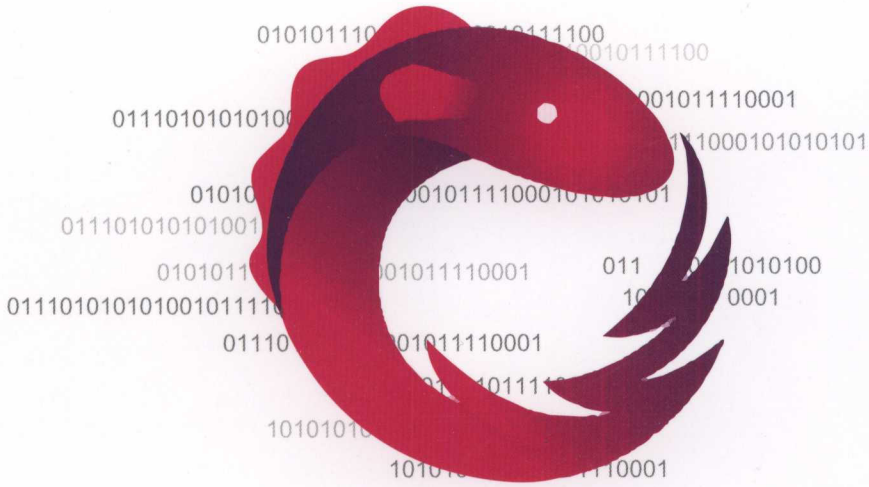
Broadview
www.broadview.com.cn

RxJava 2.x 实战

沈哲 编著



本书通过完整的体系，深入浅出地讲解了RxJava 2.x的方方面面

并通过案例详细讲解了如何解决一些生产环境的实战问题

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

作者介绍

沈哲，计算机软件与理论硕士，负责魔窗SDK的架构和研发，主要包括实现deep link(deferred deep link)、移动端原生广告、信息流广告、SDK安全、对第三方框架的裁剪，等等。保障了魔窗SDK多年的线上稳定运行。

超过8年的移动开发经验，负责过京东到家上海的移动团队，全程参与今夜酒店特价App的开发，Decarta Map SDK的开发，参与过格瓦拉App的开发。

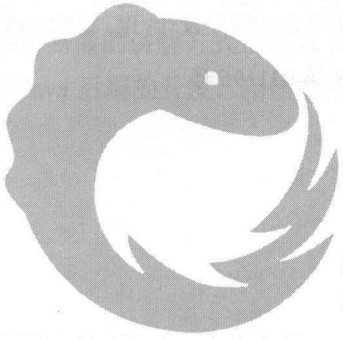
多年的服务端开发经验，负责过京东旅行邮轮业务部门的后端团队。

熟悉函数响应式编程，了解计算机视觉。

目前，关注分布式账本、区块链经济。

RxJava 2.x 实战

沈哲 编著



電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书首先讲解了函数式响应式编程的概念，包括 Observables、Subject、Processor 等，以及 RxJava 的优点和用途。然后讲解了 RxJava 中必不可少的操作符，包括创建操作符、变换操作符、过滤操作符、条件操作符、布尔操作符、合并操作符和连接操作符。接着详细介绍了 RxJava 的线程操作及线程模型，用大量的实例，讲解了 RxJava 在 Android 上和在 Java 后端的使用。最后，介绍了 Java 8 的函数式编程的特性，以及对未来编程方式的展望。

本书适合 Android 开发工程师、Java 后端开发工程师，以及对函数响应式编程、感兴趣的 IT 从业人员。

图书在版编目（CIP）数据

RxJava 2.x 实战 / 沈哲编著. —北京：电子工业出版社，2018.4
ISBN 978-7-121-33722-2

Ⅰ. ①R… Ⅱ. ①沈… Ⅲ. ①JAVA 语言－程序设计 Ⅳ. ①TP312.8

中国版本图书馆 CIP 数据核字(2018)第 031196 号

责任编辑：安　娜
印　　刷：三河市华成印务有限公司
装　　订：三河市华成印务有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　印张：26.75　字数：497 千字
版　　次：2018 年 4 月第 1 版
印　　次：2018 年 4 月第 1 次印刷
定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

前　言

编写这本书的目的

笔者从 2015 年开始关注 RxJava 以及响应式编程，当时网上的资料很少。三年后的今天，我们可以看到越来越多的 App 都在使用 Rx 相关的技术。在 Java 后端，Spring 5 也开始支持响应式编程。在开源的技术社区里，Rx、响应式编程、函数式编程都是热门话题。我们公司开发的 App，笔者也会强制大家使用 RxJava 或者 RxSwfit。本书通过完整的体系介绍了 RxJava 的方方面面。

