки", например getOrDefault, которая возвращает инкапсулированное значение экземпляра Result, если Result.isSuccess имеет значение true, или в противном случае предоставленное значение по умолчанию.

Открытые исключения

Как уже говорилось ранее, можно перехватывать *открытые* исключения, используя встроенную языковую конструкцию try/catch. В следующем коде показано, что

мы создали собственную область видимости, в которой две параллельные задачи, task1 и task2, запускаются в supervisorScope. task2 сразу же выдает ошибку:

```
fun main() = runBlocking {

    val scope = CoroutineScope(Job())

    val job = scope.launch {
        supervisorScope {
            val task1 = launch {
                // имитируем фоновую задачу
                delay(1000)
                println("Done background task")
            }

            val task2 = async {
                // пытаемся получить какое-то количество, но неудачно
                throw Exception()
                1
            }

            try {
                task2.await()
            } catch (e: Exception) {
                println("Caught exception $e")
            }
            task1.join()
        }
    }

    job.join()
    println("Program ends")
}
```

Вывод этой программы:

```
Caught exception java.lang.Exception
Done background task
Program ends
```

Данный пример демонстрирует, что в supervisorScope функция async предоставляет доступ к неперехваченным исключениям в вызове await(). Если не окружить вызов await() блоком try/catch, то область supervisorScope дает сбой и отменяет task1, а затем предоставляет доступ к родителю, вызвавшему сбой. Таким образом, это

означает, что даже при использовании supervisorScope необработанные исключения в области видимости приводят к отмене всей иерархии сопрограмм ниже в этой области — и исключение передается вверх. При обработке исключения так, как мы делали в этом примере, task2 завершается с ошибкой, а с task1 все в порядке.

Интересно, что, если не вызвать task2.await(), то программа выполняется так, как если бы никаких исключений никогда и не было — thrown `task2` молча выдает ошибку.

Теперь повторим тот же самый пример, но вместо supervisorScope воспользуемся coroutineScope:

```
fun main() = runBlocking {

    val scope = CoroutineScope(Job())
    val job = scope.launch {
        coroutineScope{
            val task1 = launch {
                delay(1000)
                println("Done background task")
            }

            val task2 = async {
                throw Exception()
                1
            }

            try {
                task2.await()
            } catch (e: Exception) {
                println("Caught exception $e")
            }
            task1.join()
        }
    }

    job.join()
    println("Program ends")
}
```

Вывод:

```
Caught exception java.lang.Exception
```

Программа аварийно завершает работу из-за java.lang.Exception — скоро мы объясним это.

Исходя из этого можно узнать, что в coroutineScope функция async предоставляет доступ к неперехваченным исключениям, но также уведомляет и родителя. Если вы не вызовете task2.await(), то программа все равно аварийно завершит работу из-за сбоя coroutineScope и *предоставит* родителю исключение, вызвавшее сбой. Тогда scope.launch будет рассматривать это исключение как *необработанное*.

Необработанные исключения

Сопрограммы работают с необработанными исключениями особым образом: они пытаются использовать CEH, если он есть в контексте сопрограмм. В противном случае они делегируют работу *глобальному обработчику*. Этот обработчик вызывает настраиваемый набор CEH *и* вызывает стандартный механизм необработанных исключений: Thread.uncaughtExceptionHandler. По умолчанию в Android ранее упомянутый набор обработчиков состоит только из одного CEH, который выводит трассировку стека необработанного исключения. Однако можно зарегистрировать собственный обработчик, который будет вызываться помимо обработчика, выводящего трассировку стека. Поэтому вы должны помнить, что если не обработаете исключение, то будет вызван обработчик Thread.uncaughtExceptionHandler.

В Android по умолчанию обработчик UncaughtExceptionHandler приводит к сбою приложения, а в JVM[9] обработчик по умолчанию выводит трассировку стека в консоль. Следовательно, если запустить эту программу не в Android, а в JVM, то результат будет следующим[10]:

```
Caught exception java.lang.Exception
(stacktrace of java.lang.Exception)
Program ends
```

Вернемся к Android. Как бы вы обработали это исключение? Поскольку coroutineScope предоставляет исключения, можно поместить coroutineScope внутрь оператора try/catch. Или, если не обработать его правильно, то scope.launch будет воспринимать это исключение как необработанное. Тогда ваш последний шанс — зарегистрировать CEH. Есть как минимум две причины, по которым можно было бы это сделать: во-первых, чтобы остановить передачу исключения и избежать сбоя программы, а, во-вторых, чтобы уведомить аналитику сбоев и повторно выбросить исключение, что может привести к сбою приложения. В любом случае мы не выступаем за тихий перехват исключений. Если вы хотите использовать CEH, следует знать несколько вещей. CEH работает только при регистрации в:

- launch (но не async), когда launch — это функция запуска корневых сопрограмм[11];
- области видимости;
- прямом потомке supervisorScope.

---

[9] Говоря JVM, мы подразумеваем настольное приложение или серверную часть.

[10] Выводится надпись Program ends, потому что необработанное исключение приводит к сбою scope, а не области видимости из runBlocking.

[11] Функция запуска корневых сопрограмм — это прямой потомок области видимости. В предыдущем примере, в строке val job = scope.launch {..}, launch — это функция запуска корневых сопрограмм.

В нашем примере CEH должен быть зарегистрирован либо в scope.launch, либо в самой области видимости. В следующем коде показано, что он зарегистрирован в корневой сопрограмме:

```
fun main() = runBlocking {

    val ceh = CoroutineExceptionHandler { _, t ->
        println("CEH handle $t")
    }

    val scope = CoroutineScope(Job())

    val job = scope.launch(ceh) {
        coroutineScope {
            val task1 = launch {
                delay(1000)
                println("Done background task")
            }

            val task2 = async {
                throw Exception()
                1
            }

            task1.join()
        }
    }

    job.join()
    println("Program ends")
}
```

Вывод:

```
Caught exception java.lang.Exception
CEH handle java.lang.Exception
Program ends
```

Тот же пример. На этот раз CEH зарегистрирован в области видимости:

```
fun main() = runBlocking {

    val ceh = CoroutineExceptionHandler { _, t ->
        println("CEH handle $t")
    }
```

```
    val scope = CoroutineScope(Job() + ceh)

    val job = scope.launch {
       // как в предыдущем примере
    }
}
```

Наконец, проиллюстрируем использование CEH в прямом потомке supervisorScope:

```
fun main() = runBlocking {

    val ceh = CoroutineExceptionHandler { _, t ->
        println("CEH handle $t")
    }

    val scope = CoroutineScope(Job())

    val job = scope.launch {
        supervisorScope {
            val task1 = launch {
                // имитируем фоновую задачу
                delay(1000)
                println("Done background task")
            }

            val task2 = launch(ceh) {
                // пытаемся получить некоторое количество, но неудачно
                throw Exception()
            }

            task1.join()
            task2.join()
        }
    }

    job.join()
    println("Program ends")
}
```

Обратите внимание, что функция запуска сопрограмм, в которой зарегистрирован CEH, — это launch. В случае с функцией async, выявляющей неперехваченные исключения, которые можно обработать с помощью конструкции try/catch, мы бы не приняли это во внимание.

Резюме

- Если функция не может вернуть управление сразу, ее можно реализовать как функцию, поддерживающую возможность приостановки. Однако модификатор `suspend` не превращает волшебным образом блокирующий вызов в неблокирующий. Используйте функцию `withContext` вместе с соответствующим диспетчером и/или вызовите другие функции, поддерживающие возможность приостановки.
- Сопрограмму можно намеренно отменить с помощью `Job.cancel()` для функции `launch` или `Deferred.cancel()` для функции `async`. Если вам нужно вызвать функции, поддерживающие возможность приостановки, в коде очистки, убедитесь, что вы обернули логику очистки в блок `withContext(NonCancellable) { .. }`. Отмененная сопрограмма останется в состоянии `Cancelling` до завершения очистки. После завершения очистки она переходит в состояние `Cancelled`.
- Сопрограмма всегда ожидает завершения дочерних сопрограмм перед тем, как завершить выполнение. Таким образом, отмена сопрограммы также отменяет все дочерние сопрограммы.
- Сопрограммы должны работать согласованно, чтобы их можно было отменить. Все функции, допускающие приостановку, из пакета *kotlinx.coroutines* можно отменять. Сюда, в частности, входит и функция `withContext`. Если вы реализуете собственную функцию, допускающую приостановку, убедитесь, что ее можно отменить, выполнив проверку с помощью `isActive` или вызвав функции `sureActive()` или `yield()` на соответствующих этапах.
- Существует две категории областей видимости сопрограмм: области, использующие `Job`, и области, использующие `SupervisorJob` (которые также называются областями видимости супервизора). Они различаются способом выполнения отмены и обработки исключений. Если в результате ошибки дочерней сопрограммы другие дочерние сопрограммы также должны быть отменены, используйте обычную область видимости. В противном случае используйте область видимости супервизора.
- Функции `launch` и `async` различаются способом обработки неперехваченных исключений. Функция `async` предоставляет исключения, которые можно перехватить, обернув вызов `await()` в `try/catch`. С другой стороны, функция `launch` обрабатывает неперехваченные исключения как необработанные, которые можно обработать с помощью CEH.
- CEH не обязателен. Его следует использовать, только когда вам действительно нужно что-то сделать с необработанными исключениями. Необработанные исключения обычно должны привести к сбою приложения. Или, по крайней мере, восстановление после некоторых исключений может оставить приложение в неопределенном состоянии. Тем не менее, если вы решите использовать CEH, его нужно установить на вершине иерархии сопрограмм — обычно в самой верхней области. Это также можно сделать в дочернем элементе `supervisorScope`.

- Если сопрограмма выдает ошибку из-за неперехваченного исключения, она отменяется вместе со всеми дочерними сопрограммами, и исключения передаются родительской сопрограмме.

Размышления напоследок

Вы узнали, как писать собственные функции, поддерживающие возможность приостановки, и как использовать их в сопрограммах. Области видимости — это место обитания сопрограмм. Вы знаете, как сделать выбор между `coroutineScope` и `supervisorScope`, чтобы реализовать желаемую политику отмены. Создаваемые вами области видимости — это дочерние элементы других областей, расположенных выше в иерархии. В Android эти "корневые" области видимости предоставляются библиотекой — их не создают самостоятельно. Хороший пример — `viewModelScope`, доступный в любом экземпляре `ViewModel`.

Сопрограммы идеально подходят для одноразовых или повторяющихся задач. Однако нам часто приходится работать с асинхронными потоками данных. Для этого есть каналы и потоки, о которых мы поговорим в следующих двух главах.

ГЛАВА 9
Каналы

Из предыдущей главы вы узнали, как создавать сопрограммы, отменять их и обрабатывать исключения. Итак, вы знаете, что, если задаче *B* требуется результат задачи *A*, можно реализовать их в виде двух функций, поддерживающих возможность приостановки, которые вызываются последовательно. А если задача *A* производит поток значений? Функция `async` и функции, допускающие приостановку, не подходят для этого. Здесь и вступает в игру примитивный тип `Channel`[1] — он заставляет сопрограммы общаться. В этой главе вы подробно узнаете, что такое каналы и как их использовать.

Используя только каналы и сопрограммы, можно проектировать сложную асинхронную логику с помощью *взаимодействующих последовательных процессов* (communicating sequential processes, CSP). Что такое CSP? Создатели Kotlin вдохновлялись несколькими уже существующими языками программирования, такими как Java, C#, JavaScript, Scala и Groovy. Примечательно, что источником вдохновения сопрограмм в Kotlin послужил язык Go и его сопрограммы (*goroutines*).

В компьютерных науках CSP — это язык параллельного программирования, который впервые был описан Тони Хоаром в 1978 г. С тех пор он развивался, и теперь термин CSP в основном используется для описания стиля программирования. Если вы знакомы с моделью актора, то CSP очень похож на нее, хотя и есть некоторые отличия. Если вы никогда не слышали о CSP, не волнуйтесь — мы кратко объясним его суть и приведем практические примеры. На данный момент можно рассматривать CSP как стиль программирования.

Как обычно, мы начнем с теории, а затем реализуем задачу из жизни. В заключение мы обсудим преимущества и недостатки CSP, используя сопрограммы.

Обзор каналов

Вернемся к нашему примеру. Представьте, что одна задача асинхронно создает список из трех экземпляров `Item` (производитель), а другая задача что-то делает с каждым из них (потребитель). Поскольку производитель не сразу возвращает

[1] Иногда в оставшейся части этой главы мы будем называть его просто каналом.

управление, можно реализовать его как функцию getItems, которая поддерживает возможность приостановки:

```
suspend fun getItems(): List<Item> {
    val items = mutableListOf<Item>()
    items.add(makeItem())
    items.add(makeItem())
    items.add(makeItem())
    return items
}

suspend fun makeItem(): Item {
    delay(10) // имитация асинхронности
    return Item()
}
```

Что касается потребителя, который обрабатывает каждый из этих экземпляров, то можно использовать следующую реализацию:

```
fun consumeItems(items: List<Item>) {
    for (item in items) println("Do something with $item")
}
```

Собираем все воедино:

```
fun main() = runBlocking {
    val items = getItems()
    consumeItems(items)
}
```

Как и следовало ожидать, фраза "Do something with..." выводится трижды. Однако в данном случае нас больше всего интересует порядок выполнения. Посмотрим поближе, что же происходит на самом деле.

На рис. 9.1 показано, что потребление начинается только после того, как были произведены все экземпляры. На это может уйти довольно много времени, и в некоторых ситуациях такое ожидание неприемлемо. Взамен можно было бы передавать каждый асинхронно созданный экземпляр, как показано на рис. 9.2.

Для этого нельзя реализовать getItems как функцию, поддерживающую возможность приостановки, как раньше. Сопрограмма должна работать как производитель экземпляров Item и отправлять их в основную сопрограмму. Это типичная проблема производителя и потребителя.

В *главе 5* мы объяснили, как использовать BlockingQueue для реализации *очередей работ* или, в данном случае, очереди данных. Напоминаем, что у BlockingQueue есть блокирующие методы put и take, которые служат для того, чтобы помещать объект в очередь и забирать его оттуда. Когда очередь используется как единственное средство взаимодействия между двумя потоками (производителем и потребителем), это дает существенное преимущество, позволяя избежать совместного использования изменяемого состояния. Кроме того, если очередь ограничена (имеет ограниче-

ния по размеру), слишком быстрый производитель в конечном счете блокируется в вызове метода put, если потребители слишком медленные. Это явление известно как противодавление: заблокированный производитель дает потребителям возможность наверстать упущенное, таким образом освобождая производителя.

Рис. 9.1. Обрабатываем все сразу

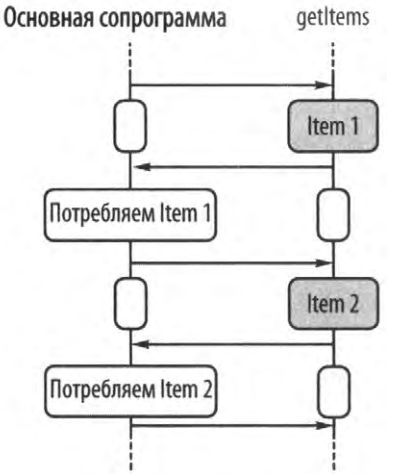


Рис. 9.2. Последовательная обработка

Использование BlockingQueue в качестве примитивного типа, поддерживающего возможность взаимодействия сопрограмм, не очень хорошая идея, поскольку сопрограмма не должна включать в себя блокировку вызовов. Вместо этого выполнение сопрограмм можно приостановить. Именно так и можно воспринимать канал — как очередь с функциями send и receive (рис. 9.3). У него также имеются аналоги, которые не являются функциями, поддерживающими возможность приостановки: trySend и tryReceive. Эти два метода неблокирующие. trySend пытается сразу добавить элемент в канал и возвращает класс-обертку вокруг результата. Этот класс-обертка, ChannelResult<T>, также указывает на успешную или неудачную операцию. tryReceive пытается немедленно получить элемент из канала и возвращает экземпляр ChannelResult<T>.

Рис. 9.3. Канал

Как и очереди, каналы бывают нескольких видов. Мы рассмотрим каждый из этих вариантов и приведем базовые примеры.

Рандеву-канал

Rendez-vous — французское слово, которое означает "встреча" или "свидание" — все зависит от контекста (здесь мы не имеем в виду CoroutineContext). У рандеву-

канала вообще нет буфера. Элемент передается от отправителя к получателю только тогда, когда вызовы send и receive встречаются (рандеву), поэтому выполнение функции send приостанавливается до тех пор, пока другая сопрограмма вызывает функцию receive, и выполнение этой функции приостанавливается до тех пор, пока другая сопрограмма не вызовет функцию send.

Другими словами, рандеву-канал реализует двустороннее взаимодействие между производителями (сопрограммами, вызывающими функцию send) и потребителями (сопрограммами, вызывающими функцию receive). Двух последовательных вызовов функции send без receive посередине быть не может.

По умолчанию при создании канала с помощью Channel<T>() вы получаете рандеву-канал.

Его можно использовать для правильной реализации предыдущего примера:

```
fun main() = runBlocking {
    val channel = Channel<Item>()
    launch {                              ❶
        channel.send(Item(1))             ❸
        channel.send(Item(2))             ❹
        println("Done sending")
    }

    println(channel.receive())            ❷
    println(channel.receive())            ❺

    println("Done!")
}

data class Item(val number: Int)
```

Вывод программы:

```
Item(number=1)
Item(number=2)
Done!
Done sending
```

В этом примере основная сопрограмма запускает дочернюю сопрограмму с помощью функции launch ❶, затем доходит до строки ❷ и приостанавливает выполнение до тех пор, пока какая-нибудь сопрограмма не отправит экземпляр Item в канал. Вскоре после этого дочерняя сопрограмма отправляет первый элемент ❸, затем доходит до соответствующей строки и приостанавливается на втором вызове функции send ❹, пока какая-нибудь сопрограмма не будет готова принять элемент. Впоследствии основная сопрограмма (которая приостановила выполнение ❷) возобновляет работу, получает первый элемент из канала и выводит его. Затем основная сопрограмма доходит до строки ❺ и немедленно получает второй элемент, т. к. до-

черняя сопрограмма уже была приостановлена в вызове send. Сразу после этого дочерняя сопрограмма продолжает выполнение (выводит фразу "Done sending").

Перебор канала

Канал можно перебирать, используя обычный цикл for. Обратите внимание, что поскольку каналы не являются обычными коллекциями[2], цикл forEach или другие подобные функции из стандартной библиотеки Kotlin использовать нельзя.

Здесь перебор канала — это особая функциональность на уровне языка, которую можно выполнить только с использованием синтаксиса цикла for:

```
for (x in channel) {
    // делаем что-то с x каждый раз,
    // когда какая-то сопрограмма отправляет элемент в канал
}
```

При каждой итерации x неявно равен channel.receive(). Следовательно, сопрограмма, выполняющая перебор канала, может делать это бесконечно, если только она не содержит условную логику для разрыва цикла. К счастью, существует стандартный механизм разрыва цикла — закрытие канала. Например:

```
fun main() = runBlocking {
    val channel = Channel<Item>()
    launch {
        channel.send(Item(1))
        channel.send(Item(2))
        println("Done sending")
        channel.close()
    }

    for (x in channel) {
        println(x)
    }
    println("Done!")
}
```

У этой программы аналогичный вывод с небольшим отличием:

```
Item(number=1)
Item(number=2)
Done sending
Done!
```

На этот раз фраза "Done sending" появляется перед сообщением "Done!" потому, что основная сопрограмма прекращает перебор канала, только когда канал закрыт. А это происходит после того, как дочерняя сопрограмма отправила все элементы.

---

[2] В частности, Channel не реализует интерфейс Iterable.

При закрытии канала туда отправляется специальный токен, указывающий, что другие элементы отправляться не будут. Поскольку элементы в канале потребляются последовательно (один за другим), все элементы, отправленные в рандеву-канал до специального токена, гарантированно будут отправлены получателю.

Будьте осторожны — попытка вызова функции `receive` из уже закрытого канала вызовет исключение `ClosedReceiveChannelException`. Однако попытка выполнить перебор такого канала никаких исключений не вызывает:

```
fun main() = runBlocking {
    val channel = Channel<Int>()
    channel.close()

    for (x in channel) {
        println(x)
    }
    println("Done!")
}
```

Вывод:

```
Done!
```

Другие виды каналов

В предыдущем примере канал создается с помощью конструктора класса. Если посмотреть на исходный код, то можно увидеть, что на самом деле это общедоступная функция, имя которой начинается с прописной буквы C, чтобы создать иллюзию использования конструктора класса:

```
public fun <E> Channel(capacity: Int = RENDEZVOUS): Channel<E> =
    when (capacity) {
        RENDEZVOUS -> RendezvousChannel()
        UNLIMITED -> LinkedListChannel()
        CONFLATED -> ConflatedChannel()
        BUFFERED -> ArrayChannel(CHANNEL_DEFAULT_CAPACITY)
        else -> ArrayChannel(capacity)
}
```

Видно, что у этой функции есть параметр `capacity`, значение которого по умолчанию — `RENDEZVOUS`. Для справки: если войти в объявление `RENDEZVOUS`, то можно увидеть, что оно равно 0. Для каждого значения `capacity` есть соответствующая реализация канала. Существуют четыре различных вида каналов: рандеву, *неограниченный*, *объединенный* и *буферизованный*. Не обращайте особого внимания на конкретные реализации (например, `RendezvousChannel()`), потому что это внутренние

классы и в будущем они могут измениться. С другой стороны, значения RENDEZVOUS, UNLIMITED, CONFLATED и BUFFERED являются частью общедоступного API.

Мы рассмотрим каждый из этих видов каналов в последующих разделах.

Неограниченный канал

У *неограниченного канала* есть буфер, который лимитирован только количеством доступной памяти. Отправители элементов в этот канал никогда не приостанавливаются, а получатели приостанавливаются, лишь когда канал пуст. Сопрограммам, обменивающимся данными по *неограниченному каналу*, не нужно встречаться.

В этот момент вы можете подумать, что у такого канала должны быть проблемы с параллельной модификацией, когда отправители и получатели выполняются из разных потоков. В конце концов, сопрограммы отправляются в потоки, поэтому канал вполне можно использовать из разных потоков. Давайте сами проверим его надежность! В следующем примере мы отсылаем значения типа Int из сопрограммы, отправленной в Dispatchers.Default, в то время как другая сопрограмма считывает значение из того же канала из основного потока, и если каналы не являются потокобезопасными, мы заметим следующее:

```
fun main() = runBlocking {
    val channel = Channel<Int>(UNLIMITED)
    val childJob = launch(Dispatchers.Default) {
        println("Child executing from ${Thread.currentThread().name}")
        var i = 0
        while (isActive) {
            channel.send(i++)
        }
        println("Child is done sending")
    }

    println("Parent executing from ${Thread.currentThread().name}")
    for (x in channel) {
        println(x)

        if (x == 1000_000) {
            childJob.cancel()
            break
        }
    }

    println("Done!")
}
```

Вывод этой программы:

```
Parent executing from main
Child executing from DefaultDispatcher-worker-2
0
1
..
1000000
Done!
Child is done sending
```

Можно запускать этот пример сколько угодно раз, и он всегда завершается без проблем параллелизма. Это связано с тем, что "под капотом" канал использует алгоритм без блокировки[3].

Каналы потокобезопасны. Несколько потоков могут одновременно вызывать методы send и receive потокобезопасным способом.

Объединенный канал

У этого канала есть буфер размером 1, и он сохраняет только последний отправленный элемент. Чтобы создать *объединенный канал*, вызовите `Channel<T>(Channel.CONFLATED)`. Например:

```
fun main() = runBlocking {
    val channel = Channel<String>(Channel.CONFLATED)

    val job = launch {
        channel.send("one")
        channel.send("two")
    }

    job.join()
    val elem = channel.receive()
    println("Last value was: $elem")
}
```

Вывод этой программы:

```
Last value was: two
```

[3] Если хотите узнать, как работает такой алгоритм, рекомендуем прочитать раздел 15.4 "Неблокирующие алгоритмы" из книги "Java Concurrency на практике". Кроме того, на сайте YouTube есть интересное видео "Lock-Free Algorithms for Kotlin Coroutines (Part 1)" (**https://oreil.ly/WDE1F**) от Романа Елизарова, ведущего разработчика сопрограмм Kotlin.

Первый отправленный элемент — "one". Когда отправляется "two", он заменяет "one" в канале. Ждем завершения отправки элементов сопрограммы, используя `job.join()`, а затем считываем значение `two` из канала.

Буферизованный канал

Буферизованный канал — это канал с фиксированной вместимостью — целым числом больше 0. Отправители здесь не приостанавливаются, если буфер не заполнен, а получатели не приостанавливаются, если буфер не пустой. Чтобы создать буферизованный канал `Int` с буфером размера 2, нужно вызвать `Channel<Int>(2)`. Вот пример использования:

```
fun main() = runBlocking<Unit> {
    val channel = Channel<Int>(2)

    launch {
        for (i in 0..4) {
            println("Send $i")
            channel.send(i)
        }
    }

    launch {
        for (i in channel) {
            println("Received $i")
        }
    }
}
```

Вывод этой программы:

```
Send 0
Send 1
Send 2
Received 0
Received 1
Received 2
Send 3
Send 4
Received 3
Received 4
```

В этом примере мы определили канал с фиксированной вместимостью, равной 2. Сопрограмма пытается отправить пять целых чисел, в то время как другая сопрограмма потребляет элементы из канала. Сопрограмма-отправитель умудряется от-

править 0 и 1 за один раз, а затем пытается отправить 3. println("Send $i") выполняется для значения 3, но при вызове метода send выполнение отправителя приостанавливается. То же самое относится и к сопрограмме-потребителю: два элемента получаются последовательно с дополнительным выводом перед приостановкой выполнения.

Функция *produce*

До сих пор вы видели, что канал можно использовать как для отправки, так *и* для получения элементов. Иногда у вас может возникнуть желание точнее указать, как следует использовать канал. При реализации канала, предназначенного только для чтения другими сопрограммами, можно применять функцию produce:

```
fun CoroutineScope.produceValues(): ReceiveChannel<String> = produce {
    send("one")
    send("two")
}
```

Как видно, функция produce возвращает экземпляр ReceiveChannel, у которого имеются только методы, относящиеся к получению (метод receive из их числа). Экземпляр ReceiveChannel нельзя использовать для отправки элементов.

Кроме того, мы определили productValues() как функцию-расширение CoroutineScope. При вызове этой функции запускается новая сопрограмма, отправляющая элементы в канал. В Kotlin есть соглашение: каждая функция, запускающая сопрограммы, должна быть определена как функция-расширение CoroutineScope. Если следовать этому соглашению, то можно легко увидеть в коде, какие функции запускают новые сопрограммы из функций, поддерживающих возможность приостановки.

Основной код, в котором используется функция productValues, может выглядеть так:

```
fun main() = runBlocking {
    val receiveChannel = produceValues()

    for (e in receiveChannel) {
        println(e)
    }
}
```

И наоборот, у SendChannel есть только методы, относящиеся к отправке. На самом деле, если посмотреть на исходный код, видно, что Channel — это интерфейс, производный от ReceiveChannel и SendChannel:

```
public interface Channel<E> : SendChannel<E>, ReceiveChannel<E> {
    // код удален для краткости
}
```

Вот как можно использовать SendChannel:

```
fun CoroutineScope.collectImages(imagesOutput: SendChannel<Image>) {
    launch(Dispatchers.IO) {
        val image = readImage()
        imagesOutput.send(image)
    }
}
```

Взаимодействующие последовательные процессы

Достаточно теории. Посмотрим, как использовать каналы для решения реальной проблемы. Представьте, что ваше приложение должно отображать "фигуры" на холсте. В зависимости от данных, вводимых пользователем, приложение должно отображать произвольное количество фигур. Мы намеренно используем общие термины — фигура может быть интересующим вас местом на карте, элементом игры, всем, что может потребовать фоновой работы, например вызовы API, чтение файлов, запросы к базе данных и т. д. В нашем примере основной поток, который уже обрабатывает пользовательский ввод, будет имитировать запросы на новые фигуры, которые нужно отобразить. Возможно, вы уже поняли, что это проблема "производитель — потребитель": основной поток выполняет запросы, а какая-то фоновая задача их обрабатывает и возвращает результаты основному потоку.

Наша реализация должна:

- быть потокобезопасной;
- уменьшить риск переполнения памяти устройства;
- не иметь конфликтов между потоками (мы не будем использовать мьютексы).

Модель и архитектура

Класс данных Shape состоит из Location и ShapeData:

```
data class Shape(val location: Location, val data: ShapeData)
data class Location(val x: Int, val y: Int)
class ShapeData
```

Учитывая местоположение (Location), нам нужно получить соответствующие данные фигуры (ShapeData) для ее построения. Итак, в этом примере Location — это входные данные, а Shape — выходные. Для краткости будем использовать слово "местоположение" для обозначения Location и слово "фигура" для обозначения Shape.

В нашей реализации мы выделим два основных компонента.

Компонент ViewModel.

Содержит бóльшую часть логики приложения, связанной с фигурами. По мере взаимодействия пользователя с пользовательским интерфейсом представление предоставляет компоненту ViewModel список местоположений.

Компонент shapeCollector

Отвечает за получение фигур по списку местоположений.

На рис. 9.4 показана двунаправленная связь между компонентом ViewModel и ShapeCollector.

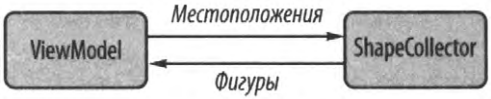


Рис. 9.4. Высокоуровневая архитектура

ShapeCollector следует простому процессу:

```
                fetchData
Location ---------------------> ShapeData
```

В качестве дополнительной предпосылки ShapeCollector должен вести внутренний "реестр" обрабатываемых местоположений. Получив местоположение для обработки, он не должен пытаться скачать его, если оно уже обрабатывается.

Первая реализация

Можно начать с первой простейшей реализации ShapeCollector, которая далека от завершения, но вы сможете уловить суть:

```
class ShapeCollector {
    private val locationsBeingProcessed = mutableListOf<Location>()

    fun processLocation(location: Location) {
        if (locationsBeingProcessed.add(location)) {
            // Получаем данные, а затем отправляем экземпляр Shape
            // обратно в компонент ViewModel.
        }
    }
}
```

Если бы мы применяли потоки, у нас было бы несколько потоков, которые совместно используют экземпляр ShapeCollector и параллельно выполняют processLocation. Однако такой подход приводит к совместному использованию изменяемых состояний. В предыдущем фрагменте кода locationBeingProcessed — один из примеров.

Как вы узнали из *главы 5*, совершать ошибки при использовании блокировок на удивление легко. Работая с сопрограммами, нам не нужно совместно использовать изменяемое состояние. Как это? Посредством сопрограмм и каналов можно *прибегать к совместному использованию, взаимодействуя* вместо того, чтобы *взаимодействовать, прибегая к совместному использованию.*

Основная идея заключается в том, чтобы инкапсулировать изменяемые состояния в сопрограммах. В случае обрабатываемого списка местоположений это можно сделать следующим образом:

```
launch {
    val locationsBeingProcessed = mutableListOf<Location>() ❶

    for (location in locations) {                          ❷
        // тот же код
    }
}
```

❶ В предыдущем примере только сопрограмма, запускаемая методом launch, может коснуться изменяемого состояния — LocationBeingProcessed.

❷ Однако теперь у нас проблема. Как предоставить местоположения? Нужно каким-то образом предоставить этот Iterable сопрограмме. Поэтому мы обратимся к каналу и будем использовать его в качестве входных данных функции, которую мы объявим. Поскольку мы запускаем сопрограмму в функции, то объявляем ее как функцию-расширение CoroutineScope:

```
private fun CoroutineScope.collectShapes(
    locations: ReceiveChannel<Location>
) = launch {
    // код удален для краткости
}
```

Так как эта сопрограмма будет получать местоположения из компонента ViewModel, мы объявляем канал как ReceiveChannel. Кстати, в предыдущем разделе вы видели, что канал можно перебирать, как и список. Итак, теперь мы можем получить соответствующие данные фигуры (ShapeData) для каждого экземпляра Location, полученного из канала. Поскольку нам понадобится сделать это параллельно, у нас может возникнуть соблазн написать что-то вроде этого:

```
private fun CoroutineScope.collectShapes(
    locations: ReceiveChannel<Location>
) = launch {
    val locationsBeingProcessed = mutableListOf<Location>()

    for (loc in locations) {
        if (!locationsBeingProcessed.contains(loc)) {
            launch(Dispatchers.IO) {
                // получаем соответствующий ShapeData
            }
        }
    }
}
```

Будьте осторожны: в этом коде есть ловушка. Дело в том, что для каждого полученного местоположения мы запускаем новую сопрограмму. Потенциально этот код может запускать много сопрограмм, если канал `locations` "списывает" большое количество элементов. По этой причине такую ситуацию также называют *неограниченным параллелизмом*. Когда мы познакомили вас с сопрограммами, то сказали, что они легковесны. Это правда, но работа, которую они выполняют, вполне может потреблять значительные ресурсы. В данном случае `launch(Dispatchers.IO)` сам по себе несет незначительные затраты, а для получения `ShapeData` может потребоваться вызов REST API на сервере с ограниченной пропускной способностью.

Поэтому придется найти способ ограничить параллелизм — нам не нужно запускать неограниченное количество сопрограмм. При столкновении с такой ситуацией с потоками обычно используют пул потоков с очередью работ (*см. главу 5*). Вместо этого мы создадим *пул*, который назовем *пулом рабочих сопрограмм*. Каждая сопрограмма из этого пула будет выполнять фактическую выборку `ShapeData` для конкретного местоположения. Для взаимодействия с этим пулом `collectShapes` должна использовать дополнительный канал, по которому она может отправлять местоположения в пул (рис. 9.5).

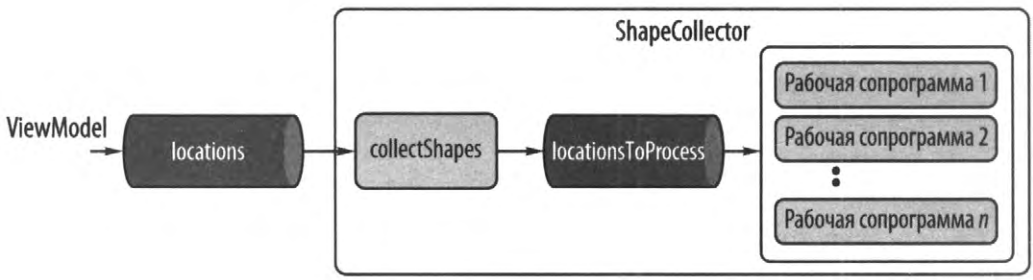


Рис. 9.5. Ограничение параллелизма

Используя каналы, *следите за отсутствием неограниченного параллелизма*. Представьте, что вам нужно создать большое количество экземпляров `Bitmap`. Базовый буфер памяти, в котором хранятся пиксельные данные, занимает довольно много места. При работе с большим количеством изображений выделение памяти для нового экземпляра `Bitmap` каждый раз, когда вам нужно создать образ, приводит к значительной нагрузке на систему (она должна выделить память в RAM, пока сборщик мусора убирает все ранее созданные экземпляры, на которые больше не ссылаются). Традиционное решение этой проблемы — организация пула `Bitmap`, что является лишь частным случаем более общего паттерна *"Организация пула объектов"*. Вместо того чтобы создавать новый экземпляр `Bitmap`, можно выбрать один из пула (и по возможности повторно использовать базовый буфер).

Вот как можно было бы изменить `collectShapes`, чтобы она принимала дополнительный параметр канала:

```
private fun CoroutineScope.collectShapes(
    locations: ReceiveChannel<Location>,
```

```
    locationsToProcess: SendChannel<Location>,
) = launch {
    val locationsBeingProcessed = mutableListOf<Location>()

    for (loc in locations) {
        if (!locationsBeingProcessed.contains(loc)) {
            launch(Dispatchers.IO) {
                locationsToProcess.send(loc)
            }
        }
    }
}
```

Обратите внимание, что теперь `collectShapes` отправляет местоположение в канал `locationToProcess`, только если оно отсутствует в списке местоположений, которые обрабатываются в данный момент.

Что касается реализации рабочей сопрограммы, то она просто читает данные из только что созданного канала, за исключением того, что, с точки зрения этой сопрограммы, это `ReceiveChannel`. Используем тот же паттерн:

```
private fun CoroutineScope.worker(
        locationsToProcess: ReceiveChannel<Location>,
) = launch(Dispatchers.IO) {
        for (loc in locationsToProcess) {
            // извлекаем ShapeData, вы увидите это позже
        }
}
```

На данный момент мы не сосредоточиваемся на том, как получить `ShapeData`. Самое важное здесь — это цикл `for`. Благодаря перебору канала `locationsToProcess` каждая отдельная рабочая сопрограмма будет получать собственное местоположение, не мешая другим. Неважно, сколько рабочих сопрограмм мы запустим: местоположение, отправленное из `collectShapes` в канал `locationToProcess`, получит только одна сопрограмма. Вы увидите, что каждая рабочая сопрограмма будет создаваться с одним и тем же экземпляром канала, когда все это заработает. В программном обеспечении, ориентированном на обработку сообщений, такой паттерн, подразумевающий доставку сообщения нескольким адресатам, называется *демультиплексором (fan-out)*.

Вернемся к отсутствующей реализации в цикле `for`. Вот что мы сделаем:

1. Получим `ShapeData` (в дальнейшем мы будем называть его просто "данные").
2. Создадим фигуру (`Shape`) из местоположения и данных.
3. Отправим фигуру в некий канал, который другие компоненты приложения будут использовать для получения фигур из `ShapeCollector`. Очевидно, что мы еще не создали его.

4. Уведомим сопрограмму collectShapes о том, что данное местоположение было обработано, отсылая его обратно отправителю. Опять же, такой канал нужно создать.

Обратите внимание, что это не единственная возможная реализация. Могут быть и другие способы адаптироваться к своим потребностям. В конце концов, именно в этом и состоит цель данной главы: показать вам примеры и вдохновить вас на последующую разработку.

Мы снова на коне. В примере 9.1 показана окончательная реализация рабочей сопрограммы.

Пример 9.1. Рабочая сопрограмма

```
private fun CoroutineScope.worker(
    locationsToProcess: ReceiveChannel<Location>,
    locationsProcessed: SendChannel<Location>,
    shapesOutput: SendChannel<Shape>
) = launch(Dispatchers.IO) {
    for (loc in locationsToProcess) {
        try {
            val data = getShapeData(loc)
            val shape = Shape(loc, data)
            shapesOutput.send(shape)
        } finally {
            locationsProcessed.send(loc)
        }
    }
}
```

Точно так же, как ранее мы адаптировали collectShapes, чтобы принимать один канал в качестве аргумента, на этот раз мы добавляем еще два канала: locationProcessed и shapeOutput.

В цикле for сначала мы получаем экземпляр ShapeData для местоположения. Наша реализация показана в примере 9.2.

Пример 9.2. Получение данных формы

```
private suspend fun getShapeData(
    location: Location
): ShapeData = withContext(Dispatchers.IO) {
        /* Моделируем задержку удаленного API */
        delay(10)
        ShapeData()
}
```

Поскольку метод getShapeData может не сразу вернуть управление, мы реализуем его как функцию, поддерживающую возможность приостановки. Если представить, что подчиненный код включает в себя удаленный API, мы используем Dispatchers.IO.

Сопрограмму collectShapes нужно снова адаптировать, т. к. она должна принять еще один канал — откуда рабочие сопрограммы отправляют обратно местоположения, которые они закончили обрабатывать. Это ReceiveChannel. Теперь collectShapes принимает два канала ReceiveChannel и один канал SendChannel.

Давайте попробуем:

```
private fun CoroutineScope.collectShapes(
    locations: ReceiveChannel<Location>,
    locationsToProcess: SendChannel<Location>,
    locationsProcessed: ReceiveChannel<Location>
): Job = launch {
    ...
    for (loc in locations) {
        // та же реализация, мы скрыли ее для краткости
    }
    // но... как перебирать locationsProcessed?
}
```

Теперь у нас проблема. Как получать элементы из нескольких каналов ReceiveChannel одновременно? Если добавить еще один цикл for непосредственно под итерацией канала locations, он не будет работать должным образом, поскольку первая итерация заканчивается только тогда, когда этот канал закрыт.

Для этой цели можно использовать выражение select.

Выражение *select*

Выражение select ожидает результата одновременной работы нескольких функций, допускающих приостановку, которые задаются с помощью *предложений* в теле вызова select. Вызывающий код приостанавливается до тех пор, пока не будет *выбрано* одно из предложений или *произойдет сбой*.

В нашем случае это работает так:

```
select<Unit> {
    locations.onReceive { loc ->
        // выполняем действие 1
    }
    locationsProcessed.onReceive { loc ->
        // выполняем действие 2
    }
}
```

Если бы выражение select могло говорить, оно бы сказало: "Всякий раз, когда канал locations получает элемент, я выполняю действие 1. Или, если канал locationProcessed что-то получает, я выполняю действие 2. Я не могу выполнять оба действия одновременно. Кстати, я возвращаю значение типа Unit".

"Я не могу выполнять оба действия одновременно" — очень важная фраза. Возможно, вам интересно, что произойдет, если действие 1 займет полчаса или, что еще хуже, если оно никогда не завершится. Аналогичная ситуация описана в *разд. "Взаимоблокировки в CSP" далее в этой главе*. Однако следующая реализация гарантированно *никогда* не будет блокироваться в течение длительного времени в каждом действии.

Поскольку select — это выражение, оно возвращает результат. Тип результата определяется типом возвращаемого значения лямбда-выражений, которые мы предоставляем для каждого выражения select — очень похоже на выражение when. В данном конкретном примере нам не нужен никакой результат, поэтому тип возвращаемого значения — Unit. Поскольку select возвращает управление после того, как каналы location или locationProcessed получают элемент, он не выполняет перебор каналов, как в предыдущем цикле for. Следовательно, нужно поместить его внутрь while (true). Полная реализация collectShapes приведена в примере 9.3.

Пример 9.3. collectShapes

```
private fun CoroutineScope.collectShapes(
    locations: ReceiveChannel<Location>,
    locationsToProcess: SendChannel<Location>,
    locationsProcessed: ReceiveChannel<Location>
) = launch(Dispatchers.Default) {

    val locationsBeingProcessed = mutableListOf<Location>()

    while (true) {
        select<Unit> {
            locationsProcessed.onReceive {                    ❶
                locationsBeingProcessed.remove(it)
            }
            locations.onReceive {                             ❷
                if (!locationsBeingProcessed.any { loc ->
                    loc == it }) {
                    /* Добавляем его в список обрабатываемых местоположений */
                    locationsBeingProcessed.add(it)

                    /* Теперь скачиваем фигуру */
                    locationsToProcess.send(it)
```

```
                }
            }
        }
    }
}
```

❶ Когда канал `locationProcessed` получает местоположение, мы знаем, что это местоположение было обработано рабочей сопрограммой. Теперь его нужно удалить из списка обрабатываемых местоположений.

❷ Когда канал `locations` получает местоположение, сначала нужно проверить, обрабатывали мы одно и то же местоположение или нет. Если нет, добавим его в список `LocationBeingProcessed`, а затем отправим в канал `locationToProcess`.

Собираем все воедино

Окончательный вариант архитектуры `ShapeCollector` принимает форму, которая показана на рис. 9.6.

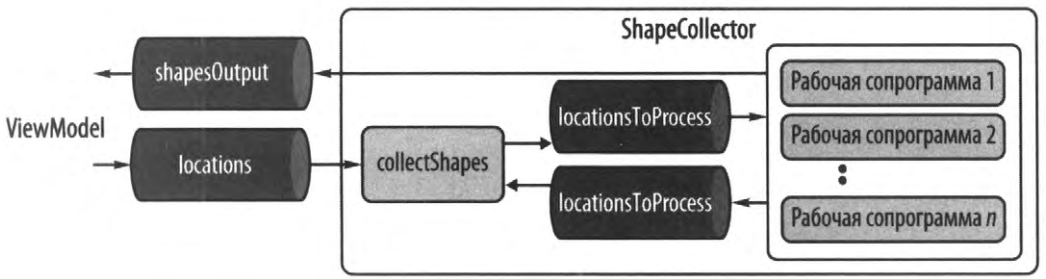


Рис. 9.6. Окончательный вариант

Помните, что все каналы, которые мы использовали для реализации `collectShapes` и методов `worker`, нужно где-то создать. С целью соблюдения инкапсуляции лучше всего это делать в методе `start`, как показано в примере 9.4.

Пример 9.4. ShapeCollector

```
class ShapeCollector(private val workerCount: Int) {
    fun CoroutineScope.start(
        locations: ReceiveChannel<Location>,
        shapesOutput: SendChannel<Shape>
    ) {
        val locationsToProcess = Channel<Location>()
        val locationsProcessed = Channel<Location>(capacity = 1)

        repeat(workerCount) {
            worker(locationsToProcess, locationsProcessed, shapesOutput)
```

```
        }
        collectShapes(locations, locationsToProcess, locationsProcessed)
    }
    private fun CoroutineScope.collectShapes // уже реализовано

    private fun CoroutineScope.worker        // уже реализовано

    private suspend fun getShapeData         // уже реализовано
}
```

Метод `start` отвечает за запуск всего механизма сбора фигур. Создаются два канала, которые используются исключительно в `ShapeCollector`: `LocationsToProcess` и `LocationsProcessed`. Здесь мы не создаем экземпляры `ReceiveChannel` или `SendChannel` явно. Мы создаем их как экземпляры `Channel`, потому что в дальнейшем они будут использоваться либо как `ReceiveChannel`, либо как `SendChannel`. Затем мы создаем и запускаем пул рабочих сопрограмм, вызывая метод `worker` столько раз, сколько было установлено `workerCount`. Это достигается с помощью функции `repeat` из стандартной библиотеки.

Наконец, мы вызываем `collectShapes` один раз. В целом в методе `start` мы запустили `workerCount + 1` сопрограмм.

Возможно, вы заметили, что `locationProcessed` создается с вместимостью, равной 1. Это сделано не случайно, и это важная деталь. В следующем разделе мы объясним почему.

Мультиплексор и демультиплексор

Вы только что видели пример, в котором несколько сопрограмм выполняют операции чтения из одного и того же канала. И правда, все рабочие сопрограммы используют для этого канал `LocationToProcess`. Экземпляр `Location`, отправленный в канал `LocationsToProcess`, будет обрабатываться только одной рабочей сопрограммой без риска возникновения проблем параллелизма. Такое конкретное взаимодействие между сопрограммами представляет собой паттерн, известный как *мультиплексор (fan-in)*. Посмотрите на рис. 9.7. С точки зрения сопрограммы, запущенной с помощью функции `collectShapes`, местоположения распределяются по пулу рабочих сопрограмм.

Разветвление достигается запуском нескольких сопрограмм, которые перебирают один и тот же экземпляр `ReceiveChannel` (см. реализацию `worker` из примера 9.1). Если одна из рабочих сопрограмм дает сбой, остальные сопрограммы продолжат выполнение операций чтения из канала, что делает систему в какой-то степени устойчивой.

И наоборот, когда несколько сопрограмм отправляют элементы в один и тот же экземпляр `SendChannel`, мы говорим о паттерне *"демультиплексор" (fan-out)*. Опять же,

у нас есть хороший пример, т. к. все рабочие сопрограммы отправляют экземпляры Shape в shapeOutput.

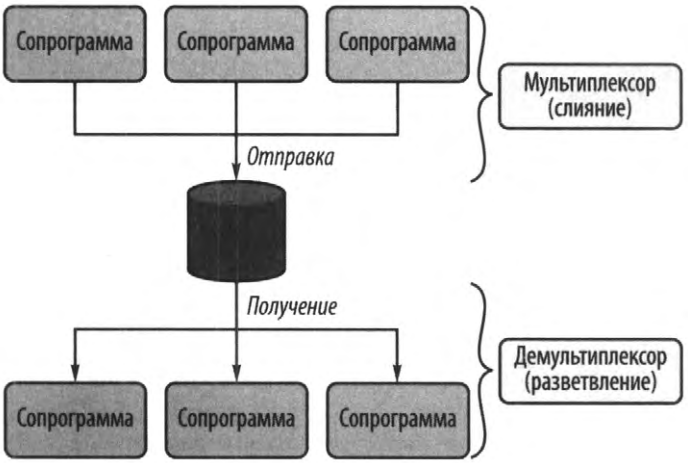


Рис. 9.7. Мультиплексор и демультиплексор

Проверка производительности

Отлично! Пришло время протестировать производительность ShapeCollector. В следующем фрагменте кода есть функция main, вызывающая функции ConsumerShapes и sendLocations. Эти функции запускают сопрограмму, которая потребляет экземпляры Shape из ShapeCollector и отправляет экземпляры Location. В целом этот код близок к тому, что мы бы написали в реальном компоненте ViewModel, как показано в примере 9.5.

Пример 9.5. ShapeCollector

```
fun main() = runBlocking<Unit {
    val shapes = Channel<Shape>()                    ❶
    val locations = Channel<Location>()

    with (ShapeCollector(4)) {                       ❷
        start(locations, shapes)
        consumeShapes(shapes)
    }

    sendLocations(locations)
}

var count = 0
```

```
fun CoroutineScope.consumeShapes(
    shapesInput: ReceiveChannel<Shape>
) = launch {
    for (shape in shapesInput) {
        // Увеличиваем счетчик фигур
        count++                                    ❸
    }
}

fun CoroutineScope.sendLocations(
    locationsOutput: SendChannel<Location>
) = launch {
    withTimeoutOrNull(3000) {                      ❹
        while (true) {
            /* Имитируем извлечение местоположения фигуры */
            val location = Location(Random.nextInt(), Random.nextInt())
            locationsOutput.send(location)
        }
    }
    println("Received $count shapes")
}
```

❶ Настраиваем каналы в соответствии с потребностями `ShapeCollector` (см. рис. 9.4).

❷ Создаем `ShapeCollector` с четырьмя рабочими сопрограммами.

❸ Функция `ConsumerShapes` только увеличивает значение счетчика. Этот счетчик объявляется глобально. И это нормально, потому что сопрограмма, запускаемая функцией `ConsumerShapes`, — единственная сопрограмма, изменяющая `count`.

❹ В функциях `sendLocations` мы устанавливаем время ожидания равным 3 секундам. `withTimeoutOrNull` — это функция, поддерживающая возможность приостановки, которая приостанавливает выполнение до тех пор, пока не истечет указанное время. Следовательно, сопрограмма, запущенная с помощью `sendLocations`, выводит полученный счетчик только через 3 секунды.

Если вы помните реализацию функции `getShapeData` из примера 9.2, то мы добавили функцию `delay(10)` для имитации вызова, допускающего приостановку, длительностью 10 мс. Запустив четыре рабочие сопрограммы в течение 3 секунд, в идеале мы бы получили 3000 : 10 · 4 = 1200 фигур, если бы наша реализация не несла затрат. У себя на компьютере мы получили 1170 фигур. Эффективность — 98%.

Немного поэкспериментировав с бóльшим количеством рабочих сопрограмм (64) с задержкой, равной 5, мы получили 122 518 фигур за 10 секунд (идеальное число — 128 000). Эффективность — 96%.

В целом пропускная способность ShapeCollector вполне приличная, даже с функцией sendLocations, которая непрерывно отсылает экземпляры Location, не делая паузы между двумя отправлениями.

Противодавление

Что произойдет, если наши рабочие сопрограммы будут слишком медленными? Такое вполне может случиться, если удаленному HTTP-вызову требуется время для ответа, или если внутренний сервер перегружен — мы не знаем. Чтобы сымитировать это, можно значительно увеличить задержку в getShapeData (см. пример 9.2). Используя delay(500), мы получили только 20 фигур за 3 секунды с четырьмя рабочими сопрограммами. Пропускная способность уменьшилась, но это не самое интересное. Как всегда в случае проблем между производителем и потребителем неприятности могут возникнуть, когда потребители замедляются, поскольку производители могут накапливать данные, и в конечном счете системе может не хватить памяти. Можно добавить журналы println() в сопрограмме-производителе и снова запустить программу:

```
fun CoroutineScope.sendLocations(locationsOutput: SendChannel<Location>) =
launch {
    withTimeoutOrNull(3000) {
        while (true) {
            /* Имитация извлечения местоположения фигуры */
            val location = Location(Random.nextInt(), Random.nextInt())
            println("Sending a new location")
            locationsOutput.send(location) // вызов, допускающий приостановку
        }
    }
    println("Received $count shapes")
}
```

Теперь фраза "Sending a new location" выводится в консоль всего около 25 раз.

В результате производитель замедляется. Каким образом?

Дело в том, что locationOutput.send(location) — это вызов, допускающий приостановку. Когда рабочие сопрограммы работают медленно, функция collectShapes (см. пример 9.3) класса ShapeCollector быстро приостанавливается на строке locationToProcess.send(it). locationsToProcess — это рандеву-канал. Следовательно, когда сопрограмма, запущенная функцией collectShapes, доходит до этой строки, она приостанавливается до тех пор, пока рабочая сопрограмма не будет готова принять местоположение из LocationToProcess. Если ранее упомянутая сопрограмма приостановлена, она больше не может выполнять операции чтения из канала locations, что соответствует locationOutput из предыдущего примера. Вот почему сопрограмма, которая была запущена с помощью функции sendLocations, в свою

очередь, приостанавливается. Когда рабочие сопрограммы, наконец, сделают свое дело, `collectShapes` сможет возобновить работу, равно как и сопрограмма-производитель.

Сходства с моделью акторов

В CSP мы создаем сопрограммы, которые инкапсулируют изменяемое состояние. Вместо того, чтобы взаимодействовать, совместно используя состояние, они совместно используют состояние, взаимодействуя (используя каналы). Сопрограмма, запущенная функцией `collectShapes` (см. пример 9.3), использует три канала для взаимодействия с другими сопрограммами — один канал `SendChannel` и два канала `ReceiveChannel` (рис. 9.8).

На языке CSP `collectShapes` и эти три канала называются *процессом*. Процесс — это вычислительная сущность, которая взаимодействует с другими акторами с помощью асинхронной передачи сообщений (каналов). Одновременно он может делать только что-то одно — читать данные или записывать их в каналы, или обрабатывать.

В модели акторов *актор* — это нечто похожее. Одно заметное отличие состоит в том, что у актора есть только один канал — почтовый ящик. Если актор должен быть отзывчивым и неблокирующим, он должен делегировать длительную обработку дочерним акторам. Это сходство — причина, по которой CSP иногда называют реализацией модели акторов.

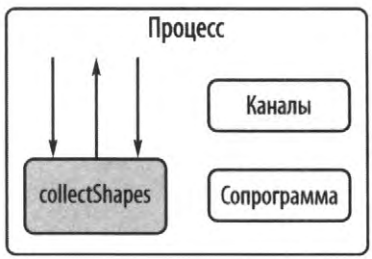


Рис. 9.8. Процесс в CSP

Последовательное выполнение внутри процесса

Мы только что видели, что *процесс* состоит из одной сопрограммы и каналов. Сама природа сопрограммы заключается в том, что сопрограмма должна выполняться в потоке. Поэтому, если она не запускает другие дочерние сопрограммы (которые выполняются последовательно, а в некоторых случаях и параллельно), все строки этой сопрограммы выполняются последовательно. Сюда входят операции чтения из каналов, отправка объектов в другие каналы и изменение закрытого состояния. Следовательно, акторы, реализованные в этой главе, могут либо выполнять операции чтения из канала, либо операции отправки в другой канал, но делать и то и

другое одновременно они не могут. В условиях нагрузки такой актор может быть эффективным, потому что он не включает блокировку вызовов, а только функции, поддерживающие возможность приостановки. Когда сопрограмма приостанавливает выполнение, это не обязательно влияет на общую эффективность, потому что поток, выполняющий приостановленную сопрограмму, может затем выполнить другую сопрограмму, которой нужно что-то сделать. Таким образом, потоки можно использовать в полной мере, не соревнуясь с блокировкой.

Размышления напоследок

У механизма, использующего стиль CSP, очень небольшие внутренние затраты. Благодаря каналам и сопрограммам это реализация без блокировки. Таким образом, *отсутствует конфликт потоков* — `ShapeCollector` с меньшей вероятностью повлияет на другие потоки вашего приложения. Точно так же существует вероятность, что диспетчеры, которые мы задействуем в `ShapeCollector`, могут использоваться в других функциях приложения. Используя реализации без блокировок, сопрограмма, приостановленная во время операции чтения из канала, не помешает базовому потоку выполнять другие сопрограммы. Иными словами, мы можем сделать больше, используя те же ресурсы.

Вдобавок данная архитектура обеспечивает встроенное противодавление. Если для извлечения некоторых экземпляров `ShapeData` внезапно потребуется больше времени, производители экземпляров `ShapeLocation` будут работать медленнее, чтобы местоположения не накапливались, что снижает риск нехватки памяти. И это противодавление мы получаем бесплатно — мы не писали явный код для этого.

Пример, приведенный в этой главе, достаточно общий, чтобы его можно было использовать как есть и адаптировать в соответствии со своими потребностями. В случае, если вам нужно значительно отклониться от него, стоит привести более подробное объяснение. Например, почему мы задали для размера буфера канала `locationProcessed` из примера 9.4 значение 1? Ответ, по общему признанию, нетривиален. Если бы мы создали обычный рандеву-канал, класс `ShapeCollector` страдал бы от *взаимоблокировок*, а это тема следующего раздела.

Взаимоблокировки в CSP

Чаще всего взаимоблокировки возникают при работе с потоками. Когда поток *A* удерживает мьютекс 1 и пытается захватить мьютекс 2, а поток *B* удерживает мьютекс 2 и пытается захватить мьютекс 1, возникает взаимоблокировка. Два потока бесконечно ждут друг друга, и ни один из них не двигается дальше. Взаимоблокировки могут иметь катастрофические последствия, если происходят в критически важных компонентах приложения. Эффективный способ избежать подобной ситуации — убедиться, что взаимоблокировка не может произойти ни при каких вообразимых обстоятельствах. Даже когда маловероятно, что условия будут соблюдены, можете быть уверены, что закон Мерфи когда-нибудь сработает.

Однако взаимоблокировки также могут возникать и в архитектуре CSP. В качестве иллюстрации можно провести небольшой эксперимент. Вместо значения 1 у вместимости канала `locationsProcessed`, как мы это делали в примере 9.4, воспользуемся каналом без буфера (рандеву-каналом) и выполним образец теста производительности из примера 9.5. Результат, выведенный в консоль, выглядит так:

```
Received 4 shapes
```

Для справки: мы должны были получить 20 фигур. Так в чем же дело?

Справедливое предупреждение: следующее объяснение содержит все необходимые детали, и оно довольно длинное. Мы рекомендуем вам найти время, чтобы внимательно прочитать его до конца. Это решающее испытание, чтобы проверить как вы усвоили тему каналов.

Вы также можете пропустить его и перейти к *разд. "TL;DR" далее в этой главе.*

Поближе посмотрим на внутреннее устройство класса `ShapeCollector` и проследим за каждым шагом, как если бы мы были живым отладчиком. Представьте, что вы только что выполнили тест производительности из примера 9.5, и первый экземпляр `Location` отправляется в канал `locations`. Он проходит через метод `collectShapes` с выражением `select`. В этот момент `locationProcessed` не может ничего предоставить, поэтому выражение `select` проходит через `location.onReceive {..}`. Если посмотреть, что происходило во втором операторе, видно, что местоположение отправляется в `LocationToProcess`, который представляет собой канал получения для каждой рабочей сопрограммы. Следовательно, выполнение сопрограммы, запускаемой методом `collectShapes` (которую мы будем называть сопрограммой `collectShapes`), приостанавливается на вызове `locationsToProcess.send(it)` до тех пор, пока рабочая сопрограмма не обменяется рукопожатием с рандеву-каналом `locationToProcess`. Происходит это довольно быстро, т. к. в это время все рабочие сопрограммы простаивают.

Когда рабочая сопрограмма получает первый экземпляр `Location`, сопрограмма `collectShapes` возобновляет работу и может получать другие местоположения. Здесь мы добавили задержку для имитации фоновой обработки. Можно считать рабочие сопрограммы медленными по сравнению с другими сопрограммами — `collectShapes` и сопрограммой-производителем, запускаемой методом `sendLocations` в тестовом образце (которую мы будем называть сопрограммой `sendLocations`). Следовательно, сопрограмма `collectShapes` получает другое местоположение, в то время как рабочая сопрограмма, которая получила первое местоположение, все еще занята его обработкой. Точно так же вторая рабочая сопрограмма быстро обрабатывает второе местоположение, а третье местоположение получает сопрограмма `collectShapes` и т. д.

Выполнение продолжается до тех пор, пока все четыре рабочие сопрограммы не будут заняты, в то время как сопрограмма `collectShapes` получает пятое местоположение. Следуя той же логике, что и раньше, она приостанавливается до тех пор, пока рабочая сопрограмма не будет готова принять экземпляр `Location`. К сожалению, все рабочие сопрограммы заняты. Поэтому сопрограмма `collectShapes` больше

не может принимать входящие местоположения. Поскольку сопрограммы collectShapes и sendLocations общаются по рандеву-каналу, сопрограмма sendLocations, в свою очередь, приостанавливается до тех пор, пока collectShapes не будет готова принять дополнительные местоположения.

Проходит время, пока рабочая сопрограмма не станет доступной для получения пятого местоположения. В конце концов, рабочая сопрограмма (вероятно, первая) завершает обработку своего экземпляра Location. После этого она отправляет результат в канал shapeOutput и пытается отправить обработанное местоположение обратно в сопрограмму collectShapes, используя канал LocationProcessed. Помните, что это наш механизм для уведомления сопрограммы collectShapes об обработке местоположения. Однако сопрограмма collectShapes приостанавливается при вызове locationToProcess.send(it). Таким образом, collectShapes не может выполнять операции чтения из канала locationProcessed. Это *взаимоблокировка*[4], которая показана на рис. 9.9.

В конце концов, первые четыре местоположения, обработанные рабочими сопрограммами, обрабатываются, и четыре экземпляра Shape отправляются в канал shapeOutput. Задержка в каждой рабочей сопрограмме составляет всего 10 мс, поэтому все рабочие сопрограммы успевают завершить выполнение до времени ожидания, составляющего 3 секунды.

Вот и результат:

```
Received 4 shapes
```

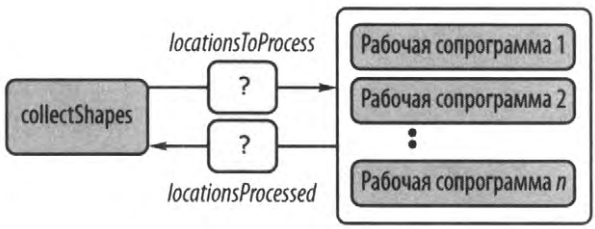


Рис. 9.9. Взаимоблокировка в CSP

Если размер буфера канала locationProcessed составлял бы не менее 1, то первая доступная рабочая сопрограмма могла бы отправить свой экземпляр Location обратно, а затем выполнить операцию чтения из канала locationsToProcess — освобождая сопрограмму collectShapes. Впоследствии в выражении select сопрограммы collectShapes канал locationsToProcess *всегда* проверяется перед каналом location. Это гарантирует, что, когда сопрограмма collectShapes в конечном счете приостанавливается в месте вызова locationsToProcess.send(it), буфер канала locationProcessed гарантированно будет пустым, поэтому рабочая сопрограмма может отправить местоположение, не будучи приостановленной. Если вам интересно, попробуйте из-

[4] Хотя здесь нет мьютекса, эта ситуация очень похожа на взаимоблокировку с участием потоков. Вот почему мы используем те же термины.

менить порядок обработки `locationProcessed.onReceive {..}` и `location.onReceive {..}` при размере буфера, равном 1, для канала `locationProcessed`. Результат будет таким: `Received 5 shapes`.

Каналы и взаимоблокировки

Чрезвычайно важен не только размер буфера канала `locationProcessed`, равный 1, но и порядок, в котором каналы считываются в выражении `select` сопрограммы `collectShapes`[5]. Что следует помнить из этого? В CSP возможны взаимоблокировки. А еще важнее: понимание того, что вызвало взаимоблокировку, является отличным упражнением, чтобы проверить, насколько хорошо вы усвоили работу каналов.

Если вернуться к структуре `ShapeCollector`, то можно представить ее в виде циклического графа (рис. 9.10).

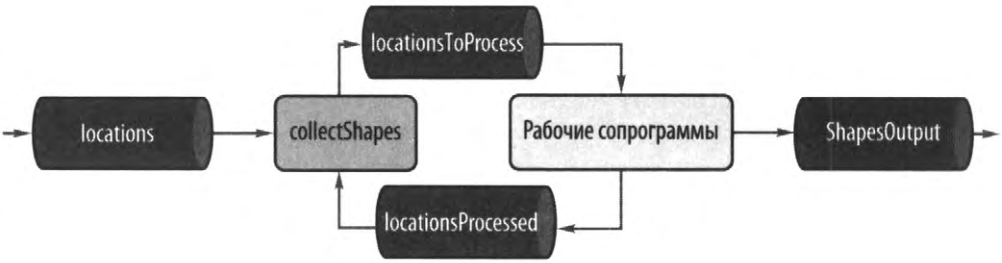


Рис. 9.10. Циклический граф

Это представление подчеркивает важное свойство структуры: она *циклична*. Экземпляры `Location` перемещаются между сопрограммой `collectShapes` и рабочими сопрограммами.

На самом деле циклы в CSP являются причиной взаимоблокировок. Без циклов взаимоблокировки не бывает. Однако иногда кроме них у вас не будет выбора. В данном случае мы привели основные моменты, касающиеся CSP, чтобы вы могли найти решение самостоятельно.

Ограничения каналов

До сих пор мы воздерживались от обсуждения ограничений каналов, поэтому сейчас опишем некоторые из них. Используя понятия из этой главы, создание потока значений типа `Int` обычно выполняется так, как показано в примере 9.6.

---

[5] На самом деле наша реализация, в которой используется размер буфера канала `locationProcessed`, равный 1, не единственная возможная реализация, работающая без взаимоблокировок. Существует по крайней мере одно решение, где `locationProcessed` используется в качестве рандеву-канала. Мы оставим его в качестве упражнения для читателей.

Пример 9.6. Генерация чисел

```
fun CoroutineScope.numbers(): ReceiveChannel<Int> = produce {
    send(1)
    send(2)
    // отсылаем другие числа
}
```

На принимающей стороне можно потреблять эти числа следующим образом:

```
fun main() = runBlocking {
    val channel = numbers()
    for (x in channel) {
        println(x)
    }
}
```

Довольно просто. Но что, если вам нужно применить преобразование для каждого из этих чисел? Представьте, что наша функция преобразования выглядела так:

```
suspend fun transform(n: Int) = withContext(Dispatchers.Default) {
    delay(10) // имитируем вычисления с интенсивным использованием ЦП
    n + 1
}
```

Можно изменить функцию numbers следующим образом:

```
fun CoroutineScope.numbers(): ReceiveChannel<Int> = produce {
    send(transform(1))
    send(transform(2))
}
```

Работает, но выглядит неэлегантно. Вот гораздо более красивое решение:

```
fun main() = runBlocking {
    /* Предупреждение: этот код не компилируется */
    val channel = numbers().map {
        transform(it)
    }
    for (x in channel) {
        println(x)
    }
}
```

На самом деле, начиная с Kotlin версии 1.4, этот код не компилируется. На заре существования каналов у нас были "операторы каналов", например map. Однако в Kotlin версии 1.3 эти операторы устарели, а в Kotlin 1.4 их убрали.

Почему? Каналы — это примитивные типы, поддерживающие возможность взаимодействий сопрограмм. Они специально разработаны для распределения значений таким образом, чтобы каждое значение было получено лишь одним получателем. Невозможно использовать каналы для рассылки значений нескольким получателям. Разработчики сопрограмм создали потоки специально для асинхронных потоков данных, в которых можно использовать операторы преобразования; мы увидим, как это делается в следующей главе.

Таким образом, каналы не являются удобным решением для реализации конвейеров преобразования данных.

"Горячие" каналы

Взглянем на исходный код функции produce. Интерес представляют две строки:

```
public fun <E> CoroutineScope.produce(                                    ❶
    context: CoroutineContext = EmptyCoroutineContext,
    capacity: Int = 0,
    @BuilderInference block: suspend ProducerScope<E>.() -> Unit
): ReceiveChannel<E> {
    val channel = Channel<E>(capacity)
    val newContext = newCoroutineContext(context)
    val coroutine = ProducerCoroutine(newContext, channel)
    coroutine.start(CoroutineStart.DEFAULT, coroutine, block)             ❷
    return coroutine
}
```

❶ produce — это функция-расширение CoroutineScope. Помните соглашение? Оно указывает на то, что данная функция запускает новую сопрограмму.

❷ Можно подтвердить это вызовом coroutine.start(). Не обращайте слишком много внимания на то, как запускается эта сопрограмма, — это внутренняя реализация.

Следовательно, при вызове функции produce запускается новая сопрограмма, которая немедленно начинает производить элементы и отправлять их в возвращаемый канал, даже если ни одна из сопрограмм их не потребляет.