对于不了解响应式编程的开发者，RxJava 的入门可能会有一些难度。笔者结合自身的学习以及使用情况，尽可能使用通俗易懂的讲解方式带领大家学习 RxJava。同时，本书还附有丰富的例子，从 Android 开发到 Java 后端的开发，相信能够让大家感受到 RxJava 的魅力。

读者对象

1）Android 开发工程师。

2）Java 后端开发工程师。

3）对函数式编程、响应式编程感兴趣的 IT 从业人员。

如何阅读本书

本书共分为 18 章。

第 1 章讲述了 RxJava 的来龙去脉，以及什么是响应式编程、什么是函数式编程。

第 2 章至第 7 章讲述了 RxJava 的基础概念，以及 RxJava 常用的操作符。

第 8 章至第 10 章为 RxJava 的高级部分。

第 11 章至第 12 章为 RxAndroid 的使用篇，介绍了常用的 RxBinding、Retrofit 等框架。

第 13 章至第 15 章为 RxJava 的实战篇，介绍了如何实现一个基于 RxJava 的 Event Bus 框架，以及 Spring Boot 如何与 RxJava 相结合使用。

第 16 章至第 18 章介绍了 Java 8 函数式编程的特性和 Kotlin，并展望未来。

勘误和支持

本书相关例子的源码都在 GitHub 上，地址：https://github.com/fengzhizi715/RxJavaInAction。

由于笔者水平有限，编写本书时难免会出现错误或者纰漏，恳请读者批评指正。读者可以关注笔者的公众号与笔者进行互动。或者通过邮箱：fengzhizi715@126.com，有关本书的任何问题都可以反馈给笔者，笔者期待与您的技术交流。

致谢

首先要感谢我的家人，最主要是感谢我的妻子。在写书期间，恰逢儿子的出生，她承担了绝大部分照顾儿子的责任。

感谢公司的支持与同事的帮助，特别是刘志强帮我整理了很多 RxJava 相关的资料，以及对本书部分章节进行了试读，并提出意见。

感谢 www.bsfans.com 罗波同学提供 UI 支持。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/33722

目　录

第1章 RxJava简介

1.1 你需要了解的函数响应式编程

如果你曾经使用过 Java，那么你一定听说过面向对象的编程思想，也可能听说过 AOP（Aspect Orient Programming，面向切面编程）的编程思想。本书要讲的是全新的编程思想，跟它们没有丝毫的联系。

1. 响应式编程（Reactive Programming，简称 RP）

在计算机中，响应式编程是一种面向数据流和变化传播的编程范式。这意味着可以在编程语言中很方便地表达静态或动态的数据流，而相关的计算模型会自动将变化的值通过数据流进行传播。

传统的编程方式是顺序执行的，需要等待直至完成上一个任务之后才会执行下一个任务。无论是提升机器的性能还是代码的性能，本质上都需要依赖上一个任务的完成。如果需要响应迅速，就得把同步执行的方式换成异步执行，方法执行变成消息发送。这就是异步编程的方式，它是响应式编程的重要特性之一。

响应式编程有以下几个特点。

◎ 异步编程：提供了合适的异步编程模型，能够挖掘多核 CPU 的能力、提高效率、降低延迟和阻塞等。

- ◎ 数据流：基于数据流模型，响应式编程提供一套统一的 Stream 风格的数据处理接口。与 Java 8 中的 Stream 相比，响应式编程除了支持静态数据流，还支持动态数据流，并且允许复用和同时接入多个订阅者。
- ◎ 变化传播：简单来说就是以一个数据流为输入，经过一连串操作转化为另一个数据流，然后分发给各个订阅者的过程。这就有点像函数式编程中的组合函数，将多个函数串联起来，把一组输入数据转化为格式迥异的输出数据。

响应式编程一方面在用户界面编程领域及基于实时系统的动画方面都有广泛的应用；另一方面，在处理嵌套回调的异步事件、复杂的列表过滤和变换的时候也都有良好的表现。

现在的 App 无论是原生 H5、HTML5 还是 Hybird，都会和与数据事件相关的 UI 事件进行大量的交互，使用响应式编程会显得更加得心应手。

这些年来前端比较流行的响应式设计，实际上是指网页能够自动调整布局和样式以适配不同尺寸的屏幕，与我们谈论的响应式编程是两个完全不同的概念。

2. 函数式编程（Functional Programming，简称 FP）

随着硬件能力的不断提升，单核 CPU 的计算能力几乎达到极限，CPU 已经进入了多核时代，程序员转而通过并发编程、分布式系统来应对越来越复杂的计算任务。

然而并发编程并不是银弹，作为一种基于共享内存的并发编程，多线程编程有常见的死锁、线程饥饿、竞争条件等问题，而且多线程的 Bug 也难以重现和定位。于是，函数式编程开始兴起。

在函数式编程中，由于数据是不可变的（immutable），因此没有并发编程的问题，是线程安全的。它将计算机运算看作数学中函数的计算，主要特点是将计算过程分解成多个可复用的函数，并且避免了状态及变量的概念。函数式编程虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

函数式编程具有以下特点。

- ◎ 函数是“第一等公民”：所谓“第一等公民”（First Class），指的是函数与其他数据类型一样，处于平等地位，可以赋值给其他变量，也可以作为参数，传入另一个函数，或者作为别的函数的返回值。

- ◎ 闭包和高阶函数：闭包是起函数的作用并可以像对象一样操作的对象。与此类似，FP 语言支持高阶函数。高阶函数可以用另一个函数（间接地，用一个表达式）作为输入参数，在某些情况下，它甚至返回一个函数作为输出参数。这两种结构结合在一起，即可以用优雅的方式进行模块化编程，这是使用 FP 的最大好处。
- ◎ 递归：把递归作为控制流程的机制。例如，在 Haskell 的世界中，没有变量赋值和流程跳转，如果要实现一些简单的功能，比如求一个数组中的最大值，则需要借助递归来实现。
- ◎ 惰性求值（Lazy Evaluation）：它表示为“延迟求值”和“最小化求值”。惰性求值使得代码具备了巨大的优化潜能。支持惰性求值的编译器会像数学家看待代数表达式那样看待函数式编程的程序，即抵消相同项从而避免执行无谓的代码，安排代码执行顺序从而实现更高的执行效率甚至是减少错误。惰性求值另一个重要的好处是它可以构造一个无限的数据类型，无须担心由无穷计算所导致的内存溢出错误。
- ◎ 没有“副作用”（Side Effect）：指的是函数内部与外部互动（最典型的情况，就是修改全局变量的值），产生运算以外的其他结果。函数式编程强调没有“副作用”，意味着函数要保持独立，所有功能就是返回一个新的值，没有其他行为，尤其是不得修改外部变量的值。

既然函数式编程已经能够解决并发的问题，那为何还需要响应式编程呢？

3. 函数响应式编程（Functional Reactive Programming，简称 FRP）

函数响应式编程结合了函数式和响应式的优点，把函数范式里的一套思路和响应式编程合起来就是函数响应式编程。

我们知道，传统的面向对象编程是通过抽象出的对象关系来解决问题，函数式编程是通过函数（function）的组合来解决问题，响应式编程是通过函数式编程的方式来解决回调地狱（Callback Hell）的问题。

用传统的面向对象来处理异步事件不是很直观，处理并发也十分麻烦，所以才产生了函数响应式编程。

下面一起进入 RxJava 的世界，体验一下函数响应式编程带来的乐趣。

1.2 RxJava 简介

1. RxJava 产生的由来

RxJava 是 Reactive Extensions 的 Java 实现，用于通过使用 Observable/Flowable 序列来构建异步和基于事件的程序的库。

RxJava 扩展观察者模式以支持数据/事件序列，并添加允许你以声明方式组合序列的操作符，同时提取对低优先级的线程、同步、线程安全性和并发数据结构等问题的隐藏。

2. 什么是 Rx

ReactiveX 是 Reactive Extensions 的缩写，一般简写为 Rx，最初是 LINQ 的一个扩展，由微软架构师 Erik Meijer 领导的团队所开发，于 2012 年 11 月开源。Rx 是一个编程模型，目标是提供一致的编程接口，帮助开发者更方便地处理异步数据流。Rx 库支持 .NET、JavaScript 和 C++。由于 Rx 近几年越来越流行，因此现在已经支持几乎全部的流行编程语言。Rx 的大部分语言库由 ReactiveX 这个组织负责维护，比较流行的有 RxJava、RxJS 和 Rx.NET，社区网站是 http://reactivex.io/。

3. ReactiveX 的历史

微软给的定义是，Rx 是一个函数库，让开发者可以利用可观察序列和 LINQ 风格查询操作符来编写异步和基于事件的程序，通过使用 Rx，开发者可以用 Observables 表示异步数据流，用 LINQ 操作符查询异步数据流，用 Schedulers 参数化异步数据流的并发处理。可以这样定义 Rx：Rx = Observables + LINQ + Schedulers。

注：在 RxJava 2.x 中，Observables 包含了 Observables、Flowable、Single、Maybe、Completable 等。

ReactiveX.io 给的定义是，Rx 是一个使用可观察数据流进行异步编程的编程接口，ReactiveX 结合了观察者模式、迭代器模式和函数式编程的精华。

这个定义听上去很拗口，不过没有关系，在第 2 章和第 4 章会分别讲述 Observable 和 Scheduler。

4. Rx 模式

（1）使用观察者模式。

◎ 创建：Rx 可以方便地创建事件流和数据流。
◎ 组合：Rx 使用查询式的操作符组合和变换数据流。
◎ 监听：Rx 可以订阅任何可观察的数据流并执行操作。

（2）简化代码。

◎ 函数式风格：对可观察的数据流使用无副作用的输入/输出函数，避免了程序里错综复杂的状态。
◎ 简化代码：Rx 的操作符通常可以将复杂的难题简化为很少的几行代码。
◎ 异步错误处理：传统的 try/catch 没办法处理异步计算，Rx 提供了合适的错误处理机制。
◎ 轻松使用并发：Rx 的 Observables（包括 Observable、Flowable、Single、Completable 和 Maybe）和 Schedulers 可以让开发者摆脱底层的线程同步和各种并发问题。