Вот почему каналы называют *"горячими"*: сопрограмма активно работает для производства или потребления данных. Если вы знаете RxJava, то это та же концепция, что и "горячие" объекты Observable: они выдают значения независимо от индивидуальных подписок. Рассмотрим простой поток:

```
fun CoroutineScope.numbers(): ReceiveChannel<Int> = produce {
    use(openConnectionToDatabase()) {
        send(1)
        send(2)
    }
}
```

Представьте, что другие сопрограммы не потребляют этот поток. Поскольку эта функция возвращает рандеву-канал, запущенная сопрограмма приостанавливается на первой функции send. Поэтому вы можете сказать: "Хорошо, у нас все в порядке — фоновая обработка не выполняется, пока мы не предоставим потребителя этому потоку". Это правда, но если вы забудете использовать поток, то соединение с базой данных останется открытым — обратите внимание, что мы использовали функцию use из стандартной библиотеки, которая эквивалентна оператору try-with-resources в Java.

Хотя сам по себе, возможно, данный фрагмент логики и не несет вреда, он может быть частью цикла повторных попыток, и в этом случае произойдет утечка значительного объема ресурсов.

Подводя итог, можно сказать, что каналы — это примитивные типы, поддерживающие возможность взаимодействий сопрограмм. Они очень хорошо работают в архитектуре, подобной CSP. Однако у нас нет удобных операторов, таких как map или filter, для их преобразования. Мы не можем передавать значения нескольким получателям. Более того, в некоторых ситуациях их "горячий" характер может вызвать утечку памяти.

Для устранения ограничений этих каналов были созданы потоки, которые мы рассмотрим в следующей главе.

Резюме

- Каналы — это примитивные типы, обеспечивающие способ передачи потоков значений между сопрограммами.
- Хотя на концептуальном уровне каналы близки к BlockingQueue в Java, фундаментальное отличие состоит в том, что методы каналов send и receive — это функции, поддерживающие возможность приостановки, а не блокирующие вызовы.
- Посредством каналов и сопрограмм можно совместно использовать состояние, взаимодействуя, вместо того чтобы *взаимодействовать, совместно используя состояние*. Цель состоит в том, чтобы избежать проблем, связанных с совместным использованием изменяемого состояния и безопасностью потоков.
- Можно реализовать сложную логику, используя стиль CSP, с помощью противодавления. Это обеспечивает потенциально превосходную производительность, поскольку неблокирующий характер функций, поддерживающих возможность приостановки, сводит конфликты между потоками к минимуму.
- Имейте в виду, что в CSP возможна взаимоблокировка, если в архитектуре есть циклы (сопрограмма отправляет объекты в другую сопрограмму, а также получает объекты из той же сопрограммы). Можно исправить это, например, изменив порядок, в котором выражение select обрабатывает каждый вариант, или отрегулировав размер буфера некоторых каналов.

- Каналы следует рассматривать как низкоуровневые примитивные типы. Взаимоблокировки в CSP — один из примеров неправильного использования каналов. В следующей главе будут представлены *потоки* — примитивные типы более высокого уровня, которые обмениваются потоками данных между сопрограммами. Это не означает, что вы не должны применять каналы — по-прежнему случаются ситуации, когда они необходимы (например, `ShapeCollector` из этой главы). Однако вы увидите, что во многих ситуациях потоки являются более подходящим выбором. В любом случае важно иметь представление о каналах, потому что (как вы увидите) иногда потоки используют каналы "под капотом".

ГЛАВА 10
Потоки

Мы уже рассмотрели сопрограммы — функции, поддерживающие возможность приостановки, и способы работы с потоками с помощью примитивного типа `Channel`. В предыдущей главе было показано, что работа с этими типами подразумевает запуск сопрограмм для выполнения операций отправки и чтения из каналов. Вышеупомянутые сопрограммы представляют собой *горячие* сущности, которые иногда трудно поддаются отладке или могут привести к утечке ресурсов, если не будут отменены, когда это нужно.

Примитивный тип `Flow`, как и `Channel`, предназначен для обработки асинхронных потоков данных, но на более высоком уровне абстракции и с лучшими инструментами. На концептуальном уровне он похож на последовательности, за исключением того, что каждый этап `Flow` может быть асинхронным. Также его легко интегрировать в структурированный параллелизм, чтобы избежать утечки ресурсов.

Однако потоки[1] не предназначены для замены каналов. Каналы — это строительные блоки потоков. Они по-прежнему подходят для некоторых архитектур, таких как CSP (*см. главу 9*). Тем не менее вы увидите, что потоки удовлетворяют большинству потребностей асинхронной обработки данных.

В этой главе мы познакомим вас с холодными и горячими потоками. Вы увидите, что *холодные* потоки больше подходят, если нужна уверенность в том, что утечка ресурсов не произойдет. *Горячие* потоки служат иной цели, например в ситуации, когда вам нужна связь "издатель — подписчик" между сущностями в приложении. К примеру, используя горячие потоки, можно реализовать шину событий.

Лучший способ освоить потоки — посмотреть, как они используются в реальных приложениях. Поэтому в данной главе также будет рассмотрен ряд типичных вариантов применения.

Введение в потоки

Снова реализуем пример 9.6, используя тип `Flow`:

```
fun numbers(): Flow<Int> = flow {
    emit(1)
```

[1] В оставшейся части главы мы будем называть тип `Flow` потоком.

```
    emit(2)
    // эмитируем другие значения
}
```

Здесь важно отметить несколько аспектов:

1. Вместо экземпляра `Channel` мы возвращаем экземпляр `Flow`.
2. В потоке вместо функции `send` используется функция `emit`, поддерживающая возможность приостановки.
3. Функция `numbers`, возвращающая экземпляр `Flow`, не является функцией с поддержкой приостановки. Ее вызов сам по себе ничего не запускает — она просто незамедлительно возвращает экземпляр `Flow`.

Подведем итог: в блоке `flow` мы определяем, как эмитировать значения. При вызове функция `numbers` быстро возвращает экземпляр `Flow`, ничего не запуская в фоновом режиме.

Что же происходит у потребителя? А вот:

```
fun main() = runBlocking {
    val flow = numbers()    ❶
    flow.collect {          ❷
        println(it)
    }
}
```

❶ Мы получаем экземпляр `Flow`, используя функцию `numbers`.

❷ Как только мы получаем поток, вместо того чтобы перебирать его (как в случае с каналом), мы используем функцию `collect`, которая на языке потоков называется *терминальным оператором*. Мы продолжим работу с *операторами потоков* и терминальными операторами в *разд. "Операторы" далее в этой главе*. На данный момент можно резюмировать назначение оператора `collect` следующим образом: он потребляет поток; например, перебирает его и выполняет заданное лямбда-выражение для каждого элемента потока.

Вот и все — мы продемонстрировали вам базовое использование потока. Как упоминалось ранее, теперь мы возьмем более реалистичный пример, чтобы вы увидели реальную пользу потоков.

Более реалистичный пример

Представьте, что нам нужно получить токены из удаленной базы данных[2], а затем запросить дополнительные данные для каждого из этих токенов.

---

[2] Обычно токен представляет собой зашифрованные регистрационные данные, которые клиентское приложение хранит в памяти, чтобы при дальнейшем доступе к базе данных явная аутентификация не требовалась.

Нужно делать это только время от времени, т. к. мы решили не поддерживать активное соединение с базой данных (что может быть недешево) и создаем соединение лишь при извлечении данных, закрывая его после завершения работы.

Сначала нужно установить соединение с базой данных. Затем мы получаем токен, используя функцию `getToken`, которая поддерживает возможность приостановки. Эта функция выполняет запрос к базе данных и возвращает токен. После этого мы асинхронно получаем необязательные данные, ассоциированные с этим токеном. В нашем примере это делается путем вызова функции `getData`, которая также поддерживает возможность приостановки и принимает токен в качестве параметра. Как только мы получаем результат от `getData`, мы заключаем его и токен в экземпляр класса `TokenData`, определенный как:

```
data class TokenData(val token: String, val opt: String? = null)
```

Подведем итог: нам нужно создать поток объектов `TokenData`. Сначала этот поток требует установить соединение с базой данных, а затем выполнять асинхронные запросы для извлечения токенов и получения связанных данных. Мы сами выбираем, сколько токенов нам нужно. После обработки всех токенов мы отключаемся и освобождаем базовые ресурсы подключения к базе данных. На рис. 10.1 показано, как реализовать такой поток.

```
private fun getDataFlow(n: Int): Flow<TokenData>{
    return flow {  this: FlowCollector<TokenData>
        connect()
        repeat(n) {  it: Int
            val token = getToken()
            val data = getData(token)
            emit(TokenData(token, data))
        }
    }.onCompletion {  this: FlowCollector<TokenData>
        disconnect()
    }
}
```

Рис. 10.1. Поток данных

Соответствующий исходный код можно найти на сайте GitHub (**https://oreil.ly/dU4uZ**).

В этой главе вместо блоков кода иногда используются изображения, потому что на скриншотах из нашей интегрированной среды разработки видны точки приостановки (на полях) и подсказки при вводе кода, которые действительно полезны.

Особенно важно отметить несколько аспектов этой реализации.

- Создание подключения к базе данных и его закрытие по завершении работы полностью прозрачно для клиентского кода, потребляющего поток. Клиентский код видит только поток `TokenData`.

- Все операции в потоке выполняются последовательно. Например, как только мы получим первый токен (скажем, "token1"), поток вызывает `getData("token1")` и приостанавливается до тех пор, пока не получит результат (скажем, "data1"). Затем поток выдает первый `TokenData("token1," "data1")`. Только после этого выполнение продолжается с "token2" и т. д.
- Вызов функции `getDataFlow` сам по себе ничего не делает. Он просто возвращает поток. Код в потоке выполняется только тогда, когда сопрограмма собирает поток, как показано в примере 10.1.

Пример 10.1. Сбор потока

```
fun main() = runBlocking<Unit> {
    val flow = getDataFlow(3) // При инициализации ничего не выполняется

    // Сопрограмма собирает поток
    launch {
        flow.collect { data ->
            println(data)
        }
    }
}
```

- Если сопрограмма, которая собирает поток, отменяется или доходит до конца потока, выполняется код внутри блока `onCompletion`, что гарантирует правильный разрыв соединения с базой данных.

Как мы уже упоминали, `collect` — это оператор, потребляющий все элементы потока. В данном примере он вызывает функцию для каждого собранного элемента потока (например, `println(data)` вызывается трижды). О других терминальных операторах мы расскажем в *разд. "Примеры использования холодного потока" далее в этой главе.*

Пока вы видели примеры потоков, которые не выполняют никакого кода, пока сопрограмма не соберет их. Говоря языком потоков, это холодные потоки.

Операторы

Если вам нужно выполнить преобразования в потоке, как если бы вы делали это в коллекциях, обратите внимание на библиотеку сопрограмм: она предоставляет такие функции, как `map`, `filter`, `debounce`, `buffer`, `onCompletion` и т. д. Эти функции называются *операторами потоков* или *промежуточными операторами*, поскольку они

работают с потоком и возвращают другой поток. Обычный оператор не следует путать с терминальным, как будет показано позже.

Ниже приведен пример использования оператора map:

```
fun main() = runBlocking<Unit> {
    val numbers: Flow<Int> = // реализация скрыта для краткости

    val newFlow: Flow<String> = numbers().map {
        transform(it)
    }
}

suspend fun transform(i :Int): String = withContext(Dispatchers.Default) {
    delay(10) // имитация реальной работы
    "${i + 1}"
}
```

Интересно, что оператор map превращает тип Flow<Int> в Flow<String>. Тип получившегося потока определяется типом значения, возвращаемым лямбда-выражением, передаваемым оператору.

На концептуальном уровне оператор map очень близок к функции-расширению map в коллекциях. Хотя есть заметное отличие: лямбда-выражение, передаваемое оператору map, может быть функцией, поддерживающей возможность приостановки.

Мы рассмотрим большинство распространенных операторов в ряде случаев использования в следующем разделе.

Терминальные операторы

Терминальный оператор легко можно отличить от других обычных операторов, поскольку это функция, поддерживающая возможность приостановки, которая запускает сбор потока. Вы уже видели оператор collect.

Доступны и другие терминальные операторы, такие как toList, collectLatest, first и т. д. Вот их краткое описание:

- toList собирает данный поток и возвращает List, содержащий все собранные элементы;
- collectLatest собирает данный поток с заданным действием. Его отличие от оператора collect заключается в том, что, когда исходный поток выдает новое значение, блок действий для предыдущего значения отменяется;
- first возвращает первый элемент, выдаваемый потоком, а затем отменяет сбор потока. Если поток был пуст, он выбрасывает исключение NoSuchElementException. Также есть разновидность этого оператора — firstOrNull, который возвращает null, если поток был пустым.

Примеры использования холодного потока

Как оказалось, выбор одного-единственного примера с использованием всех возможных операторов не лучший путь. Вместо этого мы представим различные варианты использования, которые проиллюстрируют применение нескольких операторов.

Вариант 1: интерфейс с API на базе функции обратного вызова

Предположим, что вы разрабатываете приложение для чата. Ваши пользователи могут отправлять сообщения друг другу. У сообщений есть дата, ссылка на автора сообщения, а их содержимое выглядит как обычный текст.

Это класс данных `Message`:

```
data class Message(
    val user: String,
    val date: LocalDateTime,
    val content: String
)
```

Неудивительно, что мы будем представлять поток сообщений как поток экземпляра `Message`. Каждый раз, когда пользователь отправляет сообщение в приложение, поток будет передавать его. А пока предположим, что можно вызвать функцию `getMessageFlow`, которая возвращает экземпляр `Flow<Message>`. С помощью библиотеки Kotlin Flows можно создавать собственные потоки. Тем не менее имеет смысл начать с изучения того, как использовать потоковый API в распространенных случаях:

```
fun getMessageFlow(): Flow<Message> {
    // мы реализуем его позже
}
```

Теперь предположим, что нам на лету нужно переводить все сообщения от данного пользователя на другой язык. Кроме того, мы хотим выполнять перевод, используя фоновый поток.

Мы начнем с получения потока сообщений, вызвав функцию `getMessageFlow()`, а затем применим операторы к исходному потоку, как показано ниже:

```
fun getMessagesFromUser(user: String, language: String): Flow<Message> {
    return getMessageFlow()
        .filter { it.user == user }          ❶
        .map { it.translate(language) }      ❷
        .flowOn(Dispatchers.Default)         ❸
}
```

❶ Первый оператор, `filter`, работает с исходным потоком и возвращает другой поток сообщений. Все эти потоки идут от одного и того же пользователя (`user`), передаваемого в качестве параметра.

❷ Второй оператор, `map`, работает с потоком, возвращаемым `filter`, и возвращает поток переведенных сообщений. С точки зрения оператора `filter`, исходный поток (возвращаемый функцией `getMessageFlow()`) является *вышестоящим*, а *подчиненный поток* представлен всеми операторами, которые идут *после* `filter`. Те же рассуждения применимы ко всем промежуточным операторам — у них есть свои соответствующие вышестоящий и подчиненный потоки (рис. 10.2).

❸ Наконец, оператор `flowOn` изменяет контекст потока, с которым он работает. Он изменяет контекст сопрограммы вышестоящего потока, не затрагивая при этом подчиненный поток. Таким образом, шаги 1 и 2 выполняются с помощью диспетчера `Dispatchers.Default`.

Другими словами, операторы вышестоящего потока (такие как `filter` и `map`) теперь инкапсулированы: их контекст выполнения — `Dispatchers.Default`, и так будет всегда. Неважно, в каком контексте будет собираться получившийся поток; ранее упомянутые операторы будут выполняться с использованием `Dispatchers.Default`.

Это очень важное свойство потоков, которое называется *сохранением контекста*. Представьте, что вы собираете поток в UI-потоке приложения — как правило, для этой цели используется `viewModelScope` из `ViewModel`. Было бы неловко, если бы контекст выполнения одного из операторов потока просочился вниз и повлиял на поток, в котором в конечном счете был собран `Flow`. К счастью, этого никогда не случится. Например, если вы собираете `Flow` в UI-потоке, все значения выдаются сопрограммой, использующей `Dispatchers.Main`. Все необходимые переключения контекста управляются автоматически.

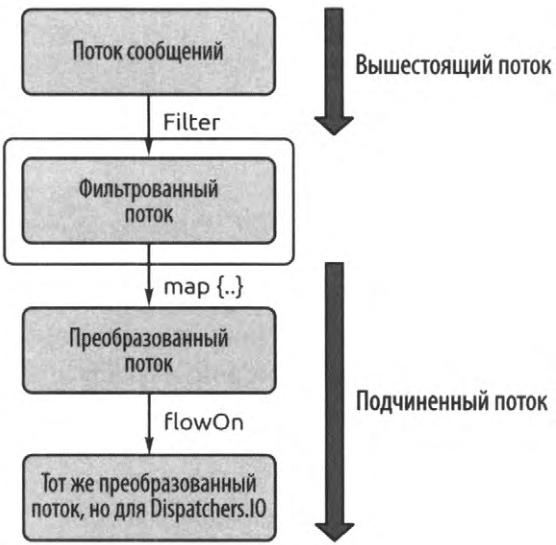


Рис. 10.2. Вышестоящий и подчиненный потоки

"Под капотом" оператор `flowOn` запускает новую сопрограмму, когда обнаруживает, что контекст вот-вот изменится. Эта сопрограмма взаимодействует с остальной частью потока через канал, управляемый изнутри.

Говоря языком потоков, промежуточный оператор, например `map`, работает с вышестоящим потоком и возвращает другой поток. С точки зрения оператора `map` возвращаемый поток является подчиненным.

Оператор `map` принимает функцию, поддерживающую возможность приостановки, в качестве блока преобразования. Итак, если бы нам нужно было выполнить перевод сообщений, используя только `Dispatchers.Default` (а не фильтрацию сообщений), можно было бы удалить оператор `flowOn` и объявить функцию `translate`:

```
private suspend fun Message.translate(
    language: String
): Message = withContext(Dispatchers.Default) {
    // это фиктивная реализация
    copy(content = "translated content")
}
```

Видите, как легко можно перенести фрагменты преобразования данных в другие потоки, располагая при этом общей картиной потока данных?

Как видите, API потоков позволяет декларативно выражать преобразование данных. При вызове `getMessagesFromUser("Amanda," "en-us")` фактически ничего не выполняется. Все эти преобразования включают промежуточные операторы, которые сработают, когда поток будет собран.

Что касается потребителя: если нам нужно передавать каждое полученное сообщение, то можно использовать функцию `collect`:

```
fun main() = runBlocking {
    getMessagesFromUser("Amanda", "en-us").collect {
        println("Received message from ${it.user}: ${it.content}")
    }
}
```

Теперь, когда мы показали, как преобразовать поток и потреблять его, можно предоставить реализацию самого потока: функцию `getMessageFlow`. Сигнатура этой функции должна возвращать поток сообщений. В данной конкретной ситуации разумно предположить, что на самом деле механизм сообщений — это служба, которая выполняется в собственном потоке. Назовем ее `MessageFactory`.

Как и у большинства подобных служб, у `MessageFactory` имеется механизм *"издатель/подписчик"* — мы можем регистрировать или отменять регистрацию наблюдателей для новых входящих сообщений, как показано ниже:

```
abstract class MessageFactory : Thread() {
    /* Внутренний список наблюдателей должен быть потокобезопасным */
    private val observers = Collections.synchronizedList(
        mutableListOf<MessageObserver>())
    private var isActive = true
```

```
    override fun run() = runBlocking {
        while(isActive) {
            val message = fetchMessage()
            for (observer in observers) {
                observer.onMessage(message)
            }
            delay(1000)
        }
    }

    abstract fun fetchMessage(): Message

    fun registerObserver(observer: MessageObserver) {
        observers.add(observer)
    }

    fun unregisterObserver(observer: MessageObserver) {
        observers.removeAll { it == observer }
    }

    fun cancel() {
        isActive = false
        observers.forEach {
            it.onCancelled()
        }
        observers.clear()
    }

    interface MessageObserver {
        fun onMessage(msg: Message)
        fun onCancelled()
        fun onError(cause: Throwable)
    }
}
```

В этой реализации мы ежесекундно проверяем, есть ли новые сообщения, и уведомляем наблюдателей. Теперь возникает вопрос: как превратить горячую сущность[3], например `MessageFactory`, в поток? `MessageFactory` называют классом *на основе*

[3] В отличие от холодной сущности, горячая сущность функционирует сама по себе, пока не будет остановлена явно.

функции обратного вызова, поскольку он содержит ссылки на экземпляры MessageObserver и вызывает методы этих экземпляров при получении новых сообщений. Чтобы соединить мир потоков и мир "функций обратного вызова", можно использовать функцию создания потоков callbackFlow. В примере 10.2 показано, как это сделать.

Пример 10.2. Создание потока из API на основе функции обратного вызова

```
fun getMessageFlow(factory: MessageFactory) = callbackFlow<Message> {
    val observer = object : MessageFactory.MessageObserver {
        override fun onMessage(msg: Message) {
            trySend(msg)
        }

        override fun onCancelled() {
            channel.close()
        }

        override fun onError(cause: Throwable) {
            cancel(CancellationException("Message factory error", cause))
        }
    }

    factory.registerObserver(observer)
    awaitClose {
        factory.unregisterObserver(observer)
    }
}
```

callbackFlow создает холодный поток, который ничего не выполняет, пока мы не вызовем терминальный оператор. Рассмотрим ее подробнее. Во-первых, это параметризованная функция, возвращающая поток заданного типа. Здесь всегда три этапа:

```
callbackFlow {
    /*
    1. Создаем экземпляр функции "обратного вызова". В данном случае это наблюдатель.
    2. Регистрируем его, используя доступный API.
    3. Прослушиваем событие закрытия, используя awaitClose, и предоставляем
       соответствующее действие, которое нужно предпринять в данном случае.
       Скорее всего, нам придется отменить регистрацию функции.
    */
}
```

Стоит взглянуть на ее сигнатуру:

```
public inline fun <T> callbackFlow(
    @BuilderInference noinline block: suspend ProducerScope<T>.() -> Unit
): Flow<T>
```

Не обольщайтесь. Здесь есть один момент, который заключается в том, что `callbackFlow` принимает функцию, поддерживающую возможность приостановки, с приемником `ProducerScope` в качестве аргумента. Это означает, что в фигурных скобках блока, следующего за `callbackFlow`, у нас есть экземпляр `ProducerScope` в виде неявного `this`.

Вот сигнатура `ProducerScope`:

```
public interface ProducerScope<in E> : CoroutineScope, SendChannel<E>
```

Итак, `ProducerScope` — это канал `SendChannel`. И вот о чем следует помнить: `callbackFlow` предоставляет экземпляр `SendChannel`, который можно использовать в своей реализации. Экземпляры объекта, полученные из функции обратного вызова, отправляются в этот канал. Это то, что мы делаем на этапе 1 в примере 10.2.

Вариант 2: параллельное преобразование потока значений

Иногда нужно применить преобразование к коллекции или потоку объектов, чтобы получить новую коллекцию преобразованных объектов. Когда эти преобразования должны выполняться асинхронно, все становится немного сложнее. Но не с потоками!

Представьте, что у нас есть список экземпляров `Location`. Каждое местоположение можно преобразовать в экземпляр `Content` с помощью функции `transform`:

```
suspend fun transform(loc: Location): Content = withContext(Dispatchers.IO)
{
    // Фактическая реализация не имеет значения
}
```

Итак, мы получаем экземпляры `Location` и должны преобразовывать их на лету с помощью функции `transform`. Однако обработка одного экземпляра может занять некоторое время. Таким образом, мы не хотим, чтобы обработка местоположения задерживала преобразование следующих входящих местоположений. Другими словами, преобразования должны выполняться *параллельно* (рис. 10.3).

В предыдущей схеме мы ограничили параллелизм четырьмя местоположениями; другими словами, в большинстве случаев четыре местоположения можно преобразовать одновременно в заданный момент времени.

На рис. 10.4 показано, как реализовать это поведение с помощью потоков.

Соответствующий исходный код можно найти на сайте GitHub[4].

[4] См. **https://oreil.ly/LhW77**.

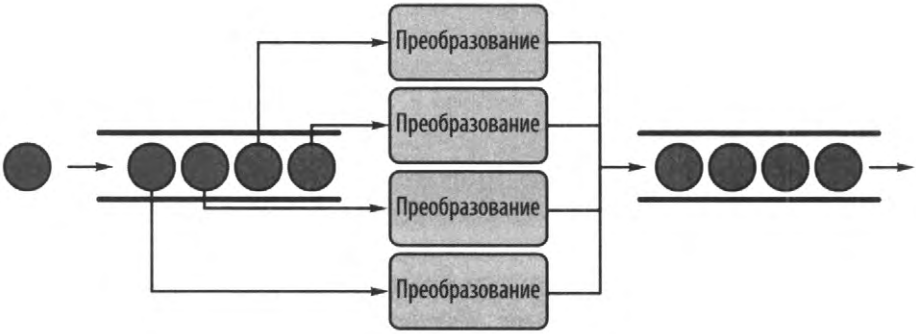


Рис. 10.3. Слияние потоков

```
18 ▶ fun main() = runBlocking { this: CoroutineScope
19       // Defining the Flow of Content - nothing is executing yet
20       val contentFlow : Flow<Content>  = locationsFlow.map { loc : Location  ->
21           flow { this: FlowCollector<Content>
22               emit(transform(loc))
23           }
24       }.flattenMerge( concurrency: 4)
25
26       // We now collect the entire flow using the toList terminal operator
27       val contents : List<Content>  = contentFlow.toList()
28   }
```

Рис. 10.4. Реализация слияния с помощью потоков

Чтобы понять, что здесь происходит, нужно усвоить, что `location.map{..}` возвращает поток потока (например, здесь тип — это `Flow<Flow<Content>>`). В операторе `map{..}` новый поток создается после выдачи местоположения вышестоящим потоком (`LocationFlow`). Каждый из этих созданных потоков имеет тип `Flow<Content>` и выполняет преобразование местоположения по отдельности.

Последний оператор, `flattenMerge`, объединяет все созданные потоки в новом потоке `Flow<Content>` (который мы назначили `contentFlow`). Кроме того, у этого оператора есть параметр `concurrency`. Наверное, было бы неуместно одновременно создавать и собирать поток каждый раз, когда мы получаем местоположение. В строке 24 (`concurrency: 4`) мы гарантируем, что на данный момент времени будет собрано не более четырех потоков. Это удобно в случае счетных задач, когда вы знаете, что ваш процессор не сможет преобразовать более четырех местоположений *параллельно* (при условии, что у него четыре ядра). Другими словами, уровень параллелизма оператора `flattenMerge` обозначает, сколько операций или преобразований будет выполняться параллельно в данный момент времени.

Благодаря природе потоков, которая поддерживает возможность приостановки, вы получаете *противодавление* бесплатно. Новые местоположения собираются из `locationsFlow` только тогда, когда доступно оборудование для их обработки. Подобный механизм можно реализовать без потоков или сопрограмм, используя пул по-

токов и блокирующую очередь. Однако для этого потребуется значительно больше кода.

На момент написания этих строк оператор `flattenMerge` помечен в исходном коде как `@Flow Preview`. Это означает, что данное объявление находится в состоянии предварительного просмотра и его можно изменить обратнонесовместимым образом с помощью миграции с максимальной эффективностью.

Мы надеемся, что к тому времени, когда мы закончим писать эту книгу, API слияния потоков будет стабилизирован. В противном случае аналогичный оператор мог бы заменить `flattenMerge`.

Что происходит в случае ошибки?

Если одна из функций `transform` вызовет исключение, весь поток будет отменен, и исключение будет передаваться подчиненным потокам. Это хорошее поведение по умолчанию, но у вас может возникнуть желание обрабатывать некоторые исключения непосредственно в самом потоке.

Мы покажем, как это сделать, в *разд. "Обработка ошибок" далее в этой главе.*

Размышления напоследок

- Вы понимаете, что мы только что создали рабочий пул, который параллельно преобразует входящий поток объектов, используя всего пять строк кода?
- Механизм управления потоками гарантированно является потокобезопасным. Больше никакой головной боли, связанной с определением правильной стратегии синхронизации для передачи ссылок на объекты из пула потоков в собирающий поток.
- Можно легко настроить уровень параллелизма, что в данном случае означает максимальное количество параллельных преобразований.

Вариант 3: создание собственного оператора

Даже если многие операторы потоков доступны "из коробки", иногда вам придется создавать свои. К счастью, потоки можно компоновать, а реализовать собственную реактивную логику не так сложно.

Например, на момент написания этих строк у оператора `Flow` нет эквивалента оператора `bufferTimeout` от Project Reactor[5].

Итак, что должен делать `bufferTimeout`? Представьте, что у вас есть вышестоящий поток элементов, но вы хотите обрабатывать эти элементы партиями с фиксиро-

[5] См. **https://oreil.ly/udGs0**.

ванной максимальной скоростью. Поток, возвращаемый bufferTimeout, должен буферизовать элементы и выдавать список (пакет) элементов, если:

- буфер переполнен;
- прошло предопределенное максимальное количество времени.

Прежде чем перейти к реализации, поговорим о ключевой идее. Поток, возвращаемый оператором bufferTimeout, "под капотом" должен потреблять вышестоящий поток и элементы буфера. Когда буфер заполнен или время ожидания истекло, поток должен передать содержимое буфера (список). Можно представить, что мы запустим сопрограмму, которая получает два типа событий.

- "Элемент только что получен из вышестоящего потока. Должны ли мы просто добавить его в буфер или также отправить весь буфер?"
- "Тайм-аут! Отправьте содержимое буфера прямо сейчас".

В *главе 9 (см. разд. "Взаимоблокировки в CSP")* мы обсуждали аналогичную ситуацию. Выражение select идеально подходит для работы с несколькими событиями, поступающими из нескольких каналов.

Теперь мы реализуем оператор bufferTimeout (рис. 10.5).

Соответствующий исходный код можно найти на сайте GitHub[6].

Поясним код.

- Во-первых, сигнатура оператора говорит о многом. Он объявлен как функция-расширение Flow<T>, поэтому его можно использовать следующим образом: upstreamFlow.bufferTime out(10, 100). Что касается типа возвращаемого значения, то это Flow<List<T>>. Помните, нам нужно обрабатывать элементы пакетами, поэтому поток, возвращаемый оператором bufferTimeout, должен возвращать элементы в виде List<T>.
- Строка 17: мы используем функцию flow{}. Напоминаем, что она предоставляет нам экземпляр FlowCollector, а блок кода — это функция-расширение с FlowCollector в качестве типа получателя. Другими словами, функцию emit можно вызвать из блока кода.
- Строка 21: мы используем coroutineScope{}, потому что будем запускать новые сопрограммы, что возможно только в CoroutineScope.
- Строка 22: с точки зрения сопрограммы[7], полученные элементы должны поступать из ReceiveChannel. Таким образом, нужно запустить еще одну внутреннюю сопрограмму, которая потребляет вышестоящий поток и отправляет его по каналу. Именно в этом и заключается цель оператора productIn.
- Строка 23: нужно сгенерировать события "тайм-аута". Именно для этой цели уже существует библиотечная функция ticker. Она создает канал, который про-

[6] См. **https://oreil.ly/JxkZj**.

[7] Сопрограммы, запущенной функцией coroutineScope{}.

изводит первый элемент после заданной начальной задержки и последующие элементы с заданной задержкой между ними. Как указано в документации, эта функция немедленно запускает новую сопрограмму, и вся ответственность за ее отмену лежит на нас.

```
/**
 * Buffers the upstream flow producing lists of elements when:
 * * A number of [maxSize] elements have been emitted
 * * A timeout of [maxDelayMillis] has expired
 *
 * Consequently, the produced lists of elements have a maximum size of [maxSize].
 */
fun <T> Flow<T>.bufferTimeout(maxSize: Int, maxDelayMillis: Long): Flow<List<T>> = flow {
    require( value: maxSize > 0) { "maxSize should be greater than 0" }
    require( value: maxDelayMillis > 0) { "maxDelayMillis should be greater than 0" }

    coroutineScope {  this: CoroutineScope
        val channel : ReceiveChannel<T>  = produceIn( scope: this)
        val ticker : ReceiveChannel<Unit>  = ticker(maxDelayMillis)
        val buffer : MutableList<T>  = mutableListOf<T>()

        suspend fun emitBuffer() {
            if (buffer.isNotEmpty()) {
                emit(buffer.toList())
                buffer.clear()
            }
        }

        try {
            whileSelect {  this: SelectBuilder<Boolean>
                channel.onReceive { value : T  ->
                    buffer.add(value)
                    if (buffer.size >= maxSize) emitBuffer()
                    true  ^onReceive
                }
                ticker.onReceive {
                    emitBuffer()
                    true  ^onReceive
                }
            }
        } catch (e: ClosedReceiveChannelException) {
            emitBuffer()
        } finally {
            channel.cancel()
            ticker.cancel()
        }
    }
}
```

Рис. 10.5. Реализация оператора `bufferTimeout`

- Строка 34: используем функцию `whileSelect`. На самом деле это просто синтаксический сахар для циклов в выражении `select`, в то время как мы видим, что возвращается значение `true`. В блоке `whileSelect{}` можно увидеть логику добавления элемента в буфер только в том случае, если он не переполнен, или в противном случае выпуск всего буфера.
- Строка 46: когда сбор вышестоящего потока завершается, сопрограмма, запущенная с помощью оператора productIn, по-прежнему будет пытаться читать данные из этого потока и будет вызвано исключение `ClosedReceiveChannelException`. Поэтому мы перехватываем его и знаем, что должны эмитировать содержимое буфера.
- Строки 48 и 49: каналы — это горячие сущности, их следует отменить, если они больше не должны использоваться. Что касается функции `ticker`, то ее тоже следует отменить.

Использование

На рис. 10.6 показан пример использования оператора `bufferTimeout`.

```
suspend fun main() {
    val flow = (1..100).asFlow().onEach { delay( timeMillis: 10) }
    val startTime = System.currentTimeMillis()
    flow.bufferTimeout( maxSize: 10,  maxDelayMillis: 50).collect {  it: List<Int>
        val time = System.currentTimeMillis() - startTime
        println("$time ms: $it")
    }
}
```

Рис. 10.6. Использование оператора `bufferTimeout`

Соответствующий исходный код можно найти на сайте GitHub[8].

Вывод:

```
139 ms: [1, 2, 3, 4]
172 ms: [5, 6, 7, 8]
223 ms: [9, 10, 11, 12, 13]
272 ms: [14, 15, 16, 17]
322 ms: [18, 19, 20, 21, 22]
...
1022 ms: [86, 87, 88, 89, 90]
1072 ms: [91, 92, 93, 94, 95]
1117 ms: [96, 97, 98, 99, 100]
```

[8] См. **https://oreil.ly/Y2xVe**.

Видно, что вышестоящий поток выдает числа от 1 до 100 с задержкой 10 мс между каждой выдачей. Мы задаем для времени ожидания значение 50 мс, и каждый эмитируемый список может содержать не более пяти чисел.

Обработка ошибок

Обработка ошибок — фундаментальный момент в реактивном программировании. Если вы знакомы с RxJava, то, вероятно, обрабатываете исключения, используя функцию обратного вызова `onError` метода `subscribe`:

```
// Пример из RxJava
someObservable().subscribe(
    { value -> /* Делаем что-то полезное */ },
    { error -> println("Error: $error") }
)
```

Используя потоки, можно обрабатывать ошибки, применяя комбинацию методов, среди которых:

- классический блок `try/catch`;
- оператор `catch` — мы рассмотрим его сразу после обсуждения блока `try/catch`.