总而言之，RxJava 是 Reactive Extensions 在 JVM 平台上的一个实现，通过使用观察者序列来构建异步、基于事件的程序。RxJava 可以说是观察者设计模式的一个扩展，支持不同的数据/事件流和额外的操作类，允许通过声明式的方式构建不同的执行序列，通过抽象的方式屏蔽底层的多线程实现、同步、线程安全、并发数据结构、非阻塞 I/O 等逻辑。 RxJava 支持 Java 5 之后的版本，还支持跑在 JVM 上的各种其他语言，例如 Groovy、Clojure、JRuby、Kotlin 和 Scala 等。笔者建议尽量使用 Java 8 及以上的版本，因为能够使用 Lambda 表达式等新特性。

RxJava 使用 Observables 来访问多对象的异步序列。 Observables 是可组合的，Java Futures 模型在用于单一层面的异步执行方面比较方便，但是在需要异步嵌套执行时使用起来却很复杂，至少在 Java 8 出现之前是这样的。用 Futures 在构建有条件的异步执行流时会非常困难，因为在运行时每次请求的延迟都是不可控的。虽然理论上能够做到，但是过程会很复杂，有时候会在 Future.get()上一直阻塞，这样的结果显然无法体现异步执行的优势。在 Java 8 之后新增了 CompletableFuture，弥补了原先 Futures 模型的问题。

当然，RxJava 不并限于是在 Java 8 之后，还是之前使用。 Observables 本身是专门用于构建异步数据流和执行片段的，它很灵活，RxJava 的 Observables 除了支持 Futures 模型的所有功能外，还支持无限的数据流；Observables 本身是一个统一抽象，用于支持不用情况下的异步数

据序列的执行过程。

1.3 为何选择 RxJava

Rx 扩展了观察者模式用于支持数据和事件序列，添加了一些操作符，让你可以使用声明式的方式来组合这些序列，而无须关注底层的实现，如线程、同步、线程安全、并发数据结构和非阻塞 I/O。

Rx 的 Observable 模型让开发者可以像使用集合数据一样操作异步事件流，对异步事件流使用各种简单、可组合的操作。

◎ 可组合

对于单层的异步操作来说，Java 中 Future 对象的处理方式非常简单有效，但是一旦涉及嵌套，它们就开始变得异常烦琐和复杂。在 Java 8 之前，使用 Future 很难很好地组合带条件的异步执行流程。从另一方面来说，RxJava 的被观察者们（Observable/Flowable/Single/Completable/Maybe）一开始就是为组合异步数据流准备的。

◎ 更灵活

RxJava 的 Observable 不仅支持处理单独的标量值（就像 Future 可以做的），还支持数据序列，甚至是无穷的数据流。Observable 是一个抽象概念，适用于任何场景。Observable 拥有它的近亲 Iterable 的全部优雅与灵活。

Observable 是异步的双向 push，Iterable 是同步的单向 pull，可以通过表 1-1 进行对比。

表 1-1

事　件	Iterable（pull）	Observable（push）
获取数据	T next()	Consumer<? super T> onNext
异常处理	throws Exception	Consumer<? super Throwable> onError
任务完成	!hasNext()	Action onComplete

◎ 无偏见

Rx 对于并发性或异步性没有任何特殊的偏好，Observable 可以用任何方式（如线程池、事件循环、非阻塞 I/O 或 Actor 模式）来满足我们的需求。无论我们选择怎样实现它，无论底层实

现是阻塞的还是非阻塞的，客户端代码都将与 Observable 的全部交互当成是异步的。

1. Observable 是如何实现的

```
public Observable<T> getData();
```

- ◎ 它能与调用者在同一线程同步执行吗？
- ◎ 它能异步地在单独的线程执行吗？
- ◎ 它会将工作分发到多个线程，返回数据的顺序是任意的吗？
- ◎ 它使用 Actor 模式而不是线程池吗？
- ◎ 它使用 NIO 和事件循环执行异步网络访问吗？
- ◎ 它使用事件循环将工作线程从回调线程分离出来吗？

从 Observer 的视角来看，这些都无所谓，重要的是：使用 Rx，可以改变我们的观念，可以在完全不影响 Observable 程序库使用者的情况下，彻底地改变 Observable 的底层实现。