Блок *try/catch*

Если определить фиктивный вышестоящий поток, содержащий только три целых числа, и намеренно выбросить исключение в блоке `collect{}`, то можно перехватить его, заключив всю цепочку в блок `try/catch` (рис. 10.7).

```
val upstream : Flow<Int> = flowOf( ...elements: 1, 2, 3)

fun main() : Unit = runBlocking {  this: CoroutineScope
    try {
        upstream.collect { value : Int ->
            if (value > 2) {
                throw RuntimeException()
            }
            println("Received $value")
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

Рис. 10.7. Блок `try/catch`

Соответствующий исходный код можно найти на сайте GitHub[9].

Вывод:

```
Received 1
Received 2
Caught java.lang.RuntimeException
```

Важно отметить, что блок `try/catch` также работает, когда исключение возбуждается в вышестоящем потоке. Например, мы получим точно такой же результат, если заменим определение вышестоящего потока (рис. 10.8).

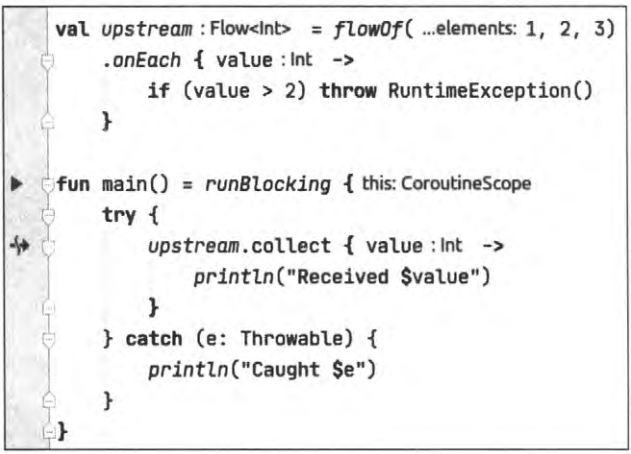


Рис. 10.8. Измененное определение вышестоящего потока

Соответствующий исходный код можно найти на сайте GitHub[10].

Однако, если вы попытаетесь перехватить исключение в самом потоке, то, скорее всего, получите неожиданные результаты. Например:

```
// Предупреждение: НЕ ДЕЛАЙТЕ ЭТОГО, этот поток поглощает подчиненные исключения
val upstream: Flow<Int> = flow {
    for (i in 1..3) {
        try {
            emit(i)
        } catch (e: Throwable) {
            println("Intercept downstream exception $e")
        }
    }
}
```

[9] См. **https://oreil.ly/qcOKV**.

[10] См. **https://oreil.ly/lrrGt**.

```
fun main() = runBlocking {
    try {
        upstream.collect { value ->
            println("Received $value")
            check(value <= 2) {
                "Collected $value while we expect values below 2"
            }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

В этом примере мы используем функцию flow для определения переменной upstream, а также обернули вызов emit оператором try/catch. Даже если это кажется бесполезным, потому что данная функция не генерирует исключений, тем не менее, возможно, это не лишено смысла, если речь идет о нетривиальной логике эмиссии. В функции main мы собираем этот поток и проверяем, что не получаем значений, строго превышающих 2. В противном случае блок catch должен вывести фразу "Caught java.lang.IllegalStateException Collected x while we expect values below 2".

Мы ожидаем увидеть следующий вывод:

```
Received 1
Received 2
Caught java.lang.IllegalStateException: Collected 3 while we expect values
below 2
```

Однако на самом деле получаем это:

```
Received 1
Received 2
Received 3
Intercept downstream exception java.lang.IllegalStateException: Collected 3
while we expect values below 2
```

Несмотря на исключение, возбужденное check(value <= 2) {..}, это исключение перехватывается не оператором try/catch из функции main, а оператором потока try/catch.

Оператор try/catch из функции flow может перехватывать *подчиненные* исключения, в том числе исключения, возбуждаемые во время сбора потока.

Разделение ответственности важно

Реализация потока не должна оказывать побочного эффекта на код, собирающий этот поток. Равно как и код, который собирает поток, не должен знать о деталях реализации вышестоящего потока. Поток всегда должен быть *прозрачным для исключений*: он должен передавать исключения, поступающие от сборщика. Другими словами, поток никогда не должен поглощать подчиненные исключения.

В этой книге мы будем использовать понятие *"прозрачность исключений"* для обозначения потока, *прозрачного для исключений*.

Нарушение прозрачности исключения

Предыдущий пример — это пример нарушения прозрачности исключений. Попытка выдачи значений из блока try/catch — еще одно нарушение. Например (и снова просим вас не делать так):

```
val violatesExceptionTransparency: Flow<Int> = flow {
    for (i in 1..3) {
        try {
            emit(i)
        } catch (e: Throwable) {
            emit(-1)
        }
    }
}

fun main() = runBlocking {
    try {
        violatesExceptionTransparency.collect { value ->
            check(value <= 2) { "Collected $value" }
        }
    } catch (e: Throwable) {
        println("Caught $e")
    }
}
```

Вывод:

```
Caught java.lang.IllegalStateException: Flow exception transparency is
violated:
Previous 'emit' call has thrown exception java.lang.IllegalStateException:
Collected 3, but then emission attempt of value '-1' has been detected.
Emissions from 'catch' blocks are prohibited in order to avoid unspecified
behaviour, 'Flow.catch' operator can be used instead.
For a more detailed explanation, please refer to Flow documentation.
```

Блок try/catch следует использовать *только* для того, чтобы поместить туда сборщика, для обработки исключений, возбуждаемых самим сборщиком, либо (возможно, хотя это и не идеальный вариант) для обработки исключений, возбуждаемых из потока.

Для обработки исключений в потоке следует использовать оператор catch.

Оператор *catch*

Оператор catch допускает декларативный стиль перехвата исключений, как показано на рис. 10.9. Он перехватывает все исключения вышестоящего потока. Говоря *все исключения*, мы подразумеваем, что он перехватывает даже экземпляры класса Throwable. Поскольку данный оператор перехватывает только исключения вышестоящего потока, у него нет проблемы с исключениями, присущей блоку try/catch.

```
val upstream : Flow<Int>  = flowOf( ...elements: 1, 2, 3)

val encapsulateError : Flow<Int>  = upstream
    .onEach { it: Int
        if (it > 2) throw RuntimeException()
    }
    .catch { e : Throwable  ->
        println("Caught $e")
    }

fun main() = runBlocking { this: CoroutineScope
    encapsulateError.collect { it: Int
        println("Received $it")
    }
}
```

Рис. 10.9. Декларативный стиль

Соответствующий исходный код можно найти на сайте GitHub[11].

Вывод:

```
Received 1
Received 2
Caught java.lang.RuntimeException
```

Поток возбуждает исключение RuntimeException, если ему передано значение больше 2. Сразу после этого, в операторе catch, идет вывод в консоль. Однако сборщик никогда не получит значение 3. Таким образом, оператор catch автоматически отменяет поток.

[11] См. **https://oreil.ly/QcUeq**.

Прозрачность исключений

Из этого оператора можно перехватывать только *вышестоящие исключения*. Говоря *"вышестоящие"*, мы имеем в виду относительно оператора `catch`. Чтобы пояснить, мы возьмем пример, в котором сборщик генерирует исключение до того, как "под капотом" поток выбросит еще одно исключение. Сборщик должен иметь возможность перехватывать возбужденное исключение (исключение не должно быть перехвачено потоком) (рис. 10.10).

```
val upstream : Flow<Int>  = flowOf( ...elements: 1, 3, -1)

val encapsulateError : Flow<Int>  = upstream
    .onEach { it: Int
        if (it < 0) throw NumberFormatException("Values should be greater than 0")
    }
    .catch { e : Throwable  ->
        println("Caught $e")
    }

fun main() = runBlocking { this: CoroutineScope
    try {
        encapsulateError.collect { it: Int
            if (it > 2) throw RuntimeException()
            println("Received $it")
        }
    } catch (e: RuntimeException) {
        println("Collector stopped collecting the flow")
    }
}
```

Рис. 10.10. Перехват возбужденного исключения

Соответствующий исходный код можно найти на сайте GitHub[12].

В этом примере сборщик выбрасывает исключение `RuntimeException`, если собирает значение, которое больше 2. Логика сбора заключена в операторе `try/catch`, потому что мы не хотим, чтобы программа аварийно завершила работу и зарегистрировала исключение. Поток возбуждает исключение `NumberformatException`, если значение отрицательное. Оператор `catch` выполняет роль охранника (регистрирует исключение и отменяет поток).

Вывод:

```
Received 0
Collector stopped collecting the flow
```

Обратите внимание, что поток не перехватил исключение, сгенерированное в сборщике, потому что оно было перехвачено в предложении `catch` оператора `try/catch`.

[12] См. **https://oreil.ly/0U5h1**.

Поток так и не возбудил исключение NumberformatException, потому что сборщик преждевременно отменил сбор.

Еще один пример

В *разд. "Вариант 2: параллельное преобразование потока значений" ранее в этой главе* мы воздержались от разговора об обработке ошибок. Предположим, что функция transform может возбуждать исключения, среди которых NumberFormatException. Можно выборочно обработать его с помощью оператора catch:

```
fun main() = runBlocking {
    // Определение потока контента - пока ничего не выполняется
    val contentFlow = locationsFlow.map { loc ->
        flow {
            emit(transform(loc))
        }.catch { cause: Throwable ->
            if (cause is NumberFormatException) {      ❶
                println("Handling $cause")
            } else {
                throw cause                            ❷
            }
        }
    }.flattenMerge(4)

    // Теперь собираем весь поток с помощью терминального оператора toList
    val contents = contentFlow.toList()
}
```

❶ Поскольку оператор catch перехватывает экземпляры класса Throwable, нужно проверить тип ошибки. Если ошибка представляет собой исключение NumberFormatException, мы обрабатываем ее в операторе if. Можно добавить туда другие проверки для различных типов ошибок.

❷ В противном случае тип ошибки неизвестен. В большинстве случаев предпочтительнее не проглатывать ошибку и повторно выбросить исключение.

Использование функции *emit* из оператора *catch*

Иногда имеет смысл эмитировать определенное значение при перехвате исключения из потока (рис. 10.11).

Соответствующий исходный код можно найти на сайте GitHub[13].

[13] См. **https://oreil.ly/vknEm**.

```
val upstream : Flow<Int>  = flowOf( ...elements: 1, 3, -1)

val encapsulateError : Flow<Int>  = upstream
    .onEach { it: Int
        if (it < 0) throw NumberFormatException("Values should be greater than 0")
    }
    .catch { e : Throwable  ->
        emit( value: 0)
    }

fun main() = runBlocking { this: CoroutineScope
    encapsulateError.collect { it: Int
        println("Received $it")
    }
}
```

Рис. 10.11. Эмитирование значения

Вывод:

```
Received 1
Received 3
Received 0
```

Выдача значений из оператора `catch` особенно полезна при *материализации исключений*.

Материализация исключений

Материализация исключений[14] — это процесс перехвата исключений и эмиссии специальных значений или объектов, представляющих эти исключения. Цель состоит в том, чтобы избежать генерации исключений в потоке, потому что тогда выполнение кода переходит в любое место, где собирается этот поток. Неважно, обрабатывает ли код сбора исключения, генерируемые потоком, или нет. Если поток выбрасывает исключения, этот код должен знать о них и перехватывать их, чтобы избежать неопределенного поведения. Таким образом, поток оказывает *побочное действие на код сбора*, что является нарушением принципа прозрачности исключений.

Код сбора не должен знать подробности реализации потока. Например, если поток — это `Flow<Number>`, вы должны ожидать получения только значений типа `Number` (или подтипов), но не исключений.

---

[14] Слово *"материализация"* происходит от названия оператора RxJava `materialize`. См. документацию RxJava для получения дополнительной информации (**https://oreil.ly/SEiRP**).

Возьмем другой пример. Представьте, что вы получаете изображения по URL-адресам. У вас есть входящий поток URL-адресов:

```
// Мы не используем реалистичные URL-адреса для краткости
val urlFlow = flowOf("url-1", "url-2", "url-retry")
```

Также у вас есть эта функция:

```
suspend fun fetchImage(url: String): Image {
    // Имитация удаленного вызова
    delay(10)

    // Имитация исключения, сгенерированного сервером или API
    if (url.contains("retry")) {
        throw IOException("Server returned HTTP response code 503")
    }

    return Image(url)
}

data class Image(val url: String)
```

Функция `fetchImage` может генерировать исключения `IOException`. Чтобы создать "поток изображений", используя переменную `urlFlow` и функцию `fetchImage`, эти исключения нужно материализовать. Что касается функции `fetchImage`, она либо завершится успешно, либо выдаст ошибку — либо вы получаете экземпляр `Image`, либо генерируется исключение. Эти результаты можно представить типом `Result` с подклассами `Success` и `Error`[15]:

```
sealed class Result
data class Success(val image: Image) : Result()
data class Error(val url: String) : Result()
```

В случае успеха мы оборачиваем фактический результат — экземпляр `Image`. В случае ошибки мы сочли уместным обернуть URL-адрес, для которого не удалось получить изображение. Однако вы можете обернуть все данные, которые могут быть полезны для кода сбора, например само исключение.

Теперь можно инкапсулировать использование `fetchImage`, создав функцию `fetchResult`, которая возвращает экземпляры `Result`:

```
suspend fun fetchResult(url: String): Result {
    println("Fetching $url..")
    return try {
        val image = fetchImage(url)
        Success(image)
```

[15] Эти подклассы представляют собой алгебраический тип данных.

```
    } catch (e: IOException) {
        Error(url)
    }
}
```

Наконец, можно реализовать переменную resultFlow и безопасно собрать ее:

```
fun main() = runBlocking {
    val urlFlow = flowOf("url-1", "url-2", "url-retry")

    val resultFlow = urlFlow
        .map { url -> fetchResult(url) }

    val results = resultFlow.toList()
    println("Results: $results")
}
```

Вывод:

```
Fetching url-1..
Fetching url-2..
Fetching url-retry..
Results: [Success(image=Image(url=url-1)), Success(image=Image(url=url-2)),
Error(url=url-retry)]
```

Бонус

Представьте, что вы хотите автоматически повторить попытку получения изображения в случае ошибки. Можно реализовать собственный оператор, который повторяет action, пока predicate возвращает true:

```
fun <T, R : Any> Flow<T>.mapWithRetry(
    action: suspend(T) -> R,
    predicate: suspend(R, attempt: Int) -> Boolean
) = map { data ->
    var attempt = 0L
    var shallRetry: Boolean
    var lastValue: R? = null
    do {
        val tr = action(data)
        shallRetry = predicate(tr, ++attempt)
        if (!shallRetry) lastValue = tr
    } while (shallRetry)
    return@map lastValue
}
```

Если вы хотите повторить попытку три раза (максимум) перед возвратом ошибки, можно использовать этот оператор следующим образом:

```
fun main() = runBlocking {
    val urlFlow = flowOf("url-1", "url-2", "url-retry")

    val resultFlowWithRetry = urlFlow
        .mapWithRetry(
            { url -> fetchResult(url) },
            { value, attempt -> value is Error && attempt < 3L }
        )

    val results = resultFlowWithRetry.toList()
    println("Results: $results")
}
```

Вывод:

```
Fetching url-1..
Fetching url-2..
Fetching url-retry..
Fetching url-retry..
Fetching url-retry..
Results: [Success(image=Image(url=url-1)), Success(image=Image(url=url-2)),
Error(url=url-retry)]
```

Горячие потоки и *SharedFlow*

Предыдущие реализации потока были *холодными*: ничего не выполняется, пока вы не начнете собирать поток. Это стало возможным, потому что, если говорить о каждом эмитируемом значении, то только один сборщик получит это значение. Поэтому нет необходимости что-либо запускать, пока сборщик не будет готов собирать значения.

Но что делать, если нужно *поделить* выдаваемые значения между несколькими сборщиками? Например, в приложении завершается такое событие, как скачивание файла. Возможно, вы захотите напрямую уведомить различные компоненты, скажем, компонент ViewModel, репозитории или даже некоторые представления. Загрузчик файлов может не знать о существовании других частей приложения. Правильное разделение ответственности начинается со слабой связанности классов, и *шина событий* — один из архитектурных паттернов, который помогает в данной ситуации.

Принцип здесь простой: загрузчик выдает событие (экземпляр класса, который, возможно, содержит некое состояние), передавая его шине событий, и все подпис-

чики впоследствии получат его. Точно так же может действовать и поток SharedFlow (рис. 10.12).

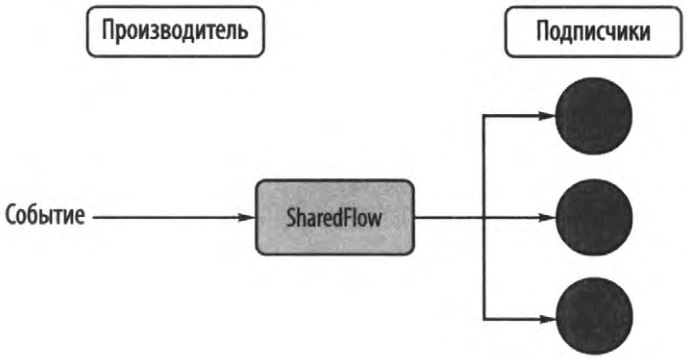


Рис. 10.12. SharedFlow

SharedFlow рассылает события всем своим подписчикам. На самом деле это набор инструментов, который можно использовать во многих ситуациях, а не только для реализации шины событий. Прежде чем привести примеры использования, мы покажем, как создать SharedFlow и настроить его.

Создаем *SharedFlow*

В простейшем случае мы вызываем функцию MutableSharedFlow() без параметров. Как следует из названия, ее состояние можно *изменять*, отправляя ей значения. При создании SharedFlow существует распространенный шаблон, который заключается в создании приватной изменяемой и общедоступной неизменяемой версий с использованием функции asSharedFlow(), как показано ниже:

```
private val _sharedFlow = MutableSharedFlow<Data>()
val sharedFlow: SharedFlow<Data> = _sharedFlow.asSharedFlow()
```

Это полезный шаблон, если вы гарантируете, что подписчики смогут только *читать значения из* потока (например, они не смогут отправлять их). Возможно, вы удивитесь, обнаружив, что MutableSharedFlow — это не класс. На самом деле это функция, принимающая параметры, о которых мы поговорим позже. На данный момент мы показываем только версию MutableSharedFlow по умолчанию, без аргументов.

Регистрируем подписчика

Подписчик регистрируется, когда приступает к сбору SharedFlow. Желательно, чтобы это была общедоступная неизменяемая версия:

```
scope.launch {
    sharedFlow.collect { data ->
```

```
            println(data)
        }
    }
```

Подписчик может находиться только в области видимости, потому что терминальный оператор `collect` представляет собой функцию, поддерживающую возможность приостановки. Это хорошо для структурированного параллелизма: если область видимости отменена, отменяется и подписчик.

Отправляем значения в *SharedFlow*

`MutableSharedFlow` предоставляет два метода для выдачи значений — `emit` и `tryEmit`.

`emit`

Приостанавливается при некоторых условиях (мы обсудим их в ближайшее время).

`tryEmit`

Никогда не приостанавливается и пытается немедленно эмитировать значение.

Почему метода два? Это связано с тем, что по умолчанию, когда `MutableSharedFlow` эмитирует значение с помощью метода `emit`, он приостанавливается до тех пор, пока *все* подписчики не приступят к обработке значения. Мы приведем пример использования метода `emit` в следующем разделе.

Однако иногда это не то, что вам нужно. Вы будете сталкиваться с ситуациями, когда нужно выдавать значения из кода, который не поддерживает возможность приостановки (*см. разд. "Использование SharedFlow в качестве шины событий" далее в этой главе*). Поэтому здесь вступает в дело метод `tryEmit`, который пытается немедленно эмитировать значение и возвращает `true`, если все удалось, или в противном случае — `false`. Мы подробнее рассмотрим нюансы работы методов `emit` и `tryEmit` в последующих разделах.

Использование *SharedFlow* для потоковой передачи данных

Предположим, вы разрабатываете новостное приложение. Одна из его особенностей состоит в том, что оно извлекает новости из API или локальной базы данных и отображает их (или новостную ленту). В идеале нужно полагаться на локальную базу данных, чтобы по возможности избегать использования API. Мы будем использовать API как единственный источник новостей, хотя можно легко расширить этот пример, чтобы добавить долговременное хранение локальных данных.

Архитектура

В нашей архитектуре компонент ViewModel использует репозиторий для получения ленты новостей, после которого он уведомляет представление. Репозиторий

отвечает за запрос к удаленному API через регулярные промежутки времени и предоставляет средство компонентам ViewModel для получения новостной ленты (рис. 10.13).

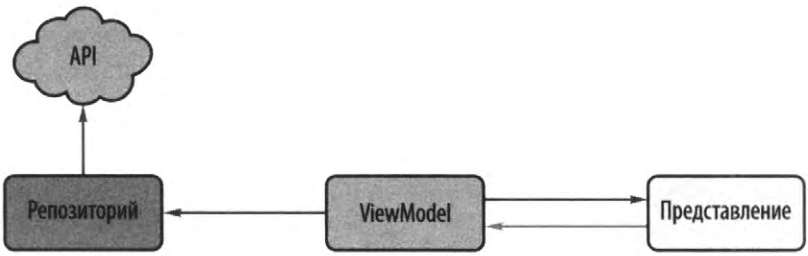


Рис. 10.13. Архитектура приложения

Реализация

Чтобы было проще, следующий класс данных `News` обозначает новости:

```
data class News(val content: String)
```

Репозиторий обращается к API через интерфейс `NewsDao`. В нашем примере объект доступа к данным (data access object, DAO) внедряется через конструктор вручную. В реальном приложении мы рекомендуем использовать фреймворк внедрения зависимостей (dependency injection, DI), например Hilt или Dagger:

```
interface NewsDao {
    suspend fun fetchNewsFromApi(): List<News>
}
```

Теперь у нас достаточно материала для реализации репозитория:

```
class NewsRepository(private val dao: NewsDao) {
    private val _newsFeed = MutableSharedFlow<News>()                    ❶
    val newsFeed = _newsFeed.asSharedFlow()                              ❷

    private val scope = CoroutineScope(Job() + Dispatchers.IO)

    init {
        scope.launch {                                                   ❸
            while (true) {
                val news = dao.fetchNewsFromApi()
                news.forEach { _newsFeed.emit(it) }                      ❹

                delay(3000)
            }
        }
    }
```

```
    fun stop() = scope.cancel()
}
```

❶ Создаем приватный изменяемый поток. Он будет использоваться только в репозитории.

❷ Создаем общедоступную неизменяемую версию потока.

❸ Как только экземпляр репозитория создан, начинаем получать новости из API.

❹ Каждый раз при получении списка экземпляров `News` мы эмитируем эти значения, используя `MutableSharedFlow`.

Осталось только реализовать компонент ViewModel, который будет подписываться на поток репозитория:

```
class NewsViewsModel(private val repository: NewsRepository) : ViewModel() {
    private val newsList = mutableListOf<News>()

    private val _newsLiveData = MutableLiveData<List<News>>(newsList)
    val newsLiveData: LiveData<List<News>> = _newsLiveData

    init {
        viewModelScope.launch {
            repository.newsFeed.collect {
                println("NewsViewsModel receives $it")
                newsList.add(it)
                _newsLiveData.value = newsList
            }
        }
    }
}
```

Вызывая `relay.newsFeed.collect { .. }`, компонент ViewModel подписывается на разделяемый поток. Каждый раз, когда репозиторий отправляет экземпляр `News` в разделяемый поток, компонент ViewModel получает новости и добавляет их в `LiveData`, чтобы обновить представление.

Обратите внимание, что сбор потоков происходит в сопрограмме, запускаемой с помощью `viewModel Scope.launch`. Это означает, что, если срок существования компонента ViewModel подходит к концу, сбор потоков будет автоматически отменен, и это хорошо.

В нашем примере мы вручную внедряем объект (в нашем случае — репозиторий) через конструктор. DI-фреймворк, определенно, поможет избежать шаблонного кода. Поскольку демонстрация DI-фреймворков не является основной темой этой главы, мы решили внедрить репозиторий в компонент ViewModel вручную.

Тест реализации

Чтобы протестировать предыдущий код, нужно смоделировать NewsDao. DAO просто отправит два фиктивных экземпляра News и увеличит счетчик:

```
val dao = object : NewsDao {
    private var index = 0

    override suspend fun fetchNewsFromApi(): List<News> {
        delay(100) // имитация сетевой задержки
        return listOf(
            News("news content ${++index}"),
            News("news content ${++index}")
        )
    }
}
```

При выполнении предыдущего кода в консоли мы видим следующее:

```
NewsViewsModel receives News(content=news content 1)
NewsViewsModel receives News(content=news content 2)
NewsViewsModel receives News(content=news content 3)
...
```

Здесь нет ничего удивительного: компонент ViewModel просто получает новости, присланные репозиторием. Все становится куда интереснее, если у нас не один, а несколько компонентов ViewModel, которые подписываются на разделяемый поток. Мы пошли дальше и создали еще один компонент ViewModel, который также пишет журналы в консоль. Через 250 мс *после* запуска программы мы создали еще один компонент ViewModel. Вот вывод, который мы получили:

```
NewsViewsModel receives News(content=news content 1)
NewsViewsModel receives News(content=news content 2)
NewsViewsModel receives News(content=news content 3)
AnotherViewModel receives News(content=news content 3)
NewsViewsModel receives News(content=news content 4)
AnotherViewModel receives News(content=news content 4)
NewsViewsModel receives News(content=news content 5)
AnotherViewModel receives News(content=news content 5)
NewsViewsModel receives News(content=news content 6)
AnotherViewModel receives News(content=news content 6)
...
```

Видно, что другой компонент ViewModel *пропустил* первые две новости. Это связано с тем, что в то время, когда разделяемый поток эмитирует первые две записи, первый компонент ViewModel является единственным подписчиком. Второй компонент ViewModel идет следом и получает только более поздние новости.

Воспроизведение значений

Что делать, если нам нужен второй компонент ViewModel, чтобы получить предыдущие новости? Разделяемый поток может *опционально* кэшировать значения, чтобы новые подписчики получили последние *n* кэшированных значений. В нашем случае, если мы хотим, чтобы разделяемый поток воспроизвел две последние записи, все, что нужно сделать, — это обновить строку в репозитории:

```
private val _newsFeed = MutableSharedFlow<News>(replay = 2)
```

После этого изменения оба компонента ViewModel получают *все* новости. Воспроизведение данных полезно и в других распространенных ситуациях. Представьте, что пользователь выходит из фрагмента, отображающего список новостей. Потенциально, ассоциированный компонент ViewModel также может быть удален, если его жизненный цикл привязан к фрагменту (или оставлен без изменений, если вы решили привязать компонент ViewModel к активности). Далее пользователь возвращается к фрагменту новостей. Что происходит тогда? Компонент ViewModel создается заново и сразу же получает две последние записи, ожидая свежих новостей. Если воспроизвести только две новости, этого может оказаться недостаточным. Поэтому можно увеличить количество повторов, скажем, до 15.

Подведем итоги. `SharedFlow` может дополнительно воспроизводить значения для новых подписчиков. Количество значений настраивается с помощью параметра `replay` функции `MutableSharedFlow`.

Приостанавливать или нет?

Есть еще одна вещь, касающаяся воспроизведения, о которой вы должны знать. Разделяемый поток с параметром `replay` больше 0 использует кэш, который работает аналогично `Channel`. Например, при создании разделяемого потока, если значение `replay` равно 3, первые три вызова метода `emit` не будут приостанавливаться. В данном случае методы `emit` и `tryEmit` делают одно и то же: добавляют новое значение в кэш (рис. 10.14).

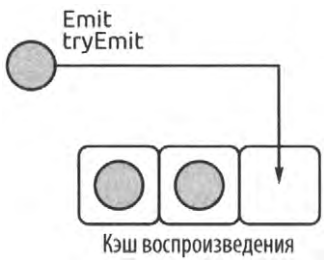


Рис. 10.14. Кэш не заполнен

При отправке четвертого значения в разделяемый поток все зависит от того, какой метод вы используете: `emit` или `tryEmit` (рис. 10.15). По умолчанию, когда кэш воспроизведения заполнен, метод `emit` приостанавливается до тех пор, пока все подписчики не начнут обрабатывать самое старое значение в кэше. Что касается мето-

да tryEmit, он возвращает false, поскольку не может добавить значение в кэш. Если вы сами не следите за четвертым значением, оно теряется.

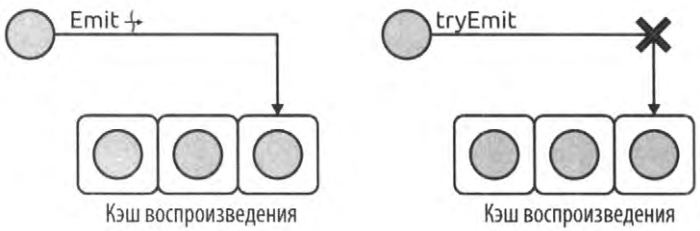


Рис. 10.15. Кэш заполнен

Такое поведение (когда кэш воспроизведения заполнен) можно изменить. Вы также можете отказаться от самого старого значения в кэше либо от значения, которое добавляется в кэш. В обоих случаях метод emit не приостанавливается, а метод tryEmit возвращает true. Следовательно, есть три возможных поведения при переполнении буфера: приостановка, удаление самого старого значения и удаление самого последнего значения.

При создании разделяемого потока применяется желаемое поведение с использованием параметра onBufferOverflow, как показано ниже.

```
MutableSharedFlow(replay = 3, onBufferOverflow = BufferOverflow.DROP_OLDEST)
```

BufferOverflow — это *перечисление* с тремя возможными значениями: SUSPEND, DROP_OLDEST и DROP_LATEST. Если не указать значение для onBufferOverflow, то значение SUSPEND будет стратегией по умолчанию.

Буферные значения

Помимо возможности воспроизведения значений, разделяемый поток может *буферизовать* значения без их повторного воспроизведения, что позволяет медленным подписчикам отставать от других, более быстрых подписчиков. Размер буфера можно настроить, как показано ниже:

```
MutableSharedFlow(extraBufferCapacity = 2)
```

По умолчанию extraBufferCapacity равен нулю. Когда вы задаете строго положительное значение, метод emit не приостанавливается, пока в буфере остается место, если только вы явно не измените стратегию переполнения буфера.

Возможно, вам интересно, в каких ситуациях может быть полезен extraBufferCapacity. Одно из непосредственных последствий создания разделяемого потока, например, когда extraBufferCapacity = 1, а onBufferOverflow = BufferOverflow.DROP_OLDEST, заключается в том, что метод tryEmit *всегда* будет успешно вставлять значение в разделяемый поток. Это гарантировано. Иногда очень удобно вставлять значения в разделяемый поток из кода, не поддерживающего возможность приостановки. Хороший пример такого варианта — использование разделяемого потока в качестве шины событий.

Использование *SharedFlow* в качестве шины событий

Шина событий нужна при наличии следующих условий:

- вам необходимо разослать событие одному или нескольким подписчикам;
- событие должно быть обработано *только один раз*;
- если компонент не зарегистрирован как подписчик, когда вы эмитируете событие, то событие для этого компонента теряется.

Обратите внимание на разницу с `LiveData`, который хранит в памяти последнее выданное значение и воспроизводит его каждый раз при повторном создании фрагмента. С шиной событий фрагмент получит событие только *один раз*. Например, если фрагмент создается заново (пользователь поворачивает устройство), то событие больше не будет обрабатываться.

Шина событий особенно полезна, если нужно, например, отобразить всплывающее сообщение, выводимое классами `Toast` или `Snackbar`. Имеет смысл отображать сообщение только один раз. Для этого репозиторий может предоставить разделяемый поток, как показано в следующем коде. Чтобы сделать открытый поток доступным для компонентов ViewModel или даже фрагментов, можно использовать фреймворк внедрения зависимостей, например Hilt или Dagger:

```
class MessageRepository {
    private val _messageFlow = MutableSharedFlow<String>(
        extraBufferCapacity = 1,
        onBufferOverflow = BufferOverflow.DROP_OLDEST
    )
    val messageEventBus = _messageFlow.asSharedFlow()

    private fun someTask() {
        // Уведомляем подписчиков для отображения сообщения
        _messageFlow.tryEmit("This is important")
    }
}
```

Мы задали для `extraBufferCapacity` значение 1, а для `onBufferOverflow` значение `DROP_OLDEST`, чтобы `_messageFlow.tryEmit` всегда успешно эмитировал значение. Почему нас волнует функция `tryEmit`? В нашем примере мы используем `_messageFlow` из функции, не поддерживающей возможность приостановки. Поэтому нельзя использовать функцию `emit` в `someTask`.

Если вы используете `_messageFlow` из сопрограммы, тогда использовать ее можно. Поведение будет точно таким же, т. к. она не будет приостанавливаться из-за присутствия буфера и политики переполнения.

Шина событий подходит для отправки разовых событий, которые некоторые компоненты могут пропустить, если они не готовы к их получению. Скажем, вы запус-

каете событие "запись остановлена", пока пользователь еще не перешел к фрагменту, отображающему записи. В результате событие теряется. Однако приложение можно спроектировать таким образом, чтобы обновлять состояние фрагмента каждый раз, когда фрагмент возобновляет работу. Следовательно, получение события "запись остановлена" полезно только тогда, когда фрагмент находится в состоянии возобновления, т. к. это должно инициировать обновление состояния. Это всего лишь пример того, когда потеря событий вполне приемлема и является частью архитектуры приложения.

Однако иногда это не то, что вам нужно. Возьмем, к примеру, службу, которая может выполнять скачивание. Если служба запускает событие "скачивание завершено", вы не хотите, чтобы пользовательский интерфейс пропустил это. Когда пользователь переходит к представлению, отображающему состояние скачивания, представление должно отображать обновленное состояние.

Вы будете сталкиваться с ситуацией, при которой необходимо совместное использование *состояния*. Она настолько распространена, что для нее была специально создана разновидность `SharedFlow`: `StateFlow`.

StateFlow: специализированная версия *SharedFlow*

При совместном использовании состояния поток состояния:

- совместно использует только одно значение: текущее *состояние*;
- воспроизводит состояние. Подписчики должны получать последнее состояние, даже если они подписываются позже;
- эмитирует начальное значение — так же, как у `LiveData`, имеется начальное значение;
- эмитирует новые значения только при изменении состояния.

Как вы узнали ранее, такого поведения можно добиться с помощью разделяемого потока:

```
val shared = MutableSharedFlow(
    replay = 1,
    onBufferOverflow = BufferOverflow.DROP_OLDEST
)
shared.tryEmit(initialValue)              // выдает начальное значение
val state = shared.distinctUntilChanged() // получаем поведение, подобное StateFlow
```

`StateFlow`[16] — это сокращенный вариант предыдущего кода. На практике все, что нужно написать, это:

```
val state = MutableStateFlow(initialValue)
```

[16] На самом деле `StateFlow` — это `SharedFlow` "под капотом".

Пример использования *StateFlow*

Представьте, что у вас есть служба скачивания, у которой может быть три возможных состояния: скачивание началось, скачивание идет и скачивание завершено (рис. 10.16).

Рис. 10.16. Состояние скачивания

Открыть доступ к потоку из службы Android можно несколькими способами. Если вам нужна высокая степень развязки, скажем, для тестирования, то это можно сделать, используя объект "репозиторий", внедренный с помощью внедрения зависимостей. После этого репозиторий внедряется во все компоненты, на которые необходимо подписаться. Или же служба может статически открыть доступ к потоку в объекте-компаньоне, что приводит к тесной связи между всеми компонентами, использующими поток. Однако это может быть приемлемо в небольшом приложении или в демонстрационных целях, как в следующем примере:

```
class DownloadService : Service() {
    companion object {
        private val _downloadState =
            MutableStateFlow<ServiceStatus>(Stopped)
        val downloadState = _downloadState.asStateFlow()
    }
    // Остальной код скрыт для краткости
}

sealed class ServiceStatus
object Started : ServiceStatus()
data class Downloading(val progress: Int) : ServiceStatus()
object Stopped : ServiceStatus()
```

"Под капотом" служба может обновить свое состояние, используя, например, `_downloadState.tryEmit (Stopped)`. При объявлении в объекте-компаньоне к потоку состояния можно легко получить доступ из компонента ViewModel.

Этот поток доступен как LiveData при использовании функции asLiveData():

```
class DownloadViewModel : ViewModel() {
    val downloadServiceStatus = DownloadService.downloadState.asLiveData()
}
```

Впоследствии представление может подписаться на LiveData:

```
class DownloadFragment : Fragment() {
    private val viewModel: DownloadViewModel by viewModels()

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

        viewModel.downloadServiceStatus.observe(this) {      ❶
            it?.also {
                onDownloadServiceStatus(it)
            }
        }
    }

    private fun onDownloadServiceStatus(
        status: ServiceStatus
    ): Nothing = when (status) {                              ❷
        Started -> TODO("Show download is about to start")
        Stopped -> TODO("Show download stopped")
        is Downloading -> TODO("Show progress")
    }
}
```

❶ Подписываемся на LiveData. Если мы не получаем пустое значение, то вызываем метод onDownloadServiceStatus.

❷ Намеренно используем выражение when, чтобы компилятор Kotlin гарантировал, что учитываются все возможные типы ServiceStatus.

Возможно, вам интересно, почему мы использовали поток состояний и вообще не использовали LiveData, устраняя необходимость в функции asLiveData() в компоненте ViewModel.

Причина проста. LiveData — это особенность Android. Это компонент с учетом жизненного цикла, который имеет смысл использовать в представлениях Android. Вы можете спроектировать собственное приложение с учетом многоплатформенного кода Kotlin. При ориентировании на Android и iOS в качестве общепринятого кода можно использовать только мультиплатформенный код. Библиотека сопрограмм является мультиплатформенной. LiveData — нет.

Однако, даже если не учитывать мультиплатформенность Kotlin, API потоков имеет больше смысла, поскольку обеспечивает бóльшую гибкость благодаря своим операторам.

Резюме

- API потоков позволяет *асинхронно преобразовывать потоки данных*. Многие его операторы уже доступны "из коробки" и охватывают большинство вариантов использования.
- Благодаря *компонуемой* природе операторов потоков при необходимости можно довольно легко спроектировать собственный оператор.
- Некоторые части потока можно выгружать в фоновый поток или пул потоков, в то же время сохраняя общее представление о преобразовании данных.
- Разделяемый поток передает значения всем своим подписчикам. Вы можете активировать буферизацию и/или воспроизведение значений. Разделяемые потоки — это набор инструментов. Можно использовать их в качестве шины событий для одноразовых событий или при более сложных взаимодействиях между компонентами.
- Когда компонент делится своим состоянием, можно использовать особую разновидность `SharedFlow` — `StateFlow`. Он воспроизводит последнее состояние для новых подписчиков и уведомляет подписчиков только при изменении состояния.

ГЛАВА 11

Вопросы производительности и инструменты профилирования Android

Умелое использование параллелизма в Android позволяет повысить производительность приложения. Вот почему программирование с использованием Kotlin стало основной темой этой книги. Чтобы найти решение проблем производительности, в первую очередь нужно уметь находить их. Не беспокойтесь: в этой главе рассматриваются популярные инструменты Android, обычно используемые для выявления потенциальных проблем, связанных с производительностью.

В естественных условиях Android сталкивается с реальными трудностями, которые влияют на производительность и время автономной работы. Например, не у всех есть безлимитные тарифные планы или надежное подключение. Реальность такова, что приложения для Android должны конкурировать друг с другом за ограниченные ресурсы. Производительность должна стать объектом пристального внимания для любого подобного приложения. Разработка для Android не останавливается на создании приложения. Эффективная разработка обеспечивает плавный и беспроблемный опыт взаимодействия. Даже если вы хорошо разбираетесь в Android, ваше приложение может столкнуться с такими проблемами, как:

- снижение производительности;
- медленный запуск или медленная реакция на действия пользователя;
- разряд батареи;
- расточительное использование ресурсов и засорение памяти;
- ошибки пользовательского интерфейса, которые не приводят к сбою или генерированию исключений, но, тем не менее, влияют на опыт взаимодействия.

Данный список внезапных и странных явлений в приложении ни в коем случае не является исчерпывающим. Как было показано в предыдущих главах, управление многопоточностью может стать непростым делом, если помимо прочего у вас есть взаимодействующие между собой компоненты Android, за которыми нужно следить. Даже если вы четко понимаете, что такое многопоточность, трудно сказать, как на самом деле работает приложение, пока вы не проанализируете его производительность с помощью инструментов профилирования. Чтобы разобраться с по-

добными неясностями, существует несколько полезных инструментов для профилирования различных аспектов Android. Четыре из них доступны непосредственно в Android Studio (рис. 11.1).

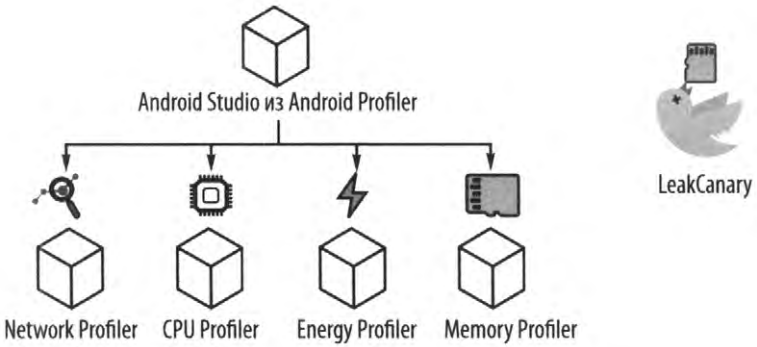


Рис. 11.1. Профилировщики Android Studio и LeakCanary полезны для определения проблем, связанных с производительностью

В этой главе мы рассмотрим инструменты профилирования *Android Profiler* из Android Studio и популярную библиотеку с открытым исходным кодом *LeakCanary*. Мы изучим каждый из этих инструментов, профилируя реальное приложение на предмет выявления потенциальных проблем, связанных с производительностью. Помните приложение для походов, описанное в предыдущих главах? Сюрприз! На его создание нас вдохновил *TrekMe*. TrekMe — это трекинг-приложение для Android, проект с открытым исходным кодом, в котором пользователи загружают интерактивные топографические маршруты для пеших прогулок, чтобы позже использовать их в автономном режиме во время походов. Поначалу этот проект был написан на Java, но в настоящее время его код на восемьдесят с лишним процентов написан на Kotlin. Вот некоторые важные функции TrekMe, доступные пользователям:

- скачивание топографических карт, которые можно использовать в автономном режиме;
- получение текущего положения устройства даже при отсутствии сети, в то время как приложение делает все возможное, чтобы продлить срок службы батареи;
- отслеживание маршрутов в мельчайших деталях без разрядки аккумулятора устройства, когда это особенно необходимо;
- доступ к другой полезной информации без подключения к Интернету.

Мы рекомендуем изучить TrekMe, чтобы вам было легче ориентироваться в этой главе. Исходный код приложения можно найти на странице GitHub[1]. После того как вы клонируете проект, откройте его в Android Studio. Наконец, запустите экземпляр эмулятора из *диспетчера виртуальных устройств Android* (AVD), в котором будете запускать TrekMe.

[1] См. **https://oreil.ly/j7KbY**.

Производительность имеет решающее значение. Нередко можно обнаружить отставание в производительности в любом приложении, но к сбору подобного рода сведений нужно подходить с осторожностью. Разработчик должен решить, какие выбрать инструменты и какие оптимизации перевешивают преимущества затрат на их создание. Профилирование приложения позволяет объективно исследовать его производительность. Чтобы подготовить вас к неожиданным сюрпризам, с которыми вы можете столкнуться, мы рассмотрим приложение TrekMe, используя Android Profiler.

Android Profiler

Android Profiler позволяет анализировать работу центрального процессора, памяти и сетевую активность. На рис. 11.2 показана Android Studio и время выполнения приложения TrekMe в нижней половине консоли.

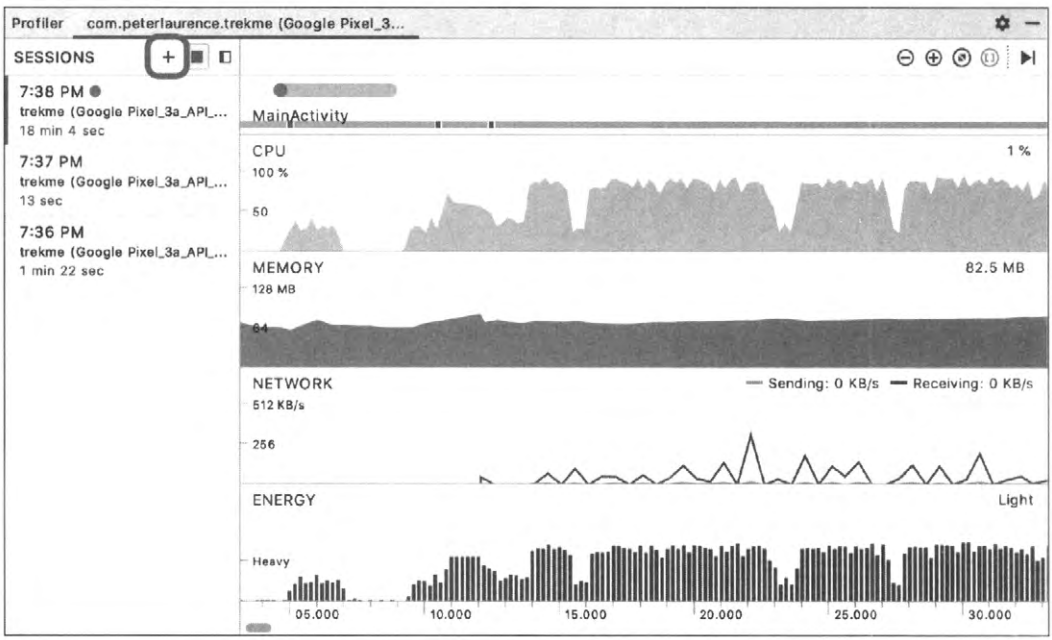


Рис. 11.2. Запись данных во время сеанса профилирования. Активный сеанс присоединяется к работающему приложению в эмуляторе (не видно на изображении)

Профилирование можно реализовать тремя способами:

1. Если приложение не запущено, щелкните по значку **Profile app** в правом верхнем углу, чтобы одновременно создать экземпляр приложения и профилировщика. Так вы создаете и компилируете новый работающий экземпляр приложения. Android Studio откроет новый сеанс, предоставив вам поток данных в режиме реального времени.

2. Если приложение уже запущено, щелкните по значку + и выберите работающий эмулятор.
3. Вы также можете импортировать ранее сохраненный сеанс профилирования с помощью значка +. После этого можно загрузить ранее сохраненный файл с расширением *hprof*.

Вы можете записывать и сохранять данные в каждом сеансе. На рис. 11.3 показан скриншот сохраненных сеансов профилирования с различными видами данных, которые можно записывать с помощью Android Profiler.

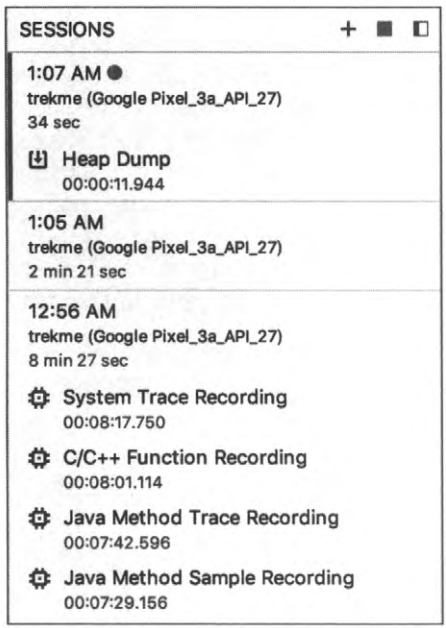


Рис. 11.3. Сохраняем дампы кучи или различные виды трассировки процессора

И *трассировки методов*, и *дампы кучи* можно сохранять как отдельные записи в рамках работающей сессии. Трассировки методов показывают трассировку стека методов и функций, которые можно записывать при профилировании процессора. Между тем дамп кучи обозначает данные, собранные при *сборе мусора*, что позволяет анализировать, какие объекты занимают ненужное место в памяти.

Android Profiler записывает по одному сеансу приложения за раз. Тем не менее вы можете сохранить несколько записей и переключаться между ними для сравнения данных. Яркая точка указывает на запись сеанса. На рис. 11.3 записаны три сеанса. У последнего сеанса имеется сохраненный дамп кучи, который относится к журналу хранимой памяти в JVM на момент создания моментального снимка. Мы расскажем об этом подробнее в *разд. "Memory Profiler" далее в этой главе*. В первом сеансе сохранены различные виды записей профиля ЦП. Мы обсудим их в *разд. "CPU Profiler"*.

Android Studio кэширует сеансы только на время существования экземпляра Android Studio. Если перезапустить Android Studio, то записанные сеансы не сохранятся.

В последующих разделах подробно показано, как Android Profiler оценивает ресурсы устройства на виртуальной машине во время выполнения. Мы будем использовать четыре профилировщика: *Network Profiler*, *CPU Profiler*, *Energy Profiler* и *Memory Profiler*. Все они записывают потоки данных во время выполнения приложения, к которым можно получить более детальный доступ в специальных представлениях.

По умолчанию TrekMe побуждает пользователей загружать подробные топографические карты непосредственно на свои устройства, пока они находятся дома и могут без труда сделать это. При создании новых топографических карт в TrekMe потребляется больше всего ресурсов. Карты могут отображаться, когда пользователь путешествует пешком, даже если мобильное покрытие ненадежно. Возможность создавать карты позволяет выбрать официальный картографический сервис, например *Национальный географический институт* (IGN) или *Геологическую службу США* (USGS), или какой-то иной сервис (рис. 11.4). TrekMe загрузит карту из выбранного сервиса, выкладывая ее тайлами один за другим.

Рис. 11.4. TrekMe позволяет создавать и загружать карты из разных сервисов

В оставшейся части этой главы мы будем профилировать приложение TrekMe, создавая карту с помощью IGN, чтобы изучить время, необходимое для загрузки карты, и убедиться, что оно оптимально. Используя профилирование, можно изучать такие вопросы, как:

- выполняем ли мы быстрые сетевые вызовы?

- возвращаются ли данные, которые мы получаем в ответе, в наиболее эффективном формате?
- какие части приложения больше всего грузят ЦП?
- какие действия Android больше всего разряжают батарею?
- какие объекты потребляют больше всего памяти в куче?
- что потребляет больше всего памяти?

В следующем разделе мы ответим на первые два вопроса с помощью Network Profiler, а оставшиеся вопросы рассмотрим в последующих разделах.

Network Profiler

При совершении сетевого вызова на устройстве под управлением Android включается радио, чтобы обеспечить обмен данными по сети. Радио остается включенным в течение короткого периода времени, чтобы убедиться в отсутствии дополнительных запросов. На некоторых телефонах при подключении к сети каждые две минуты устройство постоянно работает на полную мощь. Слишком большое количество сетевых вызовов может оказаться затратным с точки зрения ресурсов, поэтому важно анализировать и оптимизировать использование сети в приложении.

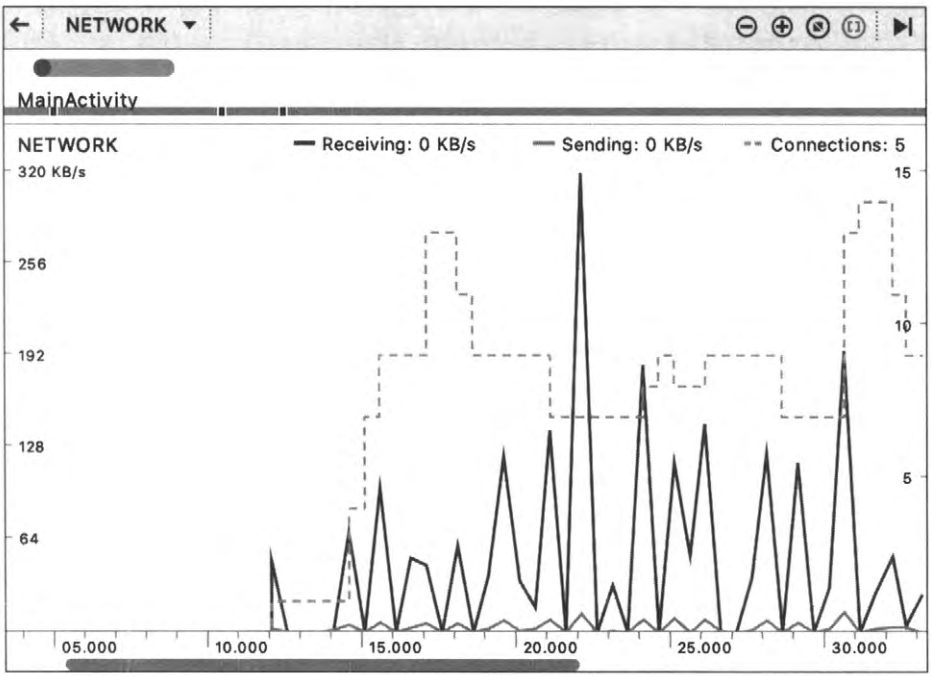


Рис. 11.5. Временна́я шкала Network Profiler с записью создания карты Испании в TrekMe. В левом верхнем углу длинная линия под меткой **MainActivity** представляет собой активный сеанс Activity, а короткая толстая линия над меткой **MainActivity** с точкой слева — это события касания

Network Profiler создает разбивки соединений, используемые библиотеками *HttpURLConnection* или *OkHttp*. Он может предоставить такую информацию, как время сетевого запроса или ответа, заголовки, cookies-файлы, форматы данных, стек вызовов и многое другое. При записи сеанса Network Profiler создает интерактивные визуальные данные, пока вы продолжаете взаимодействовать с приложением.

При создании карты с помощью IGN приложение TrekMe отображает ее на экране в виде плиток (тайлов, от англ. tile). Однако иногда кажется, что на это уходит много времени. На рис. 11.5 показан профилировщик, фиксирующий входящие или исходящие сетевые запросы, а также доступные подключения при создании карты в TrekMe.

Чтобы подробно изучить эти подключения, можно выделить выбранный диапазон временно́й шкалы. Это расширяет новое представление рабочей области Network Profiler, позволяя получить доступ к вкладкам **Connection View** (Представление "Подключение") и **Thread View** (Представление "Поток") для дальнейшего анализа этих сетевых вызовов.

Просмотр сетевых вызовов с помощью представлений *Connection View* и *Thread View*

В представлении **Connection View** показаны отправленные или полученные данные. Это видно на рис. 11.6 в выделенной части временно́й шкалы. Возможно, наиболее примечательной является способность этой вкладки сортировать файлы ресурсов по размеру, статусу и времени. Щелкнув кнопкой мыши по заголовку каждого раздела, вы выполняете сортировку по нужному фильтру. В разделе **Timeline** (Временная шкала) видны полоски, состоящие из двух цветов: светлая часть обозначает длительность запроса, а более темная часть — это длительность ответа.

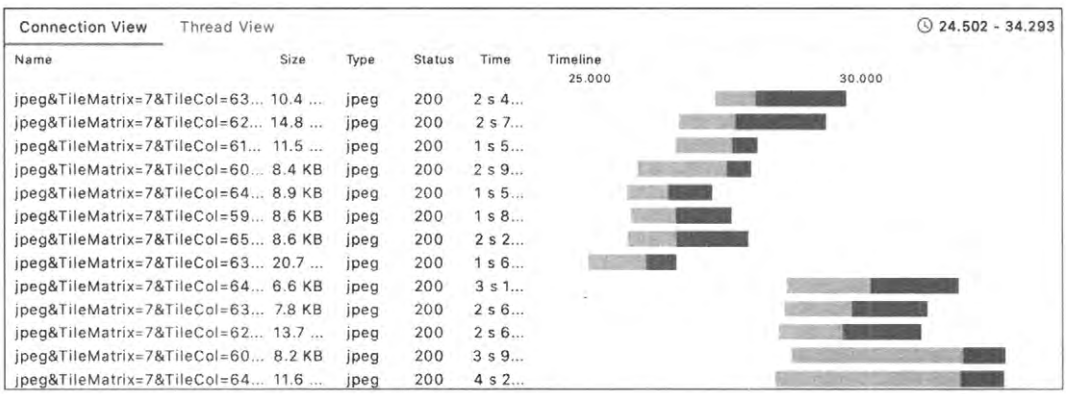


Рис. 11.6. В представлении **Connection View** показан список отдельных сетевых вызовов

Представление **Connection View** напоминает временну́ю шкалу в представлении **Thread View**, но это не совсем одно и то же. Представление **Thread View** показывает сетевые вызовы, выполняемые в назначенных инициирующих потоках, кото-

рые могут отображать несколько сетевых вызовов, выполняемых параллельно. Скриншот, изображенный на рис. 11.7, представляет собой дополнение к предыдущему изображению с использованием того же набора данных.

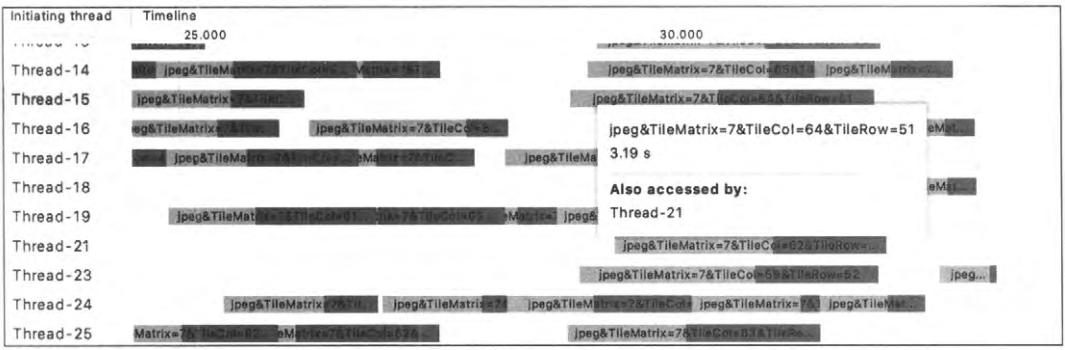


Рис. 11.7. В представлении **Thread View** показан список сетевых вызовов, выполняемых в каждом потоке

Наблюдение за тем, как рабочие потоки распределяют работу в режиме реального времени, может помочь выявить области, которые нужно улучшить. Пул потоков TrekMe отвечает за автоматическое прерывание, по мере необходимости, работы по скачиванию всех этих изображений.

На обоих изображениях показаны примерно 23 секунды сетевых вызовов, при этом время отклика демонстрирует аналогичную тенденцию. По сравнению с запросами ответы занимают непропорционально больше времени, необходимого для завершения всего вызова. На то может быть несколько причин: например, соединение с сервером может быть хуже, если устройство пытается получить эти данные из страны, которая находится далеко. Может быть, что-то не так с вызовом запроса в серверной части. Независимо от причины можно сказать, что наши сетевые вызовы, возможно, не самые быстрые. Однако наличие быстрого времени запроса *и* медленного времени отклика указывает на внешние факторы, которые устройство не контролирует.

Теперь перейдем ко второму вопросу: используем ли мы самый эффективный формат данных? Посмотрим на тип подключения на вкладке **Connection View** (Представление "Подключение"), как показано на рис. 11.6. Если вам не нужна прозрачность изображений, избегайте использования файлов PNG, поскольку этот формат не сжимается так хорошо, как JPEG или WebP. В нашем случае сетевые вызовы возвращают данные в формате JPEG. Мы хотим, чтобы файлы, обеспечивающие постоянное и хорошее качество изображения, позволяли пользователям увеличивать эти изображения настолько, насколько это нужно. JPEG-файлы также занимают меньше памяти, чем PNG-файлы.

Мы можем получить более подробную информацию о каждом сетевом вызове и его данных, выбрав любой элемент: так мы получаем доступ к новому представлению в Network Profiler с правой стороны, где находятся вкладки **Overview** (Обзор), **Re-**

sponse (Ответ), **Request** (Запрос) и **Call Stack** (Стек вызовов). В следующем разделе мы сможем изучить особенности отдельного сетевого вызова и определить, где в коде он выполняется.

Сетевой вызов, расширенный вариант: *Overview, Response, Request, Call Stack*

Разработчики приложений для Android привыкли работать с другими платформами, чтобы добиться паритета функциональных возможностей, и не только. Предположим, сетевой вызов начинает возвращать неверную информацию по запросу. Команде, занимающейся разработкой API, нужны подробности запроса и ответа, которые вы получаете на стороне клиента. Как отправить им необходимые параметры запроса и заголовки Content-Type, которые им нужно исследовать?

Network Profiler дает возможность проверять сетевые ответы и запросы на правой панели в представлениях **Connection View** или **Thread View** (рис. 11.8).

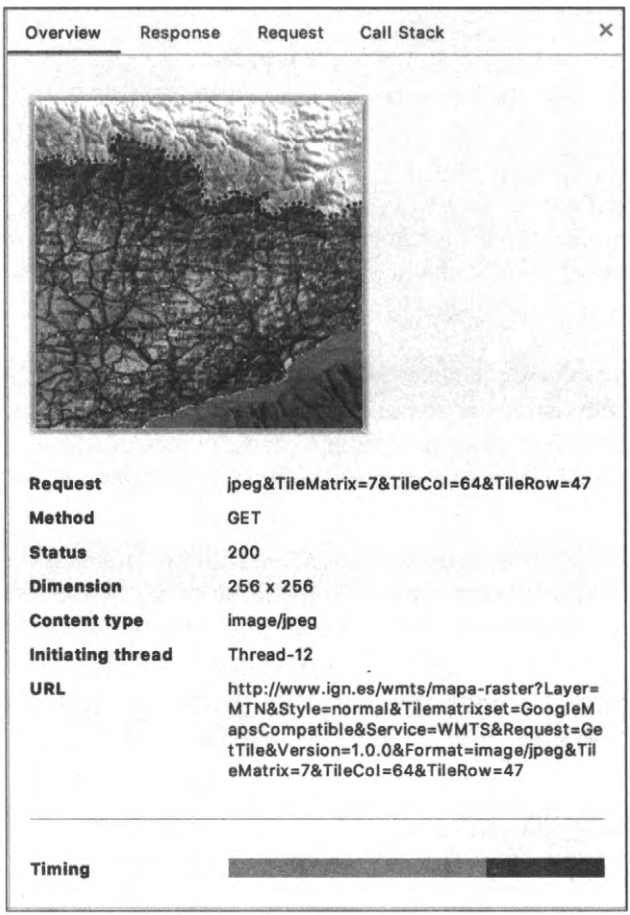


Рис. 11.8. Network Profiler позволяет проверять информацию об ответах и запросах

На вкладке **Overview** (Обзор) подробно описаны важные моменты, зафиксированные в запросе и ответе.

***Request** (Запрос).*

Путь и возможные параметры запроса.

***Status** (Состояние).*

Код состояния HTTP, возвращаемый в ответе.

***Method** (Метод).*

Тип используемого метода.

Content-Type.

Медиа-тип ресурса.

***Size** (Размер).*

Размер ресурса, возвращаемого в ответе.

Вкладки **Request** (Запрос) и **Response** (Ответ) показывают разбивку заголовков, параметров, данных тела и т. д. На рис. 11.9 изображен точный сетевой вызов, как и на предыдущем изображении, за исключением выбранной вкладки **Response** (Ответ).

Как видно из ответа, TrekMe использует базовый HTTP API. Другие типы форматов данных API возвращают HTML, JSON и другие ресурсы. Когда это применимо, вкладки **Request** (Запрос) и **Response** (Ответ) предлагают данные тела в виде форматированного или необработанного представления. В нашем случае мы получаем файлы в формате JPEG.

Наконец, на вкладке **Call Stack** (Стек вызовов) видна трассировка стека для соответствующих вызовов, чтобы выполнить сетевое подключение (рис. 11.10). Незатененные вызовы — это вызовы методов в стеке вызовов, поступающие из кода. Можно щелкнуть правой кнопкой мыши на выбранном вызове, чтобы перейти к исходному коду.

Network Profiler полезен не только для аналитики. Как видите, вы можете быстро обрабатывать большое количество информации. Network Profiler предоставляет массу возможностей: от кэширования повторяющихся вызовов до подтверждения контрактов API. Стоит иметь этот инструмент у себя в наборе.

Плохое сетевое взаимодействие не единственная причина медленного отображения. Задача создания совершенно новой топографической карты сама по себе тяжела, но, как мы уже определили, никаких дополнительных действий для улучшения времени загрузки или формата данных не требуется. Однако было бы упущением списывать медленное время загрузки только на медленное время отклика. После того как приложение TrekMe получит сетевые данные, оно должно обработать их, чтобы отобразить пользовательский интерфейс. По этой причине нужно выполнить проверку на предмет потенциальной неэффективности отрисовки карты после сетевых вызовов. В этом нам может помочь *CPU Profiler*. В следующем разделе мы рассмотрим отображение карты Испании с его помощью.

Рис. 11.9. Network Profiler захватывает сетевые вызовы для отображения карты

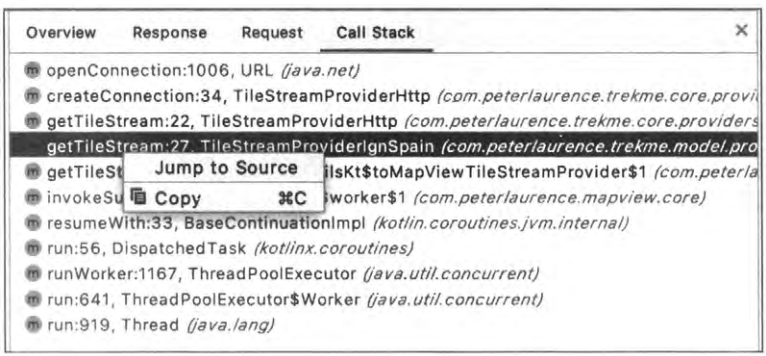


Рис. 11.10. Вкладка **Call Stack**

CPU Profiler

Хотя Network Profiler может предоставить информацию о сетевых вызовах, он не дает полной картины. У нас есть стек сетевых вызовов, но мы не знаем, как долго на самом деле выполняются определенные методы. Здесь на помощь приходит CPU Profiler. Он помогает выявить чрезмерное потребление ресурсов, анализируя, сколько времени потрачено на выполнение функции, и отслеживает, в каком потоке выполняется вызов. Почему это важно? Если TrekMe потребляет слишком много ресурсов, то работа приложения замедляется, что влияет на опыт взаимодействия. Чем больше мощности процессора используется, тем быстрее разряжается батарея.

CPU Profiler позволяет просматривать записи профиля ЦП и данные в режиме реального времени, анализируя стек вызовов по потокам (рис. 11.11).

В следующих разделах мы разберем хронологию загрузки процессора, хронологию активности потока и панели анализа. Поскольку TrekMe, похоже, тратит много времени на перенос работы в фоновые потоки, мы выберем один из них для более тщательного изучения.

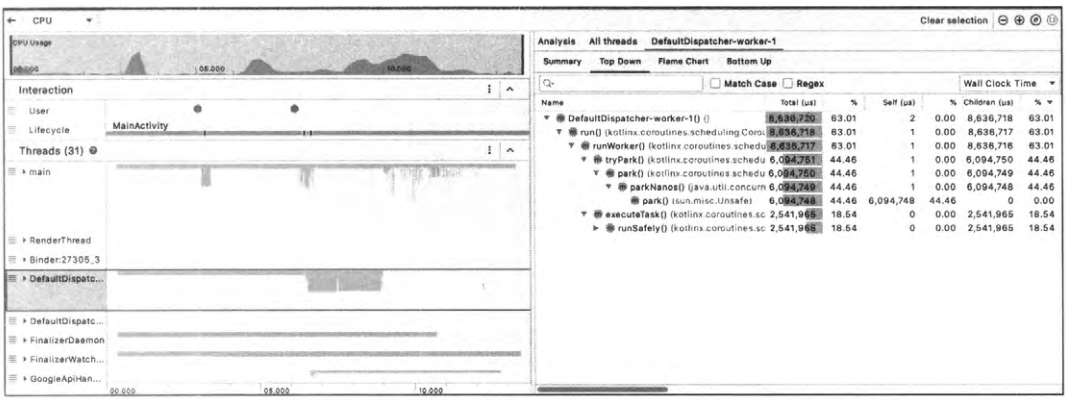


Рис. 11.11. CPU Profiler показывает стек вызовов и записанное время выполняемых методов

Хронология загрузки процессора

Хронология загрузки процессора упорядочивает региональные стеки вызовов в записанные потоки на панели **Threads** (Потоки). График на рис. 11.12 показывает всплески загрузки процессора, где число обозначает степень загрузки в процентах. Если вы сделали запись трассировки, то можете выделить нужные места на графике, чтобы увидеть подробную информацию.

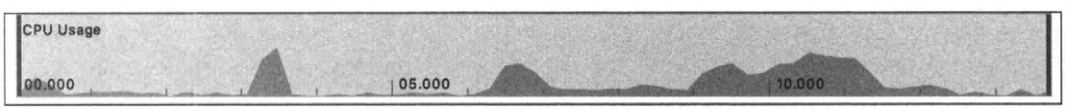


Рис. 11.12. Хронология загрузки процессора

Android Studio позволяет перетаскивать записанный образец, чтобы отобразить диаграмму вызовов. Нажав кнопку **Record** (Запись), вы попадете на отдельный экран записи трассировки. Чтобы создать более детализированные диаграммы вызовов, которые мы рассмотрим в следующем разделе, полезно выделить мелкие части записанной трассировки.