2. 使用回调存在很多问题

回调在不阻塞任何事情的情况下，解决了 Future.get()过早阻塞的问题。响应结果一旦就绪，Callback 就会被调用，它们天生就是高效率的。不过，就像使用 Future 一样，对于单层的异步执行来说，回调很容易使用，对于嵌套的异步组合而言，它们显得非常笨拙。

3. Rx 是一个多语言的实现

Rx 在大量的编程语言中都有实现，并尊重实现语言的风格，而且更多的实现正在飞速增加。

4. 响应式编程

Rx 提供了一系列的操作符，我们可以使用它们来过滤（filter）、选择（select）、变换（transform）、结合（combine）和组合（compose）多个 Observable，这些操作符让执行和复合变得非常高效。

我们可以把 Observable 当作 Iterable 推送方式的等价物，使用 Iterable，消费者从生产者那拉取数据，线程阻塞直至数据准备好。使用 Observable，在数据准备好时，生产者将数据推送给消费者。数据可以同步或异步到达，这种方式更加灵活。

下面的例子展示了相似的高阶函数在 Iterable 和 Observable 上的应用。

```
// Iterable
```

```
getDataFromLocalMemory()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed" })
  .forEach({ println "next => " + it })

// Observable
getDataFromNetwork()
  .skip(10)
  .take(5)
  .map({ s -> return s + " transformed" })
  .subscribe({ println "onNext => " + it })
```

Observable 类型给 GOF 的观察者模式添加了两种缺少的语义，这样就和 Iterable 类型中可用的操作一致了。

生产者可以发信号给消费者，通知它没有更多数据可用了（对于 Iterable，一个 for 循环正常完成表示没有数据了；对于 Observable，就是调用观察者的 onComplete 方法） 生产者可以发信号给消费者，通知它遇到了一个错误（对于 Iterable，在迭代过程中发生错误会抛出异常；对于 Observable，就是调用观察者（Observer）的 onError 方法）。有了这两种功能，Rx 就能使 Observable 与 Iterable 保持一致了，唯一的不同是数据流的方向。任何对 Iterable 的操作，都可以对 Observable 适用。

再举一个例子，某 App 从服务端获取酒店列表，并将价格大于等于 500 元的房间全部列出来展示在界面上。

```
new Thread() {
        @Override
        public void run() {
            super.run();
            //从服务端获取酒店列表
            List<Hotel> hotels = getHotelsFromServer();
            for (Hotel hotel : hotels) {
                List<Room> rooms = hotel.rooms;
                for (Room room : rooms) {
                    if (room.price >= 500) {
                        runOnUiThread(new Runnable() {
                            @Override
                            public void run() {
```

```
                        //将房间的信息添加到 UI 上
                        displayUI(room);
                    }
                });
            }
        }
    }
}.start();
```

如果使用 RxJava 的话，大致可以这样写。

```
Observable.fromIterable(getHotelsFromServer())
        .flatMap(new Function<Hotel, Observable<Room>>() {
            @Override
            public Observable<Room> apply(Hotel hotel) {
                return Observable.fromIterable(hotel.rooms);
            }
        })
        .filter(new Predicate<Room>() {
            @Override
            public boolean test(@NonNull Room room) throws Exception {
                return room.price >= 500;
            }
        })
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Room>() {
            @Override
            public void accept(Room room) {
                //将房间的信息添加到屏幕上
                displayUI(room);
            }
        });
```

当然，使用 Lanbda 表达式之后还可以更加简化一点。

```
Observable.fromIterable(getHotelsFromServer())
        .flatMap(hotel -> Observable.fromIterable(hotel.rooms))
        .filter(room -> room.price>=500)
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
```

```
                .subscribe(this::displayUI);
```

其实，下面这两句还能用 compose 操作符和 Transformer 一起封装。

```
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
```

在后面的章节中会详细描述，在这里如果没有理解也没有关系。

1.4 RxJava 能做什么

曾经有人问我，在 Android 开发中哪些地方可以使用 RxJava？我当时给的回答是，任何地方，包括 App 所依赖的底层框架。

例如，可以使用 RxBinding，它是对 Android View 事件的扩展，可以对 View 事件使用 RxJava 的各种操作。

也可以使用 RxJava 替换原先的 EventBus，在 Square Otto 的 GitHub 主页上有这样两句话。

“This project is deprecated in favor of RxJava and RxAndroid. These projects permit the same event-driven programming model as Otto, but they’re more capable and offer better control of threading.”