Хронология активности потока

Хронология активности потока сопровождает хронологию загрузки процессора, показывая каждый запущенный поток в приложении. Если имела место запись трассировки, у вас должна быть возможность выбрать поток для просмотра стека вызовов, захваченного в выбранном временно́м диапазоне. На рис. 11.13 в приложении создан и использован 31 поток. Эти потоки создаются вашим кодом, системой Android или сторонней библиотекой.

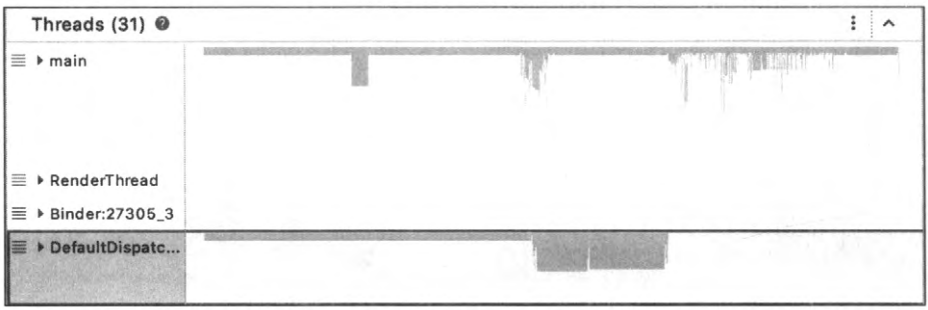


Рис. 11.13. Хронология активности потока

Самые светлые блоки обозначают работающий или активный поток. В основном потоке не так много интересного. В данном случае мы ожидаем, что фоновые потоки выполнят необходимую работу по скачиванию сетевых данных. Похоже, у нас есть основной поток, ожидающий в одном из потоков DefaultDispatcher половину времени. Дважды щелкните кнопкой мыши по отдельному потоку, чтобы раскрыть стек вызовов.

Ниже находится диаграмма вызовов (рис. 11.14).

Диаграмма вызовов показывает стек вызовов сегментированного диапазона времени для загрузки процессора. Верхние поля обозначают инкапсулирующий родительский метод, а методы ниже — это вызываемые дочерние методы. Родительский метод ожидает завершения выполнения дочерних методов, поэтому это подходящее место, чтобы увидеть, какие методы TrekMe могут выполняться в течение длительного времени, например метод `TileStreamProviderHttp`.

Если вы читаете печатное издание, имейте в виду, что методы обозначены разными цветами. Методы Android выделены оранжевым цветом, написанные вами методы — зеленым, а сторонние библиотеки — синим. В этой сопрограмме наибольшее время выполнения приходится на `TileStreamProviderHttp.getTileStream(...)`. Это

ожидаемо, учитывая, что данный вызов выполняет отдельные сетевые запросы для каждой плитки.

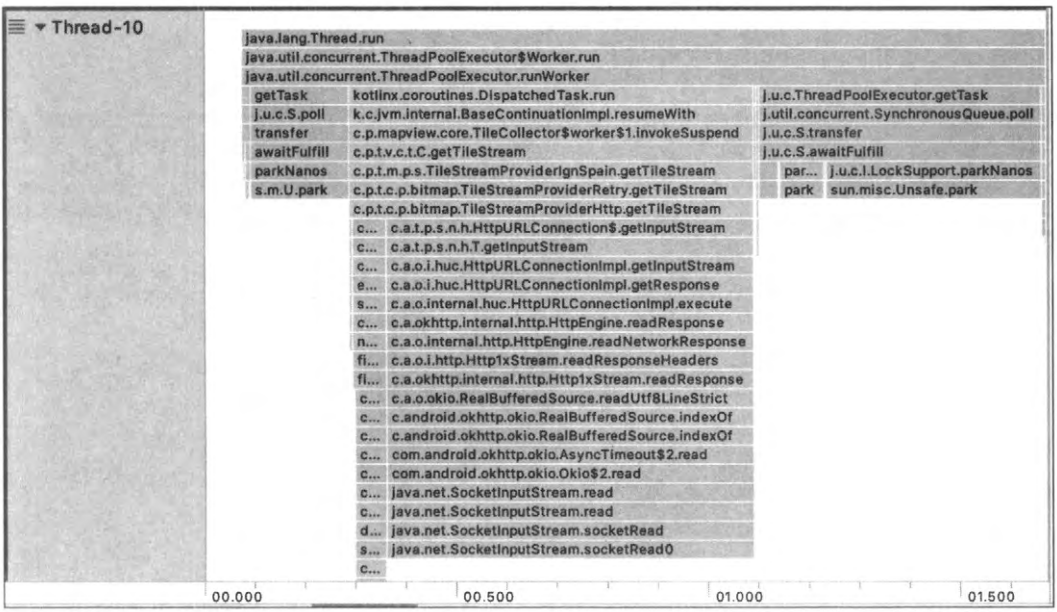


Рис. 11.14. Диаграмма вызовов показывает захваченные методы, которые идут сверху вниз

Панель *Analysis*

Панель **Analysis** (Анализ) представляет собой многоуровневую вкладку. В верхней части панели выделен активный набор потоков. Под меню с вкладками находится панель поиска над трассировкой стека. Панель поиска можно использовать для фильтрации данных трассировки, связанных с конкретным вызовом. Ниже находится набор вкладок, предназначенных для отображения визуальных данных из трассировки методов в трех представлениях: *сверху вниз*, *снизу вверх* и *огненная диаграмма.*

Представление "сверху вниз" отображает графическое представление трассировки метода сверху вниз на диаграмме. Любой вызов, выполненный внутри метода, отображается как дочерний элемент исходного метода. Показанный на рис. 11.15 метод `getTileStream`, который используется в TrekMe, ожидает серию вызовов для подключения к Интернету и чтения из потока данных.

Это представление показывает, как распределяется время процессора по трем направлениям

***Self** (Собственно).*

Само время выполнения метода.

***Children** (Потомки).*

Время, необходимое для выполнения вызываемых методов.

***Total** (Всего).*

Время из предыдущих двух направлений.

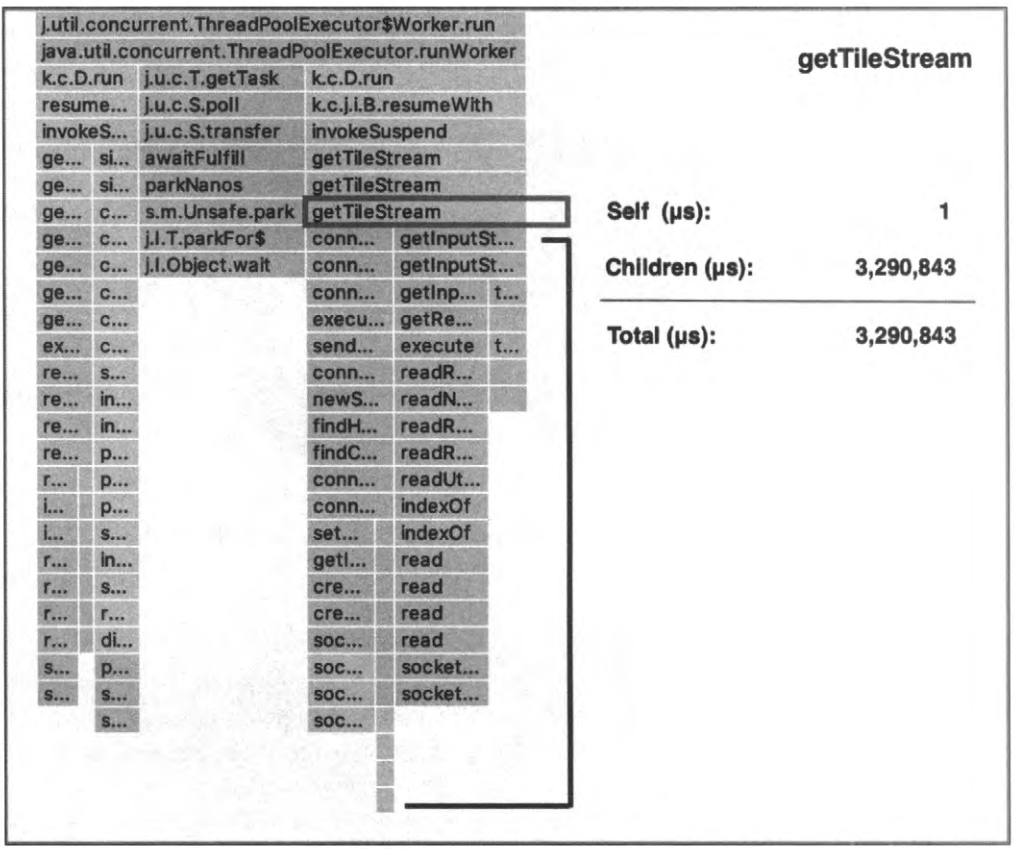


Рис. 11.15. Представление "сверху вниз"

В случае с `getTileStream` бóльшая часть времени уходит на сами сетевые вызовы: в частности, запрос на подключение и `getInputStream` для получения входящих данных из сети. Для сервера IGN это время может различаться в зависимости от того, из какой страны к нему выполняется доступ, и времени суток. Поскольку именно клиент потребляет данные сервера, TrekMe не может контролировать работу сервера.

В отличие от представления "сверху вниз", представление "снизу вверх" (рис. 11.16) показывает *листовые элементы* стека вызовов в обратном порядке. Для сравнения, такое представление отображает значительное количество методов, которые могут быть полезны для определения самых прожорливых в отношении процессорного времени методов.

Последняя вкладка — это представление "огненная диаграмма", которое обеспечивает сводное отображение операций снизу вверх. Оно предоставляет перевернутую диаграмму вызовов, чтобы лучше видеть, какие функции или методы потребляют больше времени процессора.

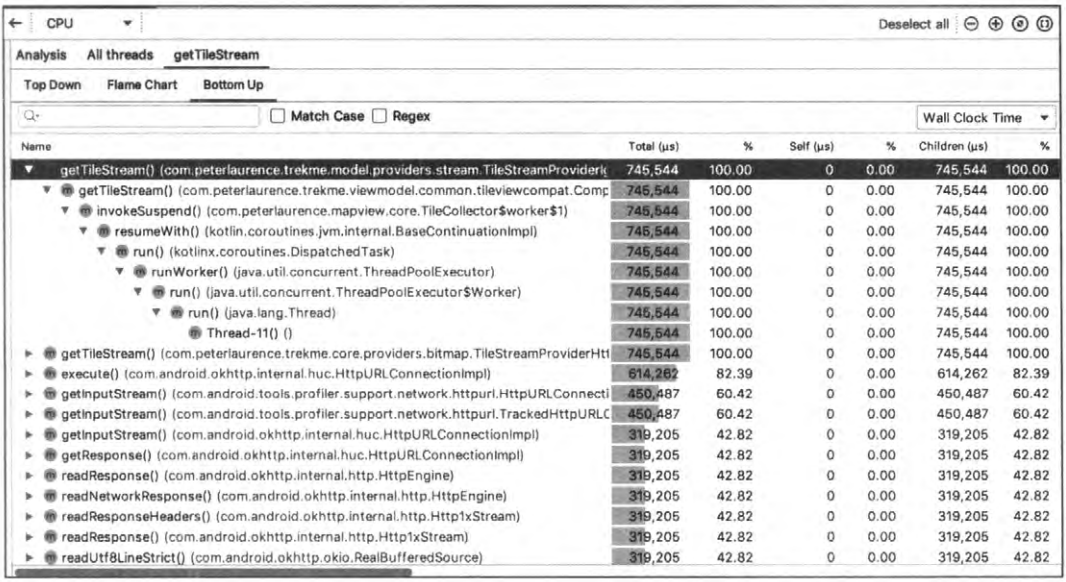


Рис. 11.16. Представление "снизу вверх"

Подводя итог, можно сказать, что при профилировании процессора у вас могут быть три различных вида представлений, в зависимости от того, насколько далеко вы хотите зайти:

- представление "сверху вниз" показывает процессорное время каждого вызова метода и время его вызываемых методов;
- представление "снизу вверх" — обратный вариант предыдущего представления. Оно наиболее удобно для сортировки методов, потребляющих наибольшее или наименьшее количество времени;
- "огненная диаграмма" инвертирует и объединяет стек вызовов по горизонтали с другими вызываемыми методами одного уровня, чтобы показать, какие из них первыми потребляют больше всего процессорного времени.

Существуют не только три различных способа отображения данных, но и различные виды стеков вызовов, которые можно записывать. В последующих разделах мы рассмотрим разные виды трассировки методов в CPU Profiler. Пока вы стараетесь понять, какую информацию CPU Profiler пытается собрать, мы обратимся к *трассировке методов* с помощью CPU Profiler и запишем сегмент TrekMe, создающий новую карту.

Трассировка методов

CPU Profiler позволяет *записывать трассировку* для анализа и отображения ее состояния, продолжительности, типа и многого другого. Трассировка относится к записи активности устройства в течение короткого периода времени. Трассировка методов не произойдет до тех пор, пока кнопка записи не будет нажата дважды: один раз, чтобы начать запись, и второй раз, чтобы закончить ее. Доступны четыре конфигурации сэмплов и трассировок (рис. 11.17).

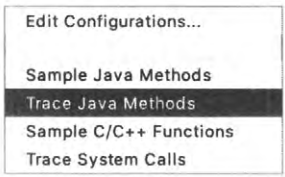


Рис. 11.17. Доступные конфигурации сэмплов и трассировок для разработчиков Android

Конфигурация *Sample Java Methods* захватывает стек вызовов приложения или диаграмму вызовов (показанную в предыдущих разделах). Диаграмма вызовов отображается под хронологией активности потока, которая показывает, какие потоки активны в определенное время. Эти трассировки хранят отдельные сеансы в правой панели для сравнения с сохраненными сеансами других пользователей.

Выбрав эту конфигурацию, можно изучить стек вызовов TrekMe, наведя указатель мыши на определенные методы (рис. 11.18).

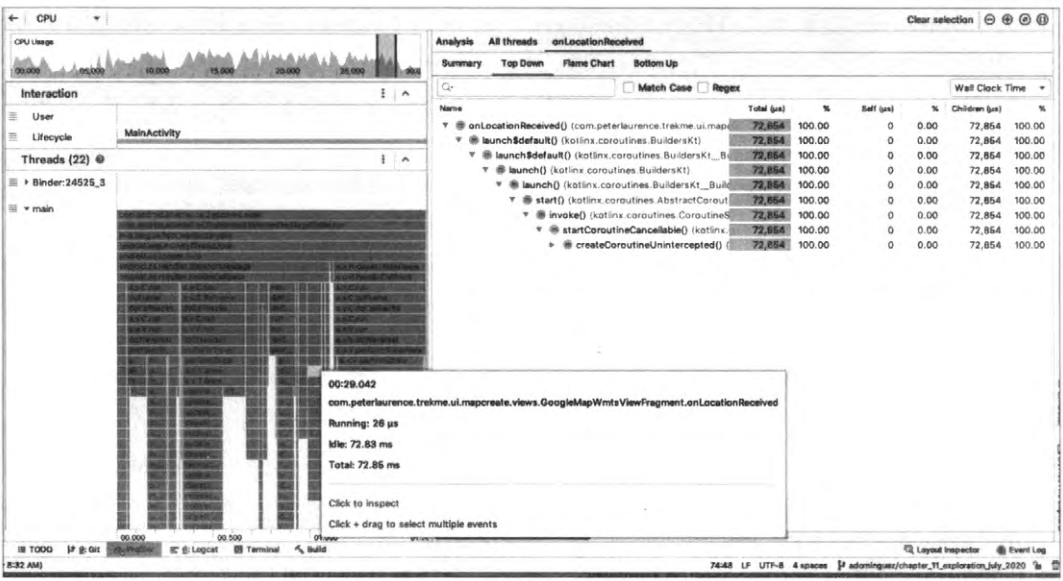


Рис. 11.18. Конфигурация Sample Java Methods

Следите за тем, чтобы запись не длилась слишком долго. Как только запись достигает допустимого размера, трассировка прекращает сбор данных, даже если запись текущего сеанса продолжается.

В отличие от Sample Java Methods, конфигурация *Trace Java Methods* объединяет ряд временны́х меток, записанных для начала и конца вызова метода. При желании можно отслеживать конфигурацию *Sample C/C++ functions*, чтобы понять, как приложение взаимодействует с Android. Запись образцов трассировки для встроенных потоков доступна для Android API 26 и более поздних версий.

В повседневной речи термины "метод" и "функция", как правило, взаимозаменяемы, когда мы говорим об анализе с трассировкой методов. На данном этапе вам, возможно, интересно, почему методы Java и функции C/C++ достаточно различаются, чтобы иметь значение при профилировании процессора.

В этих конфигурациях Android Profiler использует термин "метод" для обозначения кода на базе Java, в то время как термин "функция" обозначает потоки. Разница между ними заключается в том, что порядок выполнения методов сохраняется с помощью стека вызовов, в то время как потоки создаются и планируются самой системой Android.

Наконец, есть *Trace System Calls* (см. рис. 11.17). Это мощная конфигурация, доступная разработчикам приложений для Android. Она возвращает графическую информацию о данных отображения кадров.

Trace System Calls записывает аналитические данные в разделе **CPU Cores** (Ядра процессора), чтобы увидеть, как происходит планирование по всем направлениям. Эта конфигурация становится более значимой для обнаружения проблем в ядрах. Такие проблемы могут появляться в местах, где RenderThread выдыхается, особенно когда речь идет о кадрах красного цвета. В отличие от других конфигураций, Trace System Calls показывает состояние потока и ядро процессора, на котором она в данный момент работает (рис. 11.19).

Одной из ключевых особенностей системной трассировки является доступ к *RenderThread*. Эта конфигурация способна показать, где могут возникать проблемные места в производительности при отображении пользовательского интерфейса. В случае с рис. 11.19 видно, что бóльшая часть времени простоя приходится на отрисовку самих тайлов.

Система Android пытается перерисовать экран в зависимости от частоты обновления (от 8 до 16 мс). Пакеты работ, длительность которых превышает частоту кадров, могут привести к *пропуску кадров*, обозначенных красными слотами в разделе **Frames** (Кадры). Кадры пропускаются, если какая-то задача не возвращается до перерисовки экрана. В случае с этой записью похоже, у нас действительно есть несколько пропущенных кадров, обозначенных цифрами, которыми помечены поля в подразделе **Frame** (Кадр) раздела **Display** (Отображение).

TrekMe сохраняет каждый кадр в файл формата JPEG и загружает изображение в объект `Bitmap` для декодирования. Однако на рис. 11.19 видно, что в RenderThread

длина DrawFrame не совсем соответствует интервалам частоты отрисовки. Чуть ниже это время простоя связано с различными длительно работающими методами `decodeBitmap` в пуле потоков.

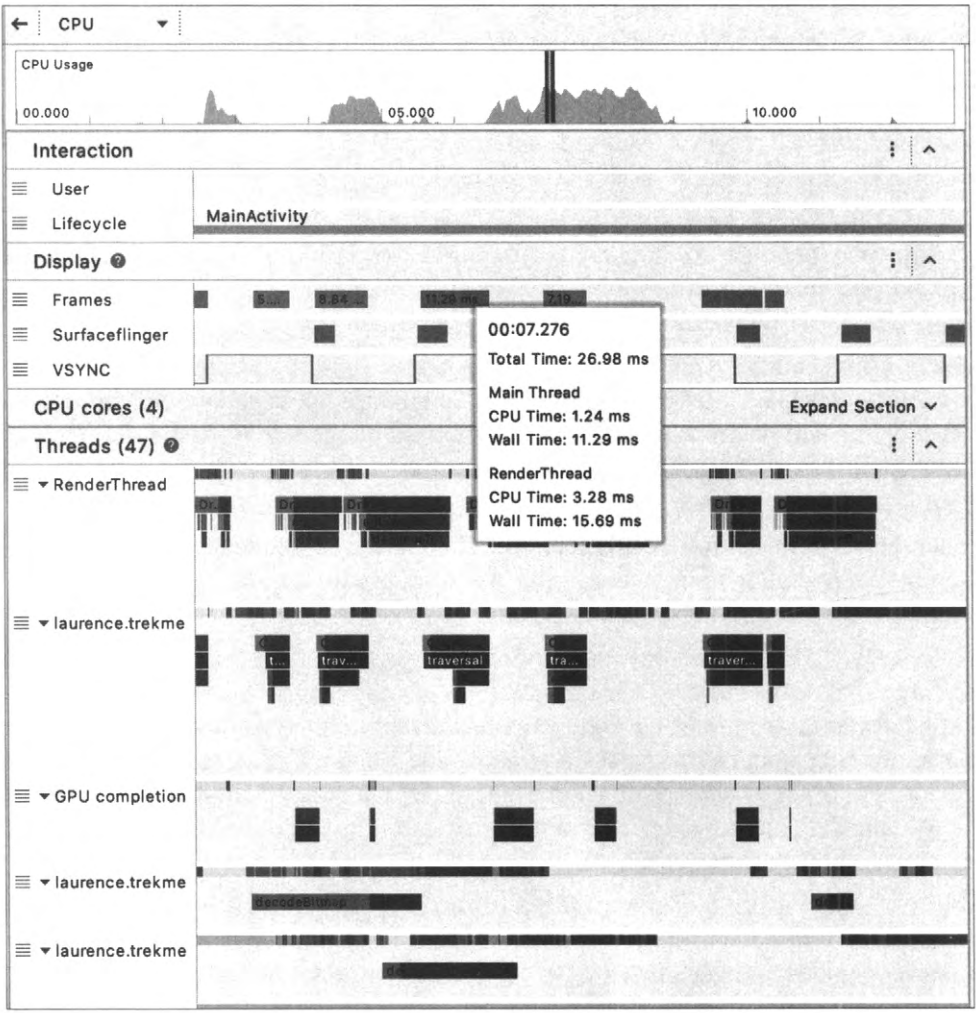


Рис. 11.19. Конфигурация Trace System Calls выявляет пропущенные кадры, где время указано в разделе **Frames**

Здесь есть несколько вариантов, которые потенциально можно было бы рассмотреть для более быстрой отрисовки, а именно кэширование сетевых ответов для изображений или даже *предварительная выборка*. Пользователям, которым нужно несколько мегабайт данных, подойдет предварительная выборка, если у устройства есть доступ как минимум к 3G-сети. Проблема состоит в том, что, возможно, это не лучший вариант для отображения растровых изображений до того, как мы *узнаем*, что нужно отобразить. Еще один вариант — потенциальное кодирование данных в более сжатый формат для упрощения декодирования. Каким бы ни было решение,

разработчик должен оценить компромиссы и усилия по реализации определенных оптимизаций.

Концепция предварительной выборки обозначает прогнозирование того, какие данные будут приходить в будущем запросе, и упреждающий захват этих данных при наличии активного радиоподключения. Каждый радиозапрос несет затраты с точки зрения времени, необходимого для включения радио и разряда батареи, который происходит, чтобы поддерживать радио в работающем состоянии, поэтому разработчики могут воспользоваться дополнительными вызовами, когда радио уже включено.

Использование конфигурации Sample Method Trace

Теперь, когда вы знакомы с тем, что предлагают конфигурации записи, мы воспользуемся конфигурацией *Sample Method Trace* для приложения TrekMe. Записи профиля процессора отделены от временно́й шкалы CPU Profiler. Для начала нажмите кнопку **Record** (Запись) в верхней части экрана, чтобы проанализировать активность процессора при взаимодействии с TrekMe.

По окончании записи появится правая панель с вкладками, отображающая время выполнения для вызовов сэмплов или трассировки. Для анализа можно сразу выделить несколько потоков. Обычный разработчик приложений для Android может не использовать все эти вкладки постоянно. Тем не менее полезно знать, какие инструменты есть в вашем распоряжении.

В TrekMe имеется предопределенный набор итерируемых плиток (тайлов) для скачивания. Несколько сопрограмм одновременно считывают итерируемый тайл и выполняют сетевой запрос для каждого тайла. Каждая сопрограмма декодирует растровое изображение сразу после успешного сетевого запроса. Эти сопрограммы отправляются в какой-нибудь диспетчер, например `Dispatchers.IO`, а отображение возможно, когда результат отправляется обратно в UI-поток. Данный поток никогда не блокируется в ожидании декодирования растрового изображения или ожидания сетевого запроса.

Уменьшенная версия хронологии загрузки процессора на рис. 11.20, на первый взгляд, кажется не чем иным, как ссылкой на предыдущее представление экрана. Однако можно взаимодействовать с этими данными для дальнейшей детализации, выделив отрезок времени с помощью селектора диапазонов (рис. 11.21).

На рис. 11.22 мы рассматриваем один из наиболее длительных методов — `getTileStream`. Левая панель под временно́й шкалой позволяет упорядочить *потоки* и *взаимодействия* с помощью перетаскивания. Возможность группировать потоки также означает, что группы трассировок стека можно выделять. Можно развернуть поток в записанной трассировке, дважды щелкнув кнопкой мыши на потоке, чтобы отобразить раскрывающийся список стека вызовов.

При выборе элемента также открывается дополнительная панель справа. Это *панель анализа*, позволяющая подробнее изучить трассировку стека и оценить время

выполнения. Отслеживание загрузки процессора важно, но, вероятно, вам хотелось бы иметь возможность анализировать, как приложение взаимодействует с аппаратными компонентами Android. В следующем разделе мы рассмотрим инструмент *Energy Profiler* из Android Studio.

Рис. 11.20. CPU Profiler разделяет записанную трассировку

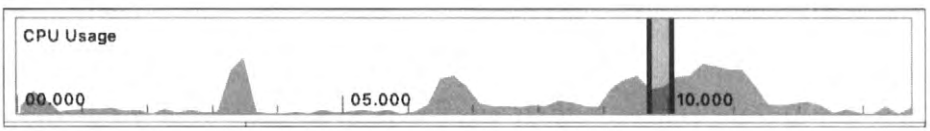


Рис. 11.21. Селектор диапазонов помогает управлять разделами выделенных диапазонов

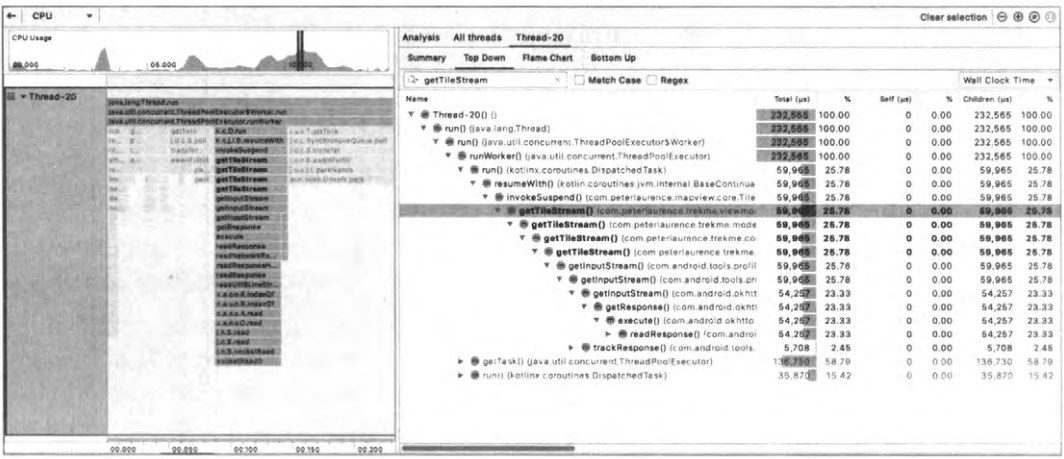


Рис. 11.22. Можно искать конкретный метод с помощью функции поиска

Чрезмерные сетевые вызовы на устройствах, работающих под управлением Android, также являются *энергоемкими*. Чем дольше радио остается в активном со-

стоянии для обмена данными по сети, тем больше потребление ресурсов процессора и тем быстрее разряжается аккумулятор. Следуя этой логике, было бы справедливо предположить, что на сетевые ресурсы приходится бóльшая часть энергопотребления. Это можно проверить с помощью Energy Profiler.

Energy Profiler

Для определения значительного объема энергопотребления лучше всего использовать Energy Profiler. Когда приложение выполняет сетевой запрос, оно включает мобильное радио. Загрузка процессора ускоряется по мере общения устройства с сетью, в результате чего аккумулятор разряжается быстрее.

TrekMe предварительно масштабирует растровые изображения, чтобы обеспечить постоянное использование памяти и энергии, когда пользователь увеличивает и уменьшает масштаб изображения. Когда пользователь создает и скачивает карту, по умолчанию детали карты скачиваются с самым высоким разрешением. На панели событий показаны более высокие уровни потребления при скачивании больших фрагментов данных.

С помощью перетаскивания можно выбрать диапазон временнóй шкалы, чтобы отобразить сведения о событиях. На рис. 11.23 видна всплывающая панель с разбивкой графика энергопотребления. Первая половина содержит категории **CPU** (ЦП), **Network** (Сеть) и **Location** (Местоположение), которые соответствуют каждой категории, представленной на составной диаграмме. Хороший знак: загрузка процессора и сети небольшая, несмотря на относительно тяжелую работу по выполнению сетевого вызова для запроса больших фрагментов данных и вывода их на экран.

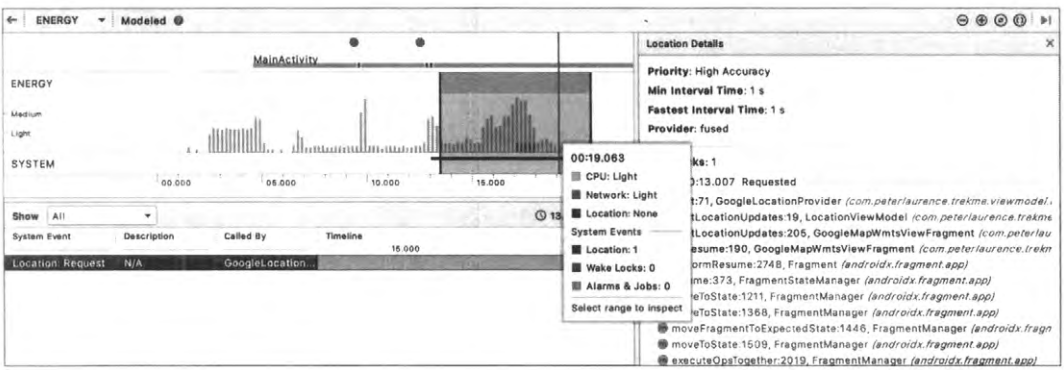


Рис. 11.23. Панель системных событий

Во второй половине описаны виды системных событий, перехваченных с устройства. Energy Profiler фиксирует определенные виды системных событий и их энергопотребление на устройстве:

- **Alarms & Jobs** (Оповещения и задания) — это системные события, предназначенные для "пробуждения" устройства в указанное время. В качестве лучшей

практики Android теперь рекомендует по возможности использовать *WorkManager* или *JobScheduler*, особенно для фоновых задач;

- в запросах о *местоположении* (**Location**) используется GPS-датчик Android, который может потреблять много энергии. Рекомендуется убедиться, что точность и частота измеряются правильно.

Хотя на рис. 11.23 показан только один запрос, существуют и другие типы системных событий, которые содержат свой уникальный набор состояний. Событие запроса может иметь состояние *Active*, *Requested* или *Request Removed*. Также, если Energy Profiler регистрирует системное событие типа *Wake Lock*, временна́я шкала может отображать состояние (состояния) в течение этого события, например *Acquired*, *Held, Released* и т. д.

При выборе определенного системного события открывается правая панель Energy Profiler для просмотра дополнительных сведений. Отсюда можно перейти непосредственно к исходному коду конкретного запроса местоположения. В TrekMe есть класс `GoogleLocationProvider`, который каждую секунду проверяет местоположение пользователя. Это не обязательно может быть проблемой — такая проверка предназначена для того, чтобы устройство постоянно обновляло ваше местоположение, что подтверждает мощь данного инструмента профилирования: можно получить точную информацию, не заглядывая в исходный код. Запросы выполняются по одному, а уже существующие запросы удаляются, чтобы создать новый после загрузки нового блока изображений.

По сравнению с проверкой местоположения можно ожидать снижения уровня энергопотребления, когда пользователь увеличивает карту. Запросов на загрузку больших блоков данных нет. Мы ожидаем некоторого энергопотребления при отслеживании местоположения пользователя, где также используется класс `GoogleLocationProvider`.

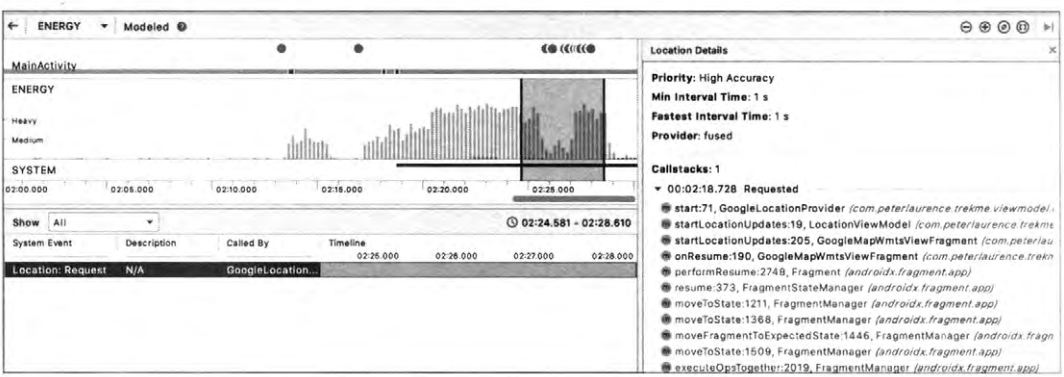


Рис. 11.24. TrekMe открывает и увеличивает существующую карту

На рис. 11.24 видны чрезмерные и быстрые события касания, обозначенные круглыми точками над составной диаграммой. Поскольку TrekMe скачало всю необходимую информацию, в настоящее время сетевые вызовы не выполняются. Тем не

менее мы замечаем, что загрузка процессора возвращается к высоким показателям. Чтобы не перегружать систему, рекомендуется ограничивать события касания, дабы избежать дублирования функций масштабирования.

До сих пор мы рассматривали оценку производительности с точки зрения вычислительной мощности. Но изучение загрузки аккумулятора или процессора не всегда позволяет диагностировать проблемы, связанные с производительностью. Иногда медленное поведение может быть связано с засорением памяти. В следующем разделе мы исследуем взаимосвязь между процессором и памятью и воспользуемся инструментом Memory Profiler для записи данных в виде GPX-файла в TrekMe.

Memory Profiler

В TrekMe можно перейти к разделу *GPX Record*. GPX (GPS eXchange Format) — это свободный текстовый формат хранения и обмена данными GPS на основе XML. Участники похода могут щелкнуть по значку воспроизведения в разделе **Control** (Управление). Тогда приложение будет отслеживать и записывать перемещения участников и их устройств, которые можно сохранить в виде GPX-файла. Позже он будет отображен в виде линейного рисунка, чтобы указать пройденный путь. Эта функция показана на рис. 11.25.

Рис. 11.25. При записи данных в виде GPX-файла в TrekMe используется `GpxRecordingService` для отслеживания GPS-координат пользователя в походе

Мы знаем, что использование местоположения в системе *может* быть тяжким бременем для процессора. Но иногда замедление может быть связано с проблемами в памяти. Процессор использует ОЗУ в качестве емкости для рабочей области, поэтому, когда ОЗУ переполнено, система Android должна выполнить дамп кучи. Если использование памяти сильно ограничено, то и возможность одновременного выполнения большого количества задач становится ограниченной. Чем больше

времени требуется для выполнения меньшего количества операций приложения, тем медлительней становится операционная система. Оперативная память распределяется между всеми приложениями: если слишком много приложений потребляют слишком много памяти, это может снизить производительность устройства или, что еще хуже, вызвать сбои, генерирующие исключения `OutOfMemoryException`.

Memory Profiler позволяет увидеть объем потребляемой памяти, выделенной для запуска приложения. С его помощью можно вручную инициировать создание дампа кучи в работающем сеансе, чтобы произвести анализ с целью определить, какие объекты хранятся в куче и сколько их.

Как показано на рис. 11.26, Memory Profiler предлагает мощные функции:

- запуск сбора мусора;
- захват дампа кучи Java;
- отслеживание выделения памяти;
- интерактивная временна́я шкала фрагментов и активностей, доступных в приложении Android;
- события пользовательского ввода;
- отображение количества оперативной памяти для разделения памяти на категории.

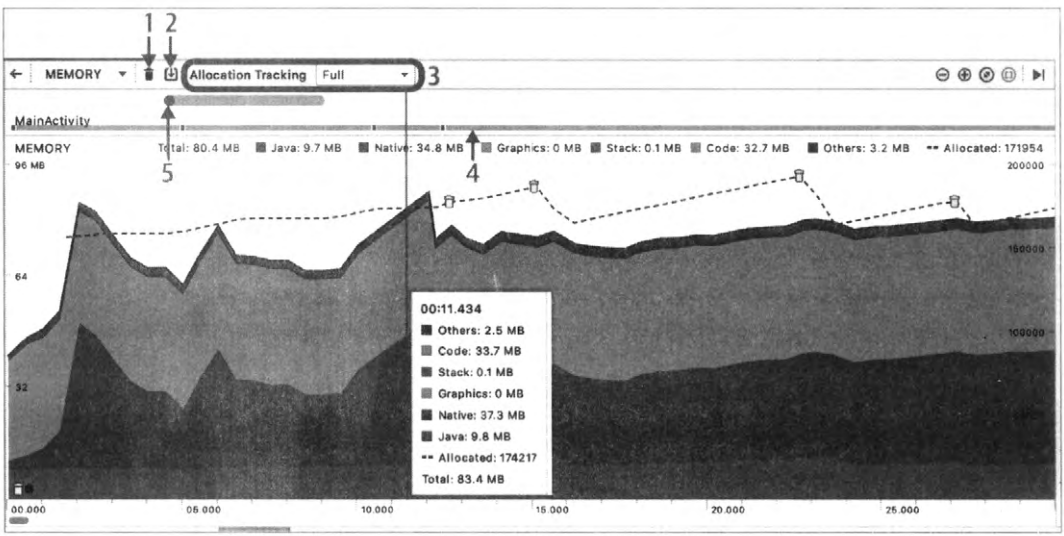


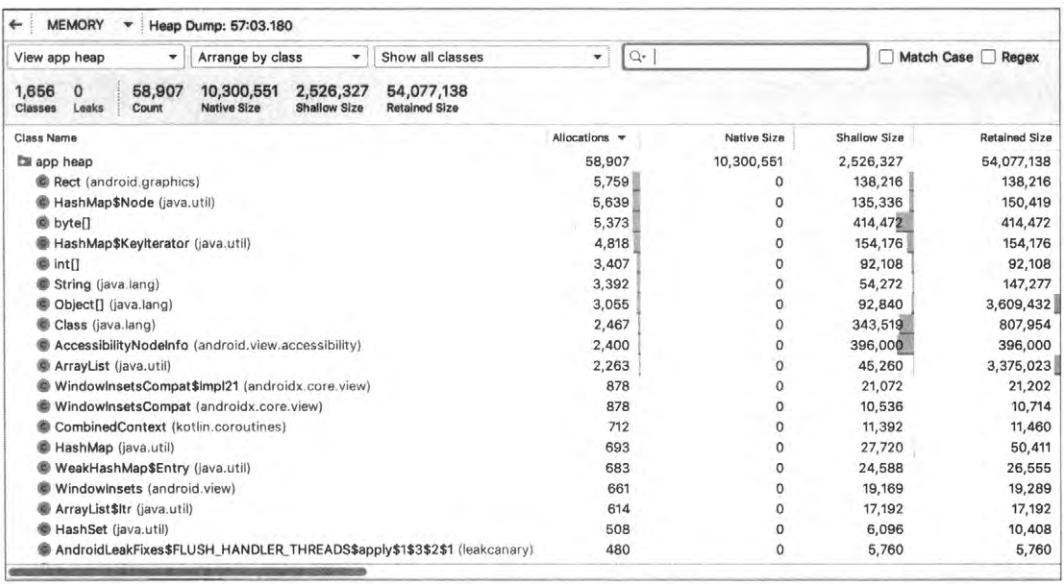
Рис. 11.26. Отслеживание выделения памяти предлагает конфигурацию Full Italicized Text, которая фиксирует выделение памяти для всех объектов, в то время как конфигурация Sampled записывает объекты через равные промежутки времени

Подобно записи сэмплов и трассировок в CPU Profiler, при захвате дампов кучи Java результаты сохраняются на панели сеансов в Android Profiler для сравнения срока службы экземпляра Android Studio.

Инициирование слишком большого количества операции по сбору мусора может повлиять на производительность: например, если оно огромное, то это может замедлить работу устройства, в зависимости от того, как часто и насколько много выделяется памяти под объекты разных поколений. Как минимум, разработчики Android должны попытаться запустить профилирование памяти для каждого приложения, чтобы убедиться, что в куче после ее использования ничего не хранится. В противном случае вы столкнетесь с явлением, известным как "утечка памяти". Обнаружение утечки памяти может спасти жизнь, особенно пользователям Android, в зависимости от более длительного времени автономной работы. То, что вы сейчас увидите, представляет собой разновидность распространенной ошибки управления памятью, которую часто допускают разработчики при работе со службами: случайное оставление службы в режиме выполнения.

TrekMe использует службу переднего плана для получения статистических данных по походу, что является естественным выбором для отслеживания местоположения пользователя. Службы, как и другие компоненты Android, запускаются в UI-потоке приложения. Однако постоянно работающие службы, как правило, истощают аккумулятор и системные ресурсы. Следовательно, важно ограничить использование служб переднего плана, чтобы не ухудшать общую производительность устройства, и останавливать их выполнение как можно скорее, если приложение должно их использовать.

Можно запустить пару записей данных в формате GPX-файла в Memory Profiler и активировать дамп кучи, чтобы увидеть, какие объекты, хранящиеся в куче, потребляют больше всего памяти (рис. 11.27).



Class Name	Allocations	Native Size	Shallow Size	Retained Size
app heap	58,907	10,300,551	2,526,327	54,077,138
Rect (android.graphics)	5,759	0	138,216	138,216
HashMap$Node (java.util)	5,639	0	135,336	150,419
byte[]	5,373	0	414,472	414,472
HashMap$KeyIterator (java.util)	4,818	0	154,176	154,176
int[]	3,407	0	92,108	92,108
String (java.lang)	3,392	0	54,272	147,277
Object[] (java.lang)	3,055	0	92,840	3,609,432
Class (java.lang)	2,467	0	343,519	807,954
AccessibilityNodeInfo (android.view.accessibility)	2,400	0	396,000	396,000
ArrayList (java.util)	2,263	0	45,260	3,375,023
WindowInsetsCompat$Impl21 (androidx.core.view)	878	0	21,072	21,202
WindowInsetsCompat (androidx.core.view)	878	0	10,536	10,714
CombinedContext (kotlin.coroutines)	712	0	11,392	11,460
HashMap (java.util)	693	0	27,720	50,411
WeakHashMap$Entry (java.util)	683	0	24,588	26,555
WindowInsets (android.view)	661	0	19,169	19,289
ArrayList$Itr (java.util)	614	0	17,192	17,192
HashSet (java.util)	508	0	6,096	10,408
AndroidLeakFixes$FLUSH_HANDLER_THREADS$apply$1$3$2$1 (leakcanary)	480	0	5,760	5,760

Рис. 11.27. Можно использовать сочетание клавиш <Ctrl>+<F> для поиска `GpxRecordingService`, чтобы сузить результаты

Дамп кучи показывает список классов, которые можно упорядочить по *выделению памяти* для кучи, *системному*, *поверхностному* или *удерживаемому размеру*. Поверхностный размер — это общий объем используемой памяти Java. Системный размер — это общая память, используемая в системной памяти. Удерживаемый размер состоит из поверхностного и удерживаемого размеров (в байтах).

В записанном дампе кучи можно упорядочить запись выделения памяти по *куче приложения*, *куче образа* или *куче Zygote*. Куча Zygote обозначает память, выделенную для процесса Zygote, которая может включать в себя общий код и ресурсы фреймворка. Куча образа хранит выделение памяти из самой операционной системы и содержит ссылки на классы, используемые в образе, который содержит наше приложение для загрузки системы. Нас больше интересует куча приложения. Это основная куча, которой приложение выделяет память.

В Memory Profiler при запуске дампа кучи отображается список объектов, которые все еще хранятся в памяти после сбора мусора. Этот список может дать вам:

- каждый экземпляр выбранного объекта, отображаемый на панели **Instance View** (Представление "Экземпляр"), с параметром **Jump to Source** (Перейти к исходнику) в коде;
- возможность просматривать данные экземпляра, щелкнув на объекте правой кнопкой мыши в разделе **References** (Ссылки) и выбрав **Go to Instance** (К экземпляру).

Помните, что утечка памяти происходит, когда кэширование содержит ссылки на ненужные объекты. На рис. 11.28 мы ищем "местоположение" с тем же дампом кучи, чтобы найти службу и иметь возможность просмотреть общее выделение памяти. Похоже, что в памяти хранится несколько экземпляров `LocationService`, хотя работать должен только один.

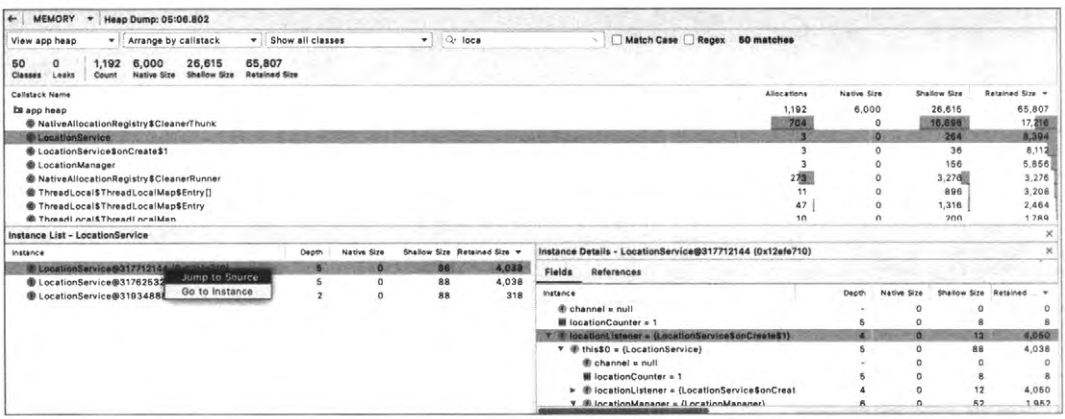


Рис. 11.28. В памяти хранится подозрительное количество экземпляров `LocationService`

Похоже, что каждый раз, когда мы нажимаем кнопку **Record** (Запись), в TrekMe создается новый экземпляр `LocationService`, который затем хранится в памяти даже после того, как служба прекращает работу. Вы можете запустить и остановить

службу, но если сохранить ссылку на нее в фоновом потоке, даже если эта служба уже не функционирует, экземпляр по-прежнему останется в куче даже после сбора мусора.

Просто запустим еще пару записей в TrekMe, чтобы подтвердить поведение, которое мы подозреваем. Можно щелкнуть правой кнопкой мыши на одном из этих экземпляров, чтобы перейти к исходнику. В *RecordingViewModel.kt* мы видим следующий код:

```
fun startRecording() {
    val intent = Intent(app, LocationServices::class.java)
    app.startService(intent)
}
```

Мы хотим проверить, действительно ли эти службы останавливаются, прежде чем запустить новую. Запускаемая служба остается активной до тех пор, пока это возможно: до тех пор, пока мы не вызовем функцию `stopService` за пределами службы или пока не будет вызвана функция `stopSelf` в самой службе. Это делает использование постоянно работающих служб затратным, т. к. Android считает, что запускаемые службы используются всегда, а это означает, что память, которую служба расходует в ОЗУ, так и не будет доступна.

Когда запись данных в формате GPX останавливается, `LocationService` передает серию событий, отслеживая местоположение по GPS, которое затем записывается и сохраняется в виде набора данных. Если файл GPX только что был записан, то служба подписывается на основной поток для отправки состояния. Поскольку `LocationService` расширяет службу, можно вызвать метод `Service::stopSelf`, чтобы остановить ее:

```
@Subscribe(threadMode = ThreadMode.MAIN)
fun onGpxFileWriteEvent(
   event: GpxFileWriteEvent
) {
   mStarted = false
   sendStatus()
   stopSelf()   // <--- служба будет остановлена,
                // и будет выпущена ссылка на сборщик мусора
}
```

Можно использовать Memory Profiler и проверить дамп кучи, чтобы убедиться, что мы храним ссылку только на одну службу. На самом деле, поскольку записи данных в формате GPX выполняются через `LocationService`, имеет смысл остановить службу, когда пользователь прекращает запись. Таким образом, при сборе мусора можно освободить память, занимаемую службой, в противном случае экземпляр `LocationService` по-прежнему будет храниться в куче.

Memory Profiler может помочь обнаружить вероятные утечки памяти в процессе просеивания дампа кучи. Вы также можете отфильтровать дамп, установив флажок **Activities/Fragments Leaks** в настройках дампа. Поиск утечек памяти может... осу-

ществляться вручную, и даже в этом случае самостоятельный поиск — это лишь один из способов перехватить их. К счастью, у нас есть LeakCanary — популярная библиотека для обнаружения утечек памяти, которая может подключаться к приложению в режиме отладки и наблюдать за ними.

Обнаружение утечек памяти с помощью LeakCanary

Во время выполнения LeakCanary автоматически обнаруживает явные и неявные утечки памяти, которые может быть трудно обнаружить вручную. Это большое преимущество, поскольку Memory Profiler требует ручного запуска дампа кучи и проверки на предмет наличия удерживаемой памяти. Когда аналитика сбоев не может обнаружить сбои, вызванные `OutOfMemoryException`, LeakCanary служит жизнеспособной альтернативой для отслеживания проблем, обнаруженных во время выполнения, и обеспечивает лучший охват при обнаружении утечек памяти.

Утечки памяти обычно возникают из-за ошибок, связанных с жизненным циклом объектов, которые не исчезают, после того как были использованы. LeakCanary может обнаруживать различные ошибки, например:

- создание нового экземпляра `Fragment` без предварительного удаления существующей версии;
- неявное или явное внедрение ссылки на `Activity` или `Context` в компонент, который не относится к Android;
- регистрация слушателя, приёмника широковещательных сообщений или подписки RxJava, когда вы забываете избавиться от слушателя или подписчика в конце жизненного цикла родителя.

В этом примере мы установили библиотеку LeakCanary в TrekMe. LeakCanary органично используется в разработке, пока не будет сохранен дамп кучи с потенциальными утечками. Можно установить LeakCanary, добавив в Gradle следующую зависимость:

```
debugImplementation 'com.squareup.leakcanary:leakcanary-android:2.*'
```

После установки в приложение LeakCanary автоматически обнаруживает утечки при удалении экземпляра `Activity` или `Fragment`, очищает `ViewModel` и т. д. Это достигается путем обнаружения сохраненных объектов, прошедших через `ObjectWatcher`. Затем LeakCanary создает дамп кучи, анализирует его и классифицирует эти утечки для удобства потребления. После установки LeakCanary приложение можно использовать как обычно. Если LeakCanary обнаруживает сохраненные экземпляры в дампе кучи, то отправляет уведомление на панель задач.

В случае с TrekMe похоже, что LeakCanary обнаружил утечку памяти в экземпляре `RecyclerView` класса `MapImportFragment` (рис. 11.29).

Сообщение об ошибке говорит нам, что экземпляр `RecyclerView` "протекает". LeakCanary указывает, что этот экземпляр содержит ссылку на экземпляр `Context`, который оборачивает активность. В случае с экземпляром `RecyclerView` что-то пре-

пятствует сбору мусора — либо неявная, либо явная ссылка на экземпляр RecyclerView, переданный компоненту, существующему дольше, чем активность.

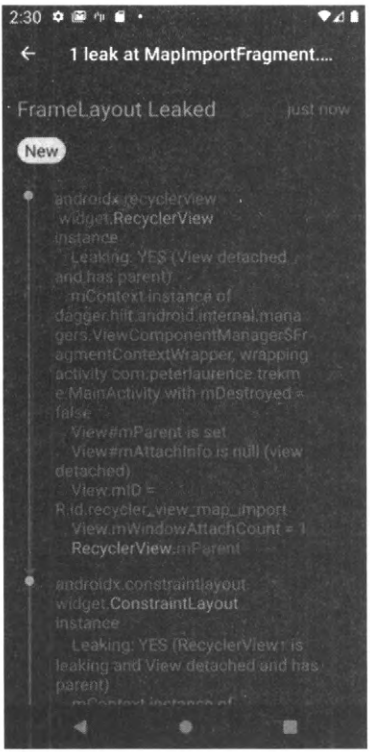


Рис. 11.29. LeakCanary показывает утечку памяти в экземпляре RecyclerView

Мы еще не совсем уверены, с чем имеем дело, поэтому начнем с изучения класса *MapImportFragment.kt*, содержащего RecyclerView (см. рис. 11.29). Возвращаясь к элементу пользовательского интерфейса recyclerViewMapImport, на который ссылаются из файла макета, хотим обратить ваше внимание на нечто любопытное:

```
class MapImportFragment: Fragment() {

    private val viewModel: MapImportViewModel by viewModels()

    /* удалено для краткости */
    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
    /* удалено для краткости */
        recyclerViewMapImport.addOnItemTouchListener(
            RecyclerItemClickListener(
                this.context,                                    ❶
                recyclerViewMapImport,
                object: RecyclerItemClickListener.onItemClickListener {
```

```
                override fun onItemClick(view: View, position: Int) {
                    binding.fab.activate()
                    single.fab(position)
                }
            })
        )
    }

    /* удалено для краткости */

    private fun FloatingActionButton.activate() {
        /* удалено для краткости */
        fab.setOnClickListener {
            itemSelected?.let { item ->
                val inputStream = context.contentResolver.
                    openInputStream(item.url)
                inputStream?.let {
                    viewModel.unarchiveAsync(it, item)              ❷
                }
            }
        }
    }
}
```

❶ В `MapImportFragment` мы привязываем собственного обработчика события щелчка по кнопке к каждому объекту `ViewHolder` в `RecyclerView`.

❷ Затем используется `Context` для получения `ContentResolver` и создания `InputStream` в качестве аргумента для `MapImportViewModel::unarchiveAsync`.

Когда пользователь щелкает на определенном элементе в `RecyclerView`, вызывается функция-расширение `FloatingActionButton::activate`. Помните, что часто причиной утечки памяти является случайное внедрение активности или контекста в компонент, который не относится к Android.

Если внимательно посмотреть на реализацию `FloatingActionButton::activate`, то можно увидеть, что мы создаем неявную ссылку на вмещающий класс, который является экземпляром `MapImportFragment`.

Как создается неявная ссылка? Мы добавляем обработчика события щелчка к кнопке. Обработчик содержит ссылку на родительский контекст (возвращаемый методом фрагмента, `getContext()`).

Чтобы получить доступ к контексту из обработчика, компилятор Kotlin создает неявную ссылку на вмещающий класс.

Изучая код класса MapImportViewModel, мы видим, что InputStream передается вниз, чтобы иметь возможность вызывать другой приватный метод в ViewModel:

```
class MapImportViewModel @ViewModelInject constructor(
    private val settings: Settings
) : ViewModel() {
    /* удалено для краткости */

    fun unarchiveAsync(inputStream: InputStream, item: ItemData) {
        viewModelScope.launch {
            val rootFolder = settings.getAppDir() ?: return@launch
            val outputFolder = File(rootFolder, "imported")
            /* удалено для краткости */
        }
    }
}
```

У объекта ViewModel есть собственный жизненный цикл. Он должен существовать дольше представления, к которому он привязан, пока Fragment не будет отсоединен. Вместо того чтобы использовать InputStream в качестве аргумента, лучше воспользоваться контекстом приложения, который доступен на протяжении всего жизненного цикла приложения и который можно внедрить через конструктор внедрения параметров в MapImportViewModel[2]. После этого можно создать InputStream прямо в MapImportViewModel::unarchiveAsync:

```
class MapImportViewModel @ViewModelInject constructor(
    private val settings: Settings,
    private val app: Application
): ViewModel() {
    /* удалено для краткости */

    fun unarchiveAsync(item: ItemData) {
        viewModelScope.launch {
            val inputStream = app.contentResolve.
                openInputStream(item.uri) ?: return@launch
            val rootFolder = settings.getAppDir() ?: return@launch
            val outputFolder = File(rootFolder, "imported")
            /* удалено для краткости */
        }
    }
}
```

[2] @ViewModelInject — специальная аннотация из фреймворка внедрения зависимостей Hilt. Однако внедрения параметров конструктора также можно добиться с помощью ручного внедрения зависимостей или фреймворков, таких как Dagger и Koin.

Конечно, включение LeakCanary может помешать разработке, если в существующем приложении много утечек памяти. В этом случае у вас может возникнуть искушение отключить LeakCanary, чтобы не прерывать текущую работу. Если вы решите включить LeakCanary в приложение, лучше всего делать это, только когда вы и ваша команда готовы разбираться с последствиями.

Резюме

Нет никаких сомнений в том, что для Android существуют мощные инструменты тестирования производительности и профилирования. Чтобы убедиться, что ваше приложение получает максимальную отдачу от аналитики, лучше всего выбрать один или два инструмента в зависимости от ситуации. В мире оптимизаций легко заблудиться, но важно помнить, что самые большие победы приносят оптимизации с наименьшими усилиями и наибольшим воздействием. Точно так же важно учитывать текущие приоритеты и рабочую нагрузку команды.

Подходите к оптимизации как диетолог, поощряя постепенные, привычные изменения вместо "аварийной диеты". Профилирование предназначено для того, чтобы показать вам, что на самом деле происходит "под капотом", но важно помнить, что обычный разработчик должен расставлять приоритеты, какие проблемы необходимо решать в мире, где время и рабочие ресурсы могут быть ограничены.

Мы надеемся, что вы почувствуете себя более подготовленным к устранению любых потенциальных ошибок, которые могут возникнуть у вас на пути, и что эта глава придаст вам уверенности для того, чтобы вы приступили к изучению некоторых из этих инструментов в собственных приложениях, дабы увидеть, как все работает "под капотом".

- Android Profiler — мощный инструмент для анализа производительности приложений, начиная от сетевых взаимодействий и процессора и заканчивая анализом памяти и энергопотребления. Android Studio кэширует записанные сеансы, а также дампы кучи и трассировки методов на протяжении всего срока службы экземпляра Android Studio, чтобы вы могли сравнить их с другими сохраненными сеансами.
- Network Profiler способен помочь решить проблемы, связанные с отладкой API. Он может предоставить информацию, полезную как для клиентского устройства, так и для сервера, откуда поступают данные, и обеспечить оптимальное форматирование данных в рамках сетевого вызова.
- CPU Profiler может дать представление о том, где больше всего времени тратится на выполнение методов, и особенно полезен для поиска проблем производительности. Вы можете записывать различные виды трассировок, чтобы иметь возможность подробно изучить определенные потоки и стеки вызовов.

- Energy Profiler проверяет, могут ли процессы ЦП, сетевые вызовы или местоположения по GPS-координатам в приложении разряжать аккумулятор устройства.
- Memory Profiler смотрит, сколько памяти выделено в куче. Он способен помочь получить представление об областях кода, которые могут использовать улучшения в памяти.
- LeakCanary — популярная библиотека с открытым исходным кодом, созданная компанией Square. Она может пригодиться для обнаружения утечек памяти, которые сложно обнаружить во время выполнения.

ГЛАВА 12

Снижение потребления ресурсов за счет оптимизации производительности

В предыдущей главе вы узнали, как изучить то, что происходит "под капотом", с помощью популярных инструментов профилирования для Android. В этой заключительной главе освещается большое количество соображений относительно оптимизации производительности. Универсального подхода не существует, поэтому полезно знать о потенциальных проблемах, связанных с производительностью (и их решениях).

Однако иногда эти вопросы могут быть результатом большого числа сложностей, и, если рассматривать их по отдельности, может показаться, что они не заслуживают внимания.

Вопросы производительности позволяют изучить проблемы, которые могут повлиять на способность приложения к масштабированию. Если вы можете использовать любую из этих стратегий в качестве "легкой добычи" в коде, стоит добиваться самой крупной победы с наименьшими усилиями. Не все разделы этой главы подходят для каждого проекта, над которым вы работаете, но тем не менее они полезны при написании любого приложения для Android. Это разные темы: от просмотра оптимизации производительности системы до формата сетевых данных, кэширования и многого другого.

Мы знаем, что система представлений должна быть заменена на Jetpack Compose: однако она никуда не денется в течение многих лет, даже при наличии Jetpack. Первая половина этой главы посвящена темам, которые могут принести пользу каждому проекту: потенциальная оптимизация системы представлений. Способ настройки иерархии представлений в конечном счете может оказать существенное влияние на производительность, если не проявить осторожность. По этой причине мы рассматриваем два простых способа оптимизации производительности представлений: уменьшение сложности иерархии представлений с помощью класса `ConstraintLayout` и создание графических ресурсов для анимационного или специального фона.

Достижение плоской иерархии представлений с помощью ConstraintLayout

Как правило, нам нужно, чтобы иерархия представлений в Android была как можно более плоской. Глубоко вложенные иерархии влияют на производительность как при первом развертывании представления, так и при взаимодействии пользователя с экраном. Когда иерархии представлений глубоко вложены, может потребоваться больше времени, чтобы отправить инструкции обратно в корневой `ViewGroup`, содержащий все ваши элементы, и вернуться назад, чтобы внести изменения в определенные представления.

Помимо инструментов профилирования, о которых шла речь в *главе 11*, Android Studio предлагает *Layout Inspector*, который анализирует приложение во время выполнения и создает трехмерное отображение элементов представления, расположенных на экране. Чтобы открыть Layout Inspector, щелкните кнопкой мыши по одноименной вкладке в Android Studio в правом нижнем углу (рис. 12.1).

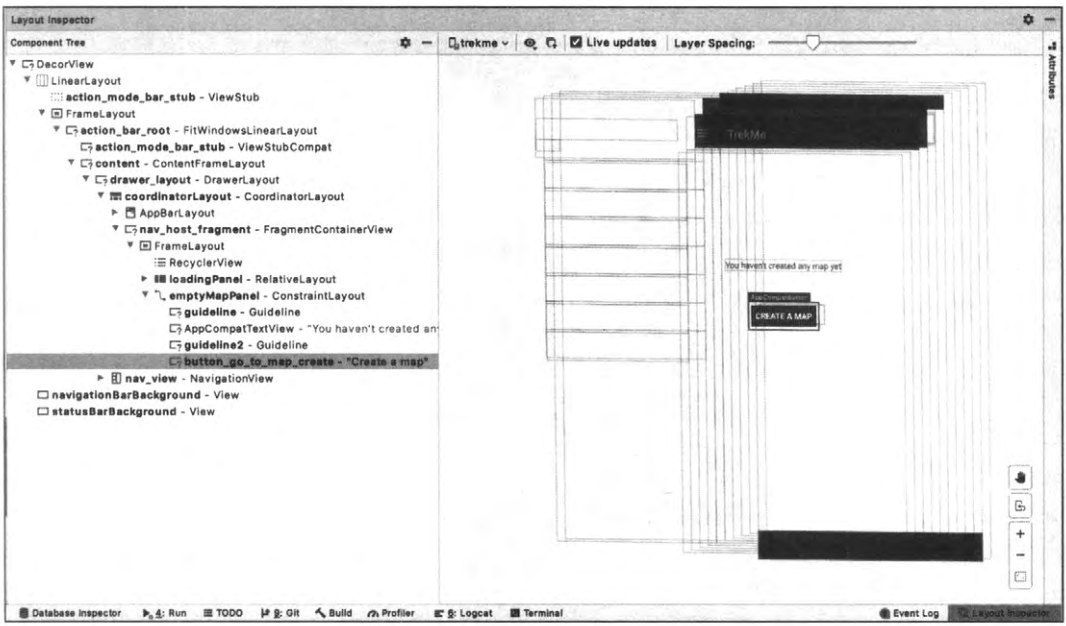


Рис. 12.1. Layout Inspector использует трехмерное отображение на устройствах, работающих под управлением Android 10 и выше

При отрисовке дочерние компоненты отображаются поверх родительского представления, накладываясь друг на друга. Слева Layout Inspector предоставляет панель **Component Tree** (Дерево компонентов), поэтому элементы можно развернуть и изучить их свойства. Чтобы лучше понять, что происходит, когда пользователи взаимодействуют с виджетами пользовательского интерфейса Android, на рис. 12.2 показана та же самая иерархия, представленная в дереве компонентов.

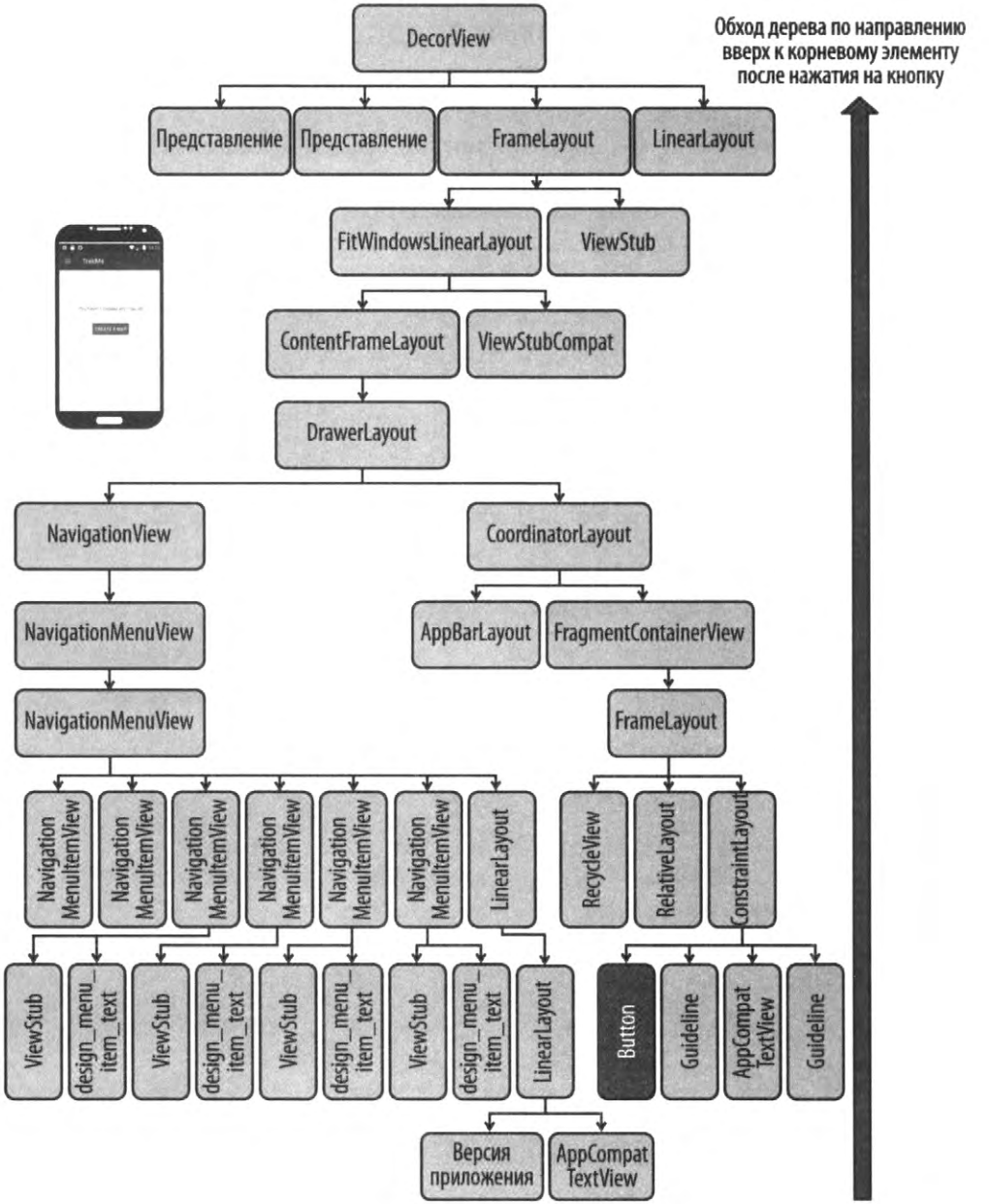


Рис. 12.2. Элементы работающей активности во всей своей полноте

Даже при относительно простом макете иерархия представлений может довольно быстро усложняться. Управление множеством вложенных макетов может быть сопряжено с дополнительными затратами, такими как повышенная сложность управления событиями касаний, более медленный рендеринг на графическом процессоре и трудности с обеспечением одинакового интервала или размера представлений на экранах разного размера.

Помимо визуальных изменений, которые может потребовать приложение, система Android сама по себе также может влиять на свойства представления. Изменения в свойствах представления, вызванные вами или операционной системой, могут привести к изменению макета иерархии представлений. Произойдет это или нет, зависит от способа реализации представлений (самостоятельно или с помощью внешней зависимости), того, как часто компоненты макета вызывают изменение размеров, и их расположения в иерархии представлений.

Нужно не только беспокоиться о сложности иерархии, но также не забывать избегать определенных типов представлений, которые в конечном счете могут стоить приложению в два раза больше обходов, необходимых для отправки инструкций в Android. При активировании относительного позиционирования некоторые старые типы макетов подвержены "двойному налогообложению".

`RelativeLayout`

В обязательном порядке всегда обходит свои дочерние элементы по крайней мере дважды: один раз для расчета макета для каждой позиции и размера и один раз для завершения позиционирования.