对于 Android 的 AsyncTask，也可以完全使用 RxJava 来替代。

这些内容，本书后续的章节都会有详细的介绍。

当然，不只是 Android App 的开发，在服务端领域的开发中也可以使用 RxJava。本书后面几章会有一些例子涉及如何在服务端开发中使用 RxJava。

总之，RxJava 能做的事情很多，如果你的项目中还没有引入 RxJava，那么现在是引入 RxJava 2.x 的最好时机，它会给你带来新的编程体验，让你感受响应式编程的乐趣。如果你的项目中已经使用了 RxJava 1.x，那么强烈建议你升级到最新的版本。RxJava 2 还是使用原来的思想，与 RxJava 1 区别不大，从 RxJava 1 迁移到 RxJava 2 成本并不高，也花费不了多少工夫。

1.5　RxJava 2 的 Hello World

学习一门新的语言或者一个新的框架，一般第一次接触时都会写一个 Hello World。

下面来看看 RxJava 版本的 Hello World 吧。

```
Observable.create(new ObservableOnSubscribe<String>() {

        @Override
        public void subscribe(@NonNull ObservableEmitter<String> e) throws
Exception {
            e.onNext("Hello World");
        }
    }).subscribe(new Consumer<String>() {
        @Override
        public void accept(@NonNull String s) throws Exception {
            System.out.println(s);
        }
    });
```

为什么这个 Hello World 版本这么长？能不能写得简单一点呢？当然可以啦！

```
Observable.just("Hello World").subscribe(new Consumer<String>() {
        @Override
        public void accept(@NonNull String s) throws Exception {
            System.out.println(s);
        }
    });
```

是不是比刚才简单了一些，还想再简单一点吗？那好，我们借助 Lambda 表达式再来看看。

```
Observable.just("Hello World").subscribe(System.out::println);
```

这次一句话就搞定了，符合 Hello World 的本意。System.out::println 是方法引用，简化版的 Lambda 表达式，如果不熟悉 Lambda 表达式也不用纠结，它是 Java 8 的特性，在本书的第 16 章中会有详细介绍，对此感兴趣的读者也可以提前去了解一下。

1.6 小结

本章简单介绍了 RxJava 的历史，RxJava 是结合了多种设计模式并优化而产生的结晶。

RxJava 这么好，到底有哪些实际应用呢？目前在移动端可以看到大量使用 RxJava 来编写的代码，也因此产生了大量的优秀框架，比如 RxBinding、Retrofit、RxLifecycle 等，服务端的开发也可以大量使用 RxJava。CouchBase 这个 NoSQL 数据库的 Java SDK 就是采用 RxJava 来编写的。除此之外，Netflix 的 Hystrix 的底层也是使用 RxJava 来编写的，在微服务架构中 Hystrix 用于帮助隔离每个服务，即使单个服务的响应失败，也不会影响整个请求的响应。

RxJava 给我们带来了一种新的编程思想，异步编程由此变得更加简单。笔者整理出的 RxJava 2 的思维导图如图 1-1 所示，本书后面的章节会对这些内容分别讲述。

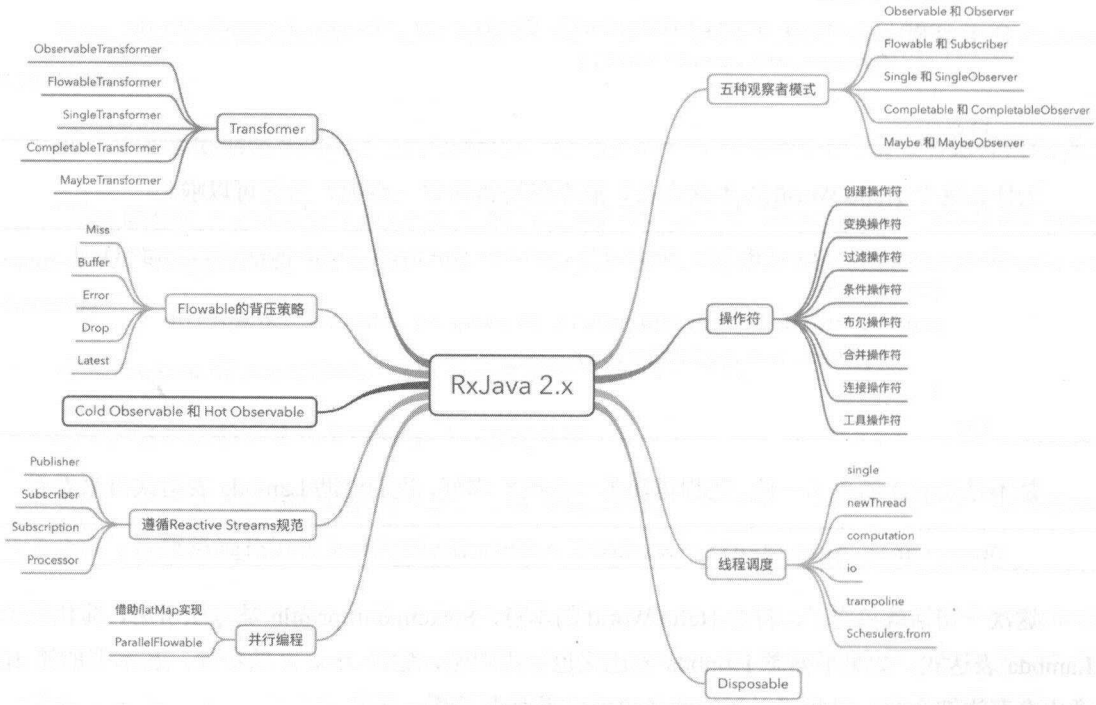


图 1-1

第2章 RxJava基础知识

2.1 Observable

RxJava 的使用通常需要三步。

（1）创建 Observable

Observable 的字面意思是被观察者，使用 RxJava 时需要创建一个被观察者，它会决定什么时候触发事件以及触发怎样的事件。有点类似上游发送命令，可以在这里决定异步操作模块的顺序和异步操作模块的次数。

（2）创建 Observer

Observer 即观察者，它可以在不同的线程中执行任务。这种模式可以极大地简化并发操作，因为它创建了一个处于待命状态的观察者哨兵，可以在未来某个时刻响应 Observable 的通知，而不需要阻塞等待 Observable 发射数据。

（3）使用 subscribe()进行订阅

创建了 Observable 和 Observer 之后，我们还需要使用 subscribe()方法将它们连接起来，这样整个上下游就能衔接起来实现链式调用。还记得第 1 章那个 RxJava 的 Hello World 吗？

```
Observable.just("Hello World").subscribe(new Consumer<String>() {
    @Override
    public void accept(@NonNull String s) throws Exception {
        System.out.println(s);
```

```
        }
    });
```

just()是 RxJava 的创建操作符，用于创建一个 Observable。Consumer 是消费者，用于接收单个值。熟悉 Java 8 的读者可以看到它与 java.util.function.Consumer 类似，RxJava 2 的命名规范参照 Java 8。

subscribe 有多个重载的方法。 第一个就是刚才的“Hello World”版本。

```
subscribe(onNext)
subscribe(onNext,onError)
subscribe(onNext,onError,onComplete)
subscribe(onNext,onError,onComplete,onSubscribe)
```

接下来，我们来看一个重载方法的版本，subscribe(onNext,onError,onComplete)。

```
    Observable.just("Hello World")
            .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println(throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("onComplete()");
                }
            });
```

执行结果。

```
Hello World
onComplete()
```

此时，除了打印了“Hello World”，还打印了“onComplete()”。在 subscribe 中 onComplete 是执行完 onNext 之后再执行的。onComplete 是一个 Action，它与 Consumer 的区别如下。

◎ Action：无参数类型。

◎ Consumer：单一参数类型。

再来看一个重载方法的版本，subscribe(onNext,onError,onComplete,onSubscribe)。

```
Observable.just("Hello World")
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String s) throws Exception {
                        System.out.println(s);
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        System.out.println(throwable.getMessage());
                    }
                }, new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("onComplete()");
                    }
                }, new Consumer<Disposable>() {
                    @Override
                    public void accept(Disposable disposable) throws Exception {
                        System.out.println("subscribe");
                    }
                });
```

执行结果：

```
subscribe
Hello World
onComplete()
```

它先打印了“subscribe”，接着打印了“Hello World”和“onComplete()”。这次先执行了 onSubscribe，再执行了 onNext 和 onComplete。千万不要着急，一会儿先介绍 do 操作符，do 操作符涉及 RxJava 内部的数据流向。

在 RxJava 2 中，Observable 不再支持订阅 Subscriber，而是需要使用 Observer 作为观察者，下面代码使用的 Observer 就等价于刚才的代码。

```
Observable.just("Hello World")
        .subscribe(new Observer<String>() {
            @Override
            public void onSubscribe(Disposable d) {
                System.out.println("subscribe");
            }

            @Override
            public void onNext(String s) {
                System.out.println(s);
            }

            @Override
            public void onError(Throwable e) {
                System.out.println(e.getMessage());
            }

            @Override
            public void onComplete() {
                System.out.println("onComplete()");
            }
        });
```

执行结果：

```
subscribe
Hello World
onComplete()
```

在 RxJava 中，被观察者、观察者、subscribe()方法三者缺一不可。只有使用了 subscribe()，被观察者才会开始发送数据，这一点极为重要。

RxJava 2 的 5 种观察者模式如图 2-1 所示。

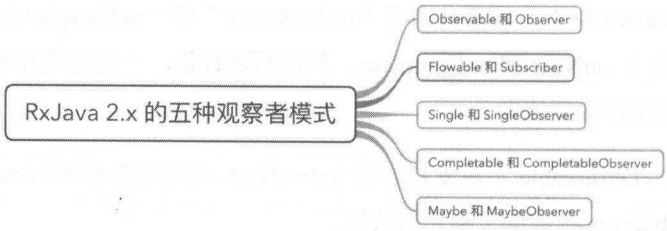


图 2-1

本章的前两个小节主要讲述 Observable 的相关内容。2.3 节以及本书的第 8 章主要讲述 Flowable。其余的三种被观察者模式会在 2.4 节中讲述。对 5 种观察者模式的描述如表 2-1 所示。