`LinearLayout`

Задает горизонтальную ориентацию или значение `true` для `android:setMeasureWithLargestChildEnabled` в вертикальной ориентации; в обоих случаях совершаются два прохода для каждого дочернего элемента.

`GridLayout`

Может привести к двойному обходу, если макет использует распределение веса или задает для `android:layout_gravity` любое допустимое значение.

Стоимость "двойного налогообложения" может стать гораздо более серьезной при расположении ближе к корню дерева и даже привести к экспоненциальному обходу. Чем глубже иерархия представлений, тем больше времени требуется для обработки событий ввода и соответствующего обновления представлений.

Полезно снижать негативное влияние изменения макета представления на скорость отклика приложения. Чтобы сделать иерархию более плоской и надежной, Android рекомендует использовать класс `ConstraintLayout`. Он помогает создавать адаптивный пользовательский интерфейс для сложных макетов с плоской иерархией.

Существует несколько правил `ConstraintLayout`, которые следует помнить:

- у каждого представления должно быть хотя бы одно ограничение по горизонтали и одно по вертикали;
- начало или конец представления могут быть связаны только с началом или концом других представлений;
- верхняя или нижняя часть представления может быть связана только с верхней или нижней частью других представлений.

На рис. 12.3 показано, как родитель привязывает представление к назначенному концу экрана в Android Studio.

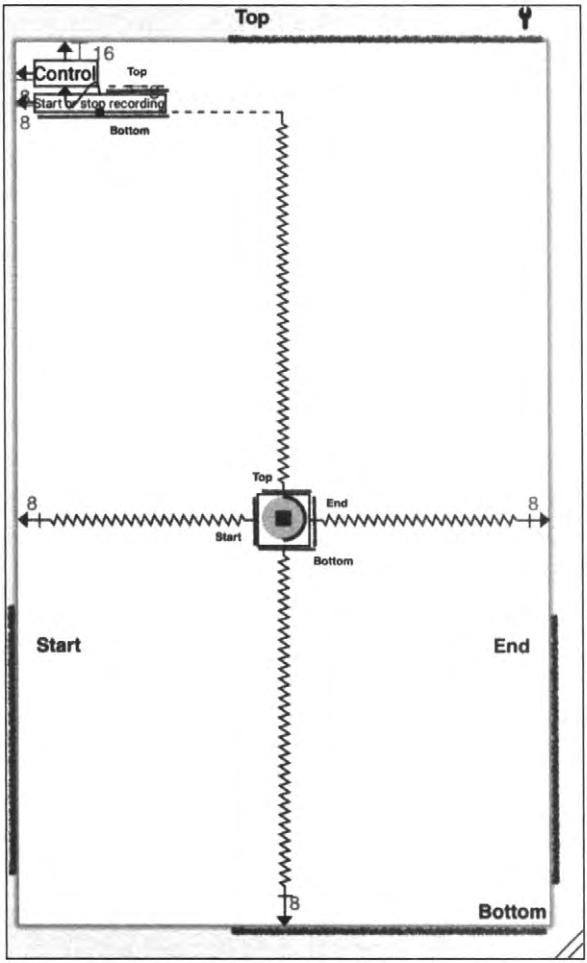


Рис. 12.3. В этом конкретном `ConstraintLayout` кнопка для указания состояния загрузки компонента или страницы ограничивает все стороны родителя центром экрана. Текстовые элементы в левом верхнем углу ограничены только левой и верхней сторонами родителя

При выделении на представлении появляются зигзагообразные линии, указывающие на ограничение стороны. Зигзагообразная линия указывает на одностороннее ограничение представления, а волнистая — на то, что два представления ограничивают друг друга.

В этой книге не рассматриваются дополнительные полезные функции `ConstraintLayout`, например барьеры, направляющие, группы и создание ограничений. Лучший способ познакомиться с ними — это самостоятельно экспериментировать с элементами в режиме разделения экрана (рис. 12.4).

Использование `ConstraintLayout`, особенно когда элементы `ViewGroup` могут быть глубоко вложенными или неэффективными, — простой способ устранить потенциальные проблемы, связанные с производительностью, во время выполнения для любо-

го приложения Android. В следующем разделе мы сместим акценты и поговорим об анимации.

Рис. 12.4. Кнопка **Split** в Android Studio, обеспечивающая режим разделения экрана, при котором можно работать с макетом, видя при этом код

Сокращение количества операций рисования с помощью экземпляров класса *Drawable*

Еще одна потенциальная проблема, связанная с производительностью любого Android-проекта — это операции рисования программными средствами во время выполнения. Время от времени разработчики сталкиваются с элементом представления, который не имеет доступа к определенным свойствам в файле макета. Предположим, вы хотите отобразить представление только с двумя верхними закругленными углами. Один из способов — воспользоваться функцией-расширением Kotlin:

```
fun View.roundCorners(resources: Resources, outline: OutLine?) {
    val adjusted = TypedValue.applyDimension(
        TypedValue.COMPLEX_UNIT_SP,
        25,
        resources?.displayMetrics
    )

    val newHeight = view.height.plus(cornerRadiusAdjusted).toInt()
    this.run { outline?.setRoundRect(0, 0, width, newHeight, adjusted)}
}
```

Это прекрасный и допустимый код; однако слишком большое количество операций рисования может в конечном счете "придушить" RenderThread и впоследствии заблокировать возможность UI-потока обрабатывать дальнейшие события до тех пор, пока операции рисования во время выполнения не будут завершены. Кроме того, затраты на изменения представлений программными средствами становятся выше, если размер конкретного представления необходимо изменить, чтобы оно соответствовало ограничениям. Изменение размера элемента представления во время выполнения означает, что вы не сможете использовать класс `LayoutInflater` для настройки соответствия элементов новым размерам исходного измененного представления.

Можно освободить себя от затрат, которые возникли бы в противном случае, используя рисованные элементы, хранящиеся в папке *res/drawable*. В следующем ко-

де показано, как получить два закругленных верхних угла элемента представления с помощью изображения, сформированного на основе разметки XML:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
    android:shape = "rectangle">
    <corners android:topLeftRadius="25dp" android:topRightRadius="25dp"/>
    <stroke android:width="1dp" android:color="#FFF"/>
    <solid android:color="#FFF"/>
</shape>
```

Затем можно добавить имя файла в качестве типа `Drawable` к фоновому атрибуту в файле макета представления:

```
android:background="@drawable/rounded_top_corners_background"
```

В предыдущем разделе мы кратко коснулись начальных этапов того, как пользовательский интерфейс отправляет инструкции в Android. Чтобы понять, где появляется анимация, мы изучим полный процесс отображения интерфейса. Рассмотрим случай, когда пользователь TrekMe нажимает кнопку **Create a Map** (Создать карту).

Этапы, которые мы обсудим в оставшейся части этого раздела, показывают, как операционная система обрабатывает пользовательские события, передаваемые через экран, и выполняет инструкции по рисованию, от программного обеспечения до аппаратного. Мы объясним все этапы вплоть до анимации на этапе синхронизации (рис. 12.5).

Рис. 12.5. Анимация происходит на этапе синхронизации, после выполнения обхода

Вертикальная синхронизация обозначает время между отрисовкой кадров на экране. В приложении, когда пользователь касается элемента представления на экране, происходит *обработка ввода*. На этапе *ввода* Android выполняет вызов, чтобы *перерисовать* все узлы элемента родительского представления, копируя набор инструкций для отслеживания измененного состояния. Так вы не перерисовываете само представление, а скорее указываете системе, какое отмеченное представление должно быть перерисовано позже. Это делается путем передачи скопированной информации вверх по иерархии представлений, чтобы ее можно было выполнить на обратном пути на более позднем этапе. На рис. 12.6 показано, как выглядит перерисовка после того, как пользователь нажал на кнопку: обход вверх по узлу, затем копирование набора инструкций `DisplayList`. Несмотря на то что стрелки повернуты вниз, указывая на дочерние элементы, обход и копирование фактически идет вверх по направлению к корню, прежде чем вернуться вниз.

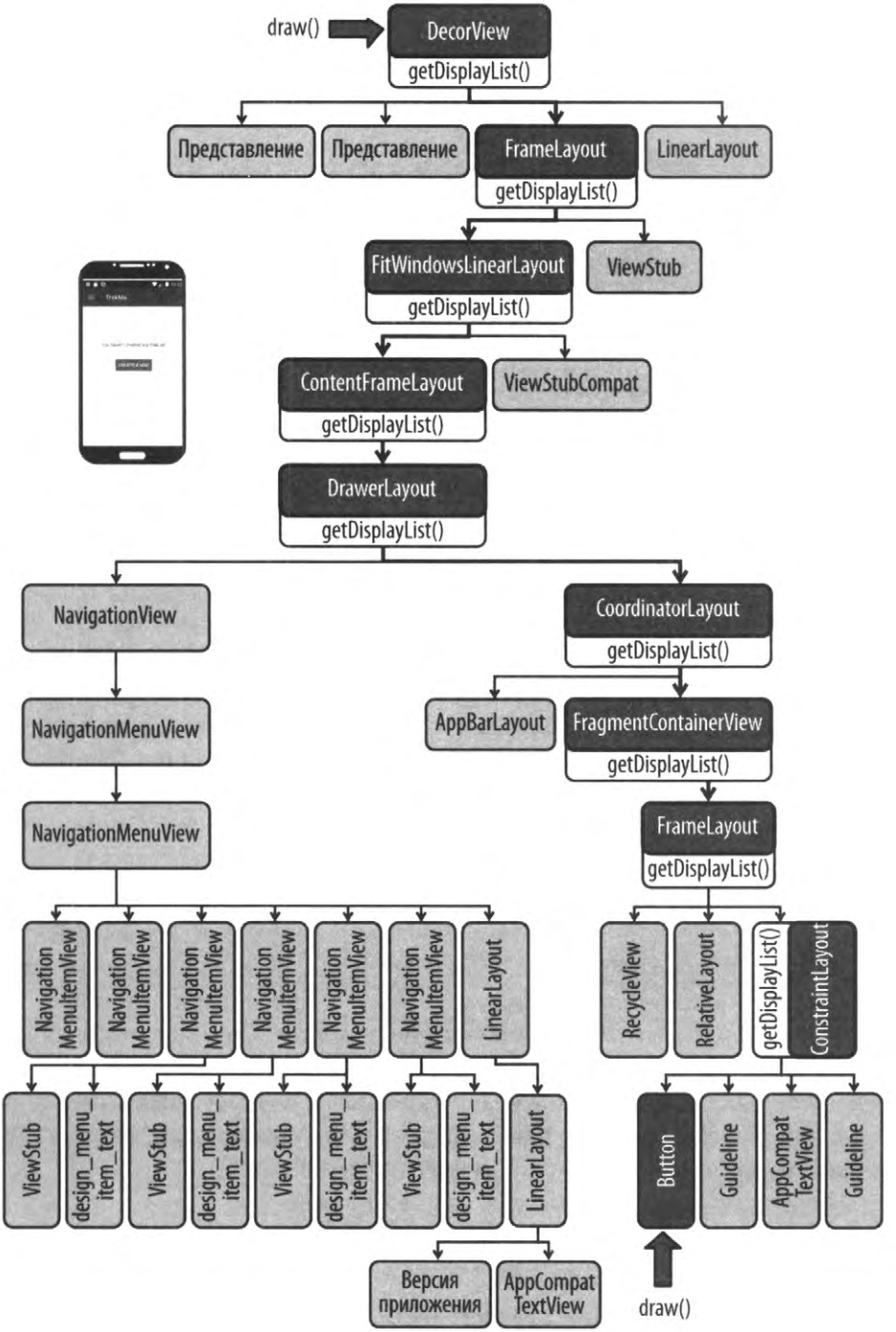


Рис. 12.6. Объект `DisplayList` представляет собой набор компактных инструкций для указания того, какие представления необходимо перерисовать. Эти инструкции копируются для каждого элемента родительского представления в корневую иерархию во время перерисовки, а затем выполняются во время обхода

Затем система пользовательского интерфейса Android планирует следующий этап, известный как *обход*, который содержит собственное подмножество этапов отображения:

Измерение.

Вычисляет `MeasureSpecs` и передает его дочернему элементу для измерения. Это делается рекурсивно, вплоть до крайних узлов.

Макет.

Задает положение представления и размер дочернего макета.

Рисование.

Отображает представления, используя набор инструкций, заданных набором инструкций `DisplayList`.

На следующем этапе — *синхронизации* — Android синхронизирует сведения `DisplayList` между центральным и графическим процессорами. Когда ЦП начинает взаимодействовать с графическим процессором, библиотека Java Native Interface (JNI) берет свой набор инструкций на уровне Java Native в UI-потоке и отправляет синтетическую копию вместе с другой информацией в графический процессор из RenderThread. RenderThread отвечает за анимацию и передачу части работы из UI-потока, чтобы разгрузить его (вместо того, чтобы отправлять работу графическому процессору). Там оба процессора взаимодействуют друг с другом, чтобы определить, какие инструкции должны быть выполнены, а затем визуально объединяются для отображения на экране. Наконец, мы достигаем стадии *выполнения*, где операционная система выполняет операции `DisplayList` оптимизированным способом (например, одновременно выполняя похожие операции). "Drawn Out: How Android Renders"[1] — отличный доклад, в котором подробнее рассказывается об отображении в Android на системном уровне[2].

Начиная с Android Oreo, анимация в виде уменьшающейся или увеличивающейся окружности, в которой отображается часть интерфейса, рябь и векторная анимация, находятся только в `RenderThread`. Это означает, что данные виды анимации не блокируют UI-поток. Их можно создавать с помощью нестандартных рисованных элементов. Рассмотрим случай, когда нам нужно анимировать затененную рябь на фоне представления всякий раз, когда пользователь нажимает некий `ViewGroup`. Для этого можно комбинировать набор рисованных элементов, начав с класса `RippleDrawable`, чтобы создать саму пульсирующую анимацию:

```
<?xml version="1.0" encoding="utf-8"?>
<ripple xmlns:android="http://schemas.android.com/apk/res/android"
        android:color="@color/primary">
    <item android:id="@android:id/mask">
```

[1] См. **https://oreil.ly/P5WbO**.

[2] Haase C., Guy R. Drawn Out: How Android Renders // Google I/O '18, 2017.

```
        <shape android:shape="rectangle">
            <solid android:color="@color/ripple_mask" />
        </shape>
    </item>
</ripple>
```

RippleDrawable, эквивалентом которого в XML является ripple, требуется атрибут цвета для волновых эффектов. Чтобы применить эту анимацию к фону, можно использовать другой файл:

```
<?xml version="1.0" encoding="utf-8"?>
<shape xmlns:android="http://schemas.android.com/apk/res/android"
        android:shape="rectangle">
    <solid android:color="@color/background_pressed" />
</shape>
```

Мы можем использовать DrawableStates — набор состояний, предоставляемых фреймворком, которые можно указать в экземпляре класса Drawable. В данном случае DrawableStates используется для селектора, чтобы определить анимацию, а также с целью определить, происходит анимация при нажатии или нет. Наконец, мы создаем экземпляр класса Drawable, используемый для отображения различных состояний. Каждое состояние представлено дочерним рисованным элементом. В данном случае мы применяем пульсирующую анимацию только при нажатии на представление:

```
<?xml version="1.0" encoding="utf-8"?>
<selector xmlns:android="http://schemas.android.com/apk/res/android"
    android:enterFadeDuration="@android:integer/config_shortAnimTime"
    android:exitFadeDuration="@android:integer/config_shortAnimTime">
    <item
        android:state_pressed="true" android:state_enabled="true"
        android:drawable="@drawable/background_pressed_ripple"/>
    <item
        android:state_pressed="false"
        android:drawable="@android:color/transparent"/>
</selector>
```

Android Jetpack

Как упоминалось в начале главы, система представлений, созданная на базе Jetpack Compose, полностью отличается от системы представлений Android, где есть свои наборы управления пользовательским интерфейсом, графикой, поведением во время выполнения или компиляции и многое другое. Если в Jetpack Compose рисование выполняется программными средствами, означает ли это, что использовать его для этой цели нецелесообразно?

Хотя с XML отображение идет быстрее, чем в самом Compose, в настоящее время ведется оптимизация по сокращению разрыва во времени обработки. Тем не менее нужно иметь в виду, что главное преимущество Compose — это возможность быстро обновлять или выполнять рекомпозицию представления Composable, делая это гораздо эффективнее по сравнению с представлениями Android.

Мы закончили обсуждение оптимизации производительности представлений и в оставшейся части главы перейдем к дополнительным советам по оптимизации производительности различных частей приложения.

Минимизация данных в сетевых вызовах

В Android важно минимизировать данные, чтобы избежать медленных загрузок, разрядки аккумулятора и использования слишком большого объема данных. В предыдущей главе мы начали рассматривать форматы полезных данных. Изображения и сериализованные форматы данных обычно являются основной причиной раздувания кода, поэтому формат данных важно проверять.

Если вам не нужна прозрачность изображений, с которыми вы работаете в проекте, лучше работать с JPG или JPEG, т. к. этот формат изначально не поддерживает прозрачность и сжимается лучше, чем PNG. Когда дело доходит до растровых изображений для миниатюр, вероятно, имеет смысл выводить изображение в гораздо более низком разрешении.

В этой отрасли в сети обычно используется формат JSON. К сожалению, JSON и XML не подходят для сжатия, поскольку формат данных учитывает пробелы, кавычки, символы перевода каретки и многое другое. Двоичные форматы сериализации, например *буферы протоколов*, — это доступный формат данных в Android, который может служить более дешевой альтернативой. Можно определять структуры данных, которые Protobuf способен сжимать намного сильнее, чем данные в формате XML и JSON. Для получения дополнительной информации о буферах протоколов посетите сайт Google Developers[3].

Организация пула и кэширование объектов *Bitmap*

В приложении TrekMe используется организация пула, чтобы избежать выделения памяти для слишком большого количества объектов `Bitmap`. Суть организации пула состоит в повторном использовании уже существующего экземпляра, когда это возможно. Откуда берется этот "существующий экземпляр"? После того как `Bitmap` больше не отображается, вместо того чтобы сделать его доступным для сбора мусора (просто не сохраняя ссылку на него), можно поместить его в "пул объектов

---

[3] См. **https://oreil.ly/6dUL0**.

Bitmap". Это контейнер для доступных объектов Bitmap, которые будут использоваться в последующем. Например, TrekMe использует простое удаление из очереди в памяти в качестве пула. Чтобы загрузить изображение в существующий объект Bitmap, нужно указать, какой экземпляр Bitmap вы хотите использовать. Это можно сделать посредством параметра inBitmap[4] из BitMapFactory.Options:

```
// получаем экземпляр Bitmap из пула
BitmapFactory.Options().inBitmap = pool.get()
```

Стоит отметить, что библиотеки для загрузки изображений, такие как Glide, могут избавить вас от необходимости самостоятельно обрабатывать растровые изображения. Используя эти библиотеки, вы бесплатно кэшируете растровые изображения в своих приложениях. В тех случаях, когда сетевые вызовы выполняются медленно, получение нового экземпляра Bitmap может быть затратным. В такой ситуации выборка из кэша растровых изображений может сэкономить вам массу времени и ресурсов. Если пользователь снова возвращается на экран, экран может загрузиться почти сразу, вместо того чтобы выполнять еще один сетевой запрос. Различают два вида кэширования: кэширование *в памяти* и кэширование на основе *файловой системы*. Кэширование в памяти обеспечивает самое быстрое извлечение объектов за счет использования бóльшего объема памяти. Кэширование на основе файловой системы обычно более медлительное, но занимает меньше памяти. Некоторые приложения используют алгоритм LRU cache в памяти[5], в то время как другие применяют кэширование на основе файловой системы или оба этих подхода.

Например, если в приложении вы выполняете запросы по протоколу HTTP, то можете использовать *OkHttp*, чтобы предоставить доступ к отличному API с целью кэширования на основе файловой системы. OkHttp (которая также входит в состав популярной библиотеки *Retrofit* в качестве транзитивной зависимости) — это популярная клиентская библиотека, широко используемая в Android для сетевых взаимодействий. Добавить кэширование относительно просто:

```
val cacheSize = 10 * 1024 * 1024
val cache = Cache(rootDir, cacheSize)

val client = OkHttpClient.Builder()
                .cache(cache)
                .build()
```

Теперь можно легко создавать конфигурации с собственными перехватчиками, которые лучше подходят для варианта использования приложения. Например, перехватчики могут сделать так, что кэш будет обновляться с заданным интервалом. Кэширование — отличный инструмент для устройства, работающего с ограничен-

---

[4] Экземпляр Bitmap, который вы предоставляете, должен быть изменяемым.

[5] LRU (least recently used) — вытеснение давно неиспользуемых. Поскольку объекты нельзя кэшировать бесконечно, кэширование всегда связано со стратегией вытеснения, чтобы поддерживать желаемый или приемлемый размер кэша. В этом алгоритме "самые старые" объекты удаляются первыми.

ными ресурсами в своем окружении. По этой причине разработчики приложений для Android должны использовать кэш для отслеживания расчетных вычислений.

Dropbox Store[6] — прекрасная библиотека с открытым исходным кодом, которая поддерживает кэширование как *в памяти*, так и *на основе файловой системы*.

Избавляемся от ненужной работы

Чтобы приложение потребляло ресурсы экономно, не следует оставлять код, который делает ненужную работу. Даже опытные разработчики обычно совершают подобные ошибки, что приводит к дополнительной работе и неэффективному выделению памяти. Например, собственные представления в Android требуют особого внимания. Рассмотрим собственное представление с окружностью. Для его реализации можно создать подкласс любого представления и переопределить метод `onDraw`. Вот одна из возможных реализаций класса `CircleView`:

```
// Внимание: это пример того, как НЕЛЬЗЯ делать!
class CircleView @JvmOverloads constructor(
    context: Context,
) : View(context) {

    override fun onDraw(canvas: Canvas) {
        super.onDraw(canvas)
        canvas.save()
        // Никогда не инициализируйте здесь выделение памяти под объекты!
        val paint: Paint = Paint().apply {
            color = Color.parseColor("#55448AFF")
            isAntiAlias = true
        }
        canvas.drawCircle(100f, 100f, 50f, paint)
        canvas.restore()
    }
}
```

Метод `onDraw` вызывается каждый раз, когда представление необходимо перерисовать. Такое может происходить довольно часто, особенно если это представление анимированное. Поэтому никогда не следует создавать экземпляры новых объектов в этом методе. Такие ошибки приводят к ненужному выделению памяти для боль-

[6] См. **https://oreil.ly/urfwv**.

шого количества объектов, что создает большую нагрузку на сборщика мусора. В предыдущем примере новый экземпляр объекта `Paint` создается каждый раз, когда слой отображения рисует `CircleView`. Ни при каких обстоятельствах не нужно этого делать.

Вместо этого лучше один раз создать экземпляр объекта `Paint` в качестве атрибута класса:

```
class CircleView @JvmOverloads constructor(
    context: Context,
) : View(context) {

    private var paint: Paint = Paint().apply {
        color = Color.parseColor("#55448AFF")
        isAntiAlias = true
    }
        set(value) {
            field = value
            invalidate()
        }

    override fun onDraw(canvas: Canvas) {
       super.onDraw(canvas)
       canvas.save()
       canvas.drawCircle(100f, 100f, 50f, paint)
       canvas.restore()
    }
}
```

Теперь память под объект `Paint` выделяется только один раз. Для этого класса иногда в качестве значения `paint` задавались разные цвета. Однако, если присваивание не является динамическим, можно сделать еще один шаг, используя отложенное вычисление этого значения.

Нам нужно, чтобы по возможности внедрение было сбалансированным, а зависимости — легковесными. Для репозиториев, служб и других одноэлементных зависимостей (зависимостей, представляющих собой одиночные объекты в памяти, таких как `object`) имеет смысл использовать отложенное делегирование, чтобы у нас был одиночный экземпляр, а не копии одного и того же объекта, находящегося в куче.

Рассмотрим код, с которым мы работали в *разд. "Обнаружение утечек памяти с помощью LeakCanary" главы 11*:

```
class MapImportViewModel @ViewModelInject constructor(
    private val settings: Settings,
    private val app: Application
```

```
): ViewModel() {
    /* удалено для краткости */

    fun unarchiveAsync(item: ItemData) {
        viewModelScope.launch {
            val inputStream = app.contentResolve.
                openInputStream(item.uri) ?: return@launch
            val rootFolder = settings.getAppDir() ?: return@launch
            val outputFolder = File(rootFolder, "imported")
            /* удалено для краткости */
        }
    }
}
```

В этом классе зависимость `settings` внедряется с помощью Hilt — это видно по аннотации `@ViewModelInject`. Когда мы писали этот пример, то использовали Hilt 2.30.1-alpha, и в `ViewModel` можно было внедрять только зависимости, доступные в области видимости активности. Другими словами, только что созданная `MapImportViewModel` всегда внедряется в один и тот же экземпляр `Settings`, если активность не создается повторно. Итак, суть такова: фреймворк внедрения зависимостей, например Hilt, может помочь нам определить жизненный цикл зависимостей. В TrekMe область видимости `Settings` ограничена приложением.

Таким образом, технически `Settings` — это синглтон.

Android Jetpack

Hilt — это DI-фреймворк (dependency injection — внедрение зависимостей), предоставляющий стандартный способ использования внедрения зависимостей в приложении. Он обладает преимуществом автоматического управления жизненными циклами, и у него есть расширения, доступные для использования с такими компонентами Jetpack, как ViewModel и WorkManager.

Попытка избежать ненужной работы распространяется на все области разработки приложений для Android. При рисовании объектов для отображения их в пользовательском интерфейсе имеет смысл перерабатывать уже нарисованные пикселы. Кроме того, поскольку мы знаем, что выполнение сетевых вызовов в Android разряжает аккумулятор, полезно проверять, сколько вызовов было осуществлено и как часто они выполнялись. Возможно, у вас в приложении есть корзина покупок. Возможно, для бизнеса имеет смысл выполнять обновления на удаленном сервере, чтобы пользователь мог получать кросс-платформенный доступ к корзине. С другой стороны, вероятно, также стоит изучить возможность обновления корзины пользователя в локальном хранилище (за исключением периодического обновления сети). Конечно, такого рода бизнес-решения выходят за рамки этой книги, но соображения технического характера всегда могут помочь создать более продуманную функциональность.

Использование статических функций

Когда метод или свойство не привязаны ни к одному из экземпляров класса (например, не изменяют состояние объекта), иногда имеет смысл использовать *статические функции* или *свойства*. Мы продемонстрируем различные сценарии, в которых использование статических функций более целесообразно, нежели наследование.

Kotlin упрощает использование статических функций. Объект-компаньон в объявлении класса содержит статические константы, свойства и функции, на которые можно ссылаться в любом месте проекта. Например, служба Android может предоставлять доступ к статическому свойству `isStarted`, изменить которое в состоянии лишь сама служба, как показано в примере 12.1.

Пример 12.1. GpxRecordingService.isStarted

```
class GpxRecordingService {

    /* Удалено для краткости */

    companion object {
        var isStarted: Boolean = false
        private set(value) {
            EventBus.getDefault().post(GpxRecordServiceStatus(value))
            field = value
        }
    }
}
```

В этом примере `GpxRecordingService` может "под капотом" изменить значение свойства `isStarted`. При этом событие отправляется через шину событий, уведомляя все зарегистрированные компоненты. Более того, статус `GpxRecordingService` доступен из любого места приложения в виде свойства `GpxRecordingService.isStarted` с доступом только для чтения. Но не забывайте: следует избегать случайного сохранения активности, фрагмента, представления или контекста в статическом члене: это может привести к серьезной утечке памяти!

Минификация и обфускация с *R8* и *ProGuard*

Обычно для промышленной эксплуатации принято *минифицировать* или сокращать код в сборках выпуска, чтобы неиспользуемый код и ресурсы можно было удалить. Минификация кода позволяет более безопасно отправлять небольшие APK-файлы в Google PlayStore. Так вы сокращаете код, удаляя неиспользуемые ме-

тоды. Минификация кода также обеспечивает возможность *обфускации* в качестве дополнительной меры безопасности. В ходе обфускации имена классов, полей или методов искажаются, а атрибуты отладки удаляются, чтобы препятствовать обратной разработке.

Для пользователей Android R8 теперь является инструментом минификации по умолчанию. Он предоставляется плагином Android Gradle версии 5.4.1 и выше. ProGuard, более строгий и мощный предшественник R8, уделял больше внимания оптимизации сильной рефлексии, подобной той, что встречается в Gson. По сравнению с ним более новый инструмент минификации R8 не поддерживает эту возможность. Тем не менее он успешно обеспечивает меньшее сжатие и оптимизацию для Kotlin.

Настройки можно выполнить с помощью файла конфигурации proguard (в конце раздела будет показано, как это сделать). R8 считывает правила, предусмотренные для этого файла, и выполняет минификацию и обфускацию. После этого можно назначить файл proguard определенному варианту и виду сборки в *build.gradle*:

```
buildTypes {
    release {
        minifyEnabled true
        shrinkResources true
    proguardFile getDefaultProguardFile('proguard-android-
optimize.txt'), 'proguard-rules.pro'
    }
}
```

Обычно для загрузки файла в PlayStore код APK минифицируют. Тем не менее важно проявлять бдительность и предотвращать непреднамеренную минификацию или обфускацию кода, который может потребоваться сторонней библиотеке во время выполнения. Kotlin использует метаданные в классах Java для конструкций Kotlin. Однако когда R8 минифицирует классы Kotlin, он не может сохранить состояние с метаданными Kotlin. В лучшем случае минификация или обфускация таких классов способны привести к нестабильному поведению, а в худшем — к необъяснимым сбоям.

Чтобы продемонстрировать сценарий, в котором ProGuard случайно запутывает слишком большое количество кода приложения, мы понаблюдаем за нестабильным поведением в популярной библиотеке с открытым исходным кодом Retrofit. Возможно, ваше приложение отлично работает в режиме отладки, но в режиме выпуска сетевой вызов необъяснимым образом возвращает исключение `NullPointerException`. К сожалению, модели Gson в Kotlin остаются пустыми даже при аннотировании свойств или полей с помощью аннотации Retrofit, `@SerializedName`, благодаря рефлексии. В результате нужно добавить в файл proguard правило, чтобы предотвратить обфускацию класса модели Kotlin.

Часто у вас может появляться необходимость включить классы модели, добавляя их непосредственно в файл proguard. Вот пример добавления классов предметной

области модели в файл proguard, чтобы сборки выпусков случайно не "запутали" вышеупомянутые классы:

```
# Retrofit 2.X
-dontwarn retrofit2.**
-keep class retrofit2.** { *; }
# Kotlin source code whitelisted here
-keep class com.some.kotlin.network.model.** { *; }
-keepattributes Signature
-keepattributes Exceptions
-keepclasseswithmembers class * {
    @retrofit2.http.* <methods>;
}
```

Хороший совет: всегда проводите тестирование сборки выпуска!

Резюме

В этой главе были предложены следующие важные советы по оптимизации производительности.

- В представлениях Android глубоко вложенные иерархии занимают больше времени для рисования и обхода, чем более плоские иерархии. Рассмотрите возможность использования класса `ConstraintLayout`, где вложенные представления можно уплощать.
- В представлениях Android лучше перемещать операции рисования программными средствами и анимацию в графические ресурсы, чтобы передать часть работы и разгрузить RenderThread во время выполнения.
- Использование форматов JSON и XML для сетевых данных не подходит для сжатия. Используйте буферы протоколов.
- По возможности избегайте лишней работы: убедитесь, что вы не отключаете ненужные сетевые вызовы для постоянных обновлений и попытки переработать нарисованные объекты.
- Оптимизации производительности и памяти можно достичь благодаря объективному анализу кода, который вы пишете. Вы непреднамеренно создаете объекты внутри цикла, которые можно было бы создать за его пределами? Какие затратные операции можно было бы сократить, чтобы сделать их менее интенсивными?
- Можно использовать файл ProGuard, чтобы сделать приложение как можно меньше, и добавить собственные правила для сжатия, обфускации и оптимизации приложения.

Посмотрим правде в глаза: Android может стать вызовом, к которому вы должны быть готовы. Это нормально, если вы воспринимаете информацию пошагово, по мере того как она становится актуальной для вас. Такая стратегия гарантирует возможности обучения, которые останутся с вами надолго. Неважно, в какой точке пути вы находитесь: один из лучших ресурсов для изучения Kotlin и Android (помимо этой книги) — это сообщество разработчиков программного обеспечения с открытым исходным кодом. Сообщества Android и Kotlin предоставляют самую последнюю и актуальную информацию. Чтобы быть в курсе событий, можно обратиться к дополнительным ресурсам: Twitter, Slack[7] и KEEP[8]. Возможно, вы также обнаружите, что возвращаетесь к этой книге, чтобы еще раз рассмотреть популярные и извечные проблемы, связанные с Android, которые появляются время от времени. Надеемся, она вам понравилась.

[7] См. **https://oreil.ly/m853Y**.

[8] См. **https://oreil.ly/KZPlx**.

Предметный указатель

Г

Д

И

К

Л

М

О

П

С

Т

У

Ф

Ч, Э, Я

O'REILLY®

Программирование на Kotlin для Android

Разработка мобильных приложений для Android может показаться сложной задачей, особенно если для этого требуется изучить новый язык программирования. Речь идет о Kotlin, ставшем официальным языком разработки для этой операционной системы. Книга поможет быстро освоить этот язык, обладающий целым рядом технологических преимуществ, а также перейти с Java на Kotlin.

Авторы приводят реализацию наиболее распространенных задач в нативной разработке для Android и показывают, как Kotlin помогает решить проблему параллелизма. Делая акцент на структурированном параллелизме, новой парадигме асинхронного программирования, книга помогает освоить одну из самых мощных конструкций Kotlin — сопрограммы.

- **Познакомьтесь с основами Kotlin и его фреймворком коллекций**
- **Изучите операционную систему Android, контейнер приложения и его компоненты**
- **Познакомьтесь с потокобезопасностью и узнайте, как работать с параллелизмом**
- **Пишите последовательный асинхронный код с низкими затратами**
- **Изучите структурированный параллелизм с помощью сопрограмм и узнайте, как сопрограммы взаимодействуют между собой с помощью каналов**
- **Узнайте, как использовать потоки для асинхронной обработки данных**
- **Изучите вопросы производительности с помощью инструментов профилирования**
- **Оптимизируйте производительность, чтобы сократить потребление ресурсов**

Пьер-Оливье Лоранс — ведущий инженер-программист компании Safran Aircraft Engines.

Аманда Хинчман-Домингес — эксперт по языку Kotlin в программе Google Developer Expert. Работает разработчиком для Android в компании Groupon и является активным участником глобального сообщества Kotlin.

Дж. Блейк Мик — старший инженер-программист компании Couchbase и автор нескольких книг, среди которых *«Программирование под Android»*.

Майк Данн работал ведущим инженером по мобильным технологиям в издательстве O'Reilly Media. Один из авторов книги *«Нативная разработка мобильных приложений. Перекрестный справочник для iOS и Android»*.