表 2-1

类　型	描　述
Observable	能够发射 0 或 *n* 个数据，并以成功或错误事件终止
Flowable	能够发射 0 或 *n* 个数据，并以成功或错误事件终止。支持背压，可以控制数据源发射的速度
Single	只发射单个数据或错误事件
Completable	从来不发射数据，只处理 onComplete 和 onError 事件。可以看成 Rx 的 Runnable
Maybe	能够发射 0 或者 1 个数据，要么成功，要么失败。有点类似于 Optional

从表 2-1 中可以看出，5 种被观察者类型中只有 Flowable 支持背压，如果有需要背压的情况，则必须使用 Flowable。

do 操作符

do 操作符可以给 Observable 的生命周期的各个阶段加上一系列的回调监听，当 Observable 执行到这个阶段时，这些回调就会被触发。在 RxJava 中包含了很多的 doXXX 操作符。

例如下面的代码，基本包含了 Observable 的完整生命周期。

```
Observable.just("Hello")
        .doOnNext(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("doOnNext: " + s);
            }
        })
        .doAfterNext(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("doAfterNext: " + s);
            }
        })
        .doOnComplete(new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("doOnComplete: ");
            }
```

```
                })
                //订阅之后回调的方法
                .doOnSubscribe(new Consumer<Disposable>() {
                    @Override
                    public void accept(@NonNull Disposable disposable) throws
Exception {
                        System.out.println("doOnSubscribe: ");
                    }
                })
                .doAfterTerminate(new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("doAfterTerminate: ");
                    }
                })
                .doFinally(new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("doFinally: ");
                    }
                })
                //Observable 每发射一个数据就会触发这个回调，不仅包括 onNext，还包括
//onError 和 onCompleted
                .doOnEach(new Consumer<Notification<String>>() {
                    @Override
                    public void accept(@NonNull Notification<String>
stringNotification) throws Exception {
                        System.out.println("doOnEach:
"+(stringNotification.isOnNext()?"onNext":stringNotification.isOnComplete()
?"onComplete":"onError"));
                    }
                })
                //订阅后可以取消订阅
                .doOnLifecycle(new Consumer<Disposable>() {
                    @Override
                    public void accept(@NonNull Disposable disposable) throws
Exception {
                        System.out.println("doOnLifecycle:
"+disposable.isDisposed());
                    }
                }, new Action() {
                    @Override
```

```
            public void run() throws Exception {
                System.out.println("doOnLifecycle run: ");
            }
        })
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("收到消息: " + s);
            }
        });
```

执行结果：

```
doOnSubscribe:
doOnLifecycle: false
doOnNext: Hello
doOnEach: onNext
收到消息: Hello
doAfterNext: Hello
doOnComplete:
doOnEach: onComplete
doFinally:
doAfterTerminate:
```

执行结果显示了 RxJava 的内部数据流向。最开始是 doOnSubscribe，等到观察者消费完之后，会执行 doFinally、doAfterTerminate。表 2-2 总结了一些常用的 do 操作符的用途。

表 2-2

操作符	用　途
doOnSubscribe	一旦观察者订阅了 Observable，它就会被调用
doOnLifecycle	可以在观察者订阅之后，设置是否取消订阅
doOnNext	它产生的 Observable 每发射一项数据就会调用它一次，它的 Consumer 接受发射的数据项。一般用于在 subscribe 之前对数据进行处理
doOnEach	它产生的 Observable 每发射一项数据就会调用它一次，不仅包括 onNext，还包括 onError 和 onCompleted
doAfterNext	在 onNext 之后执行，而 doOnNext()是在 onNext 之前执行
doOnComplete	当它产生的 Observable 在正常终止调用 onComplete 时会被调用

续表

操作符	用　途
doFinally	在当它产生的 Observable 终止之后会被调用，无论是正常终止还是异常终止。doFinally 优先于 doAfterTerminate 的调用
doAfterTerminate	注册一个 Action，当 Observable 调用 onComplete 或 onError 时触发

2.2 Hot Observable 和 Cold Observable

1. Observable 的分类

在 RxJava 中，Observable 有 Hot 和 Cold 之分。

Hot Observable 无论有没有观察者进行订阅，事件始终都会发生。当 Hot Observable 有多个订阅者时（多个观察者进行订阅时），Hot Observable 与订阅者们的关系是一对多的关系，可以与多个订阅者共享信息。

Cold Observable 是只有观察者订阅了，才开始执行发射数据流的代码。并且 Cold Observable 和 Observer 只能是一对一的关系。当有多个不同的订阅者时，消息是重新完整发送的。也就是说，对 Cold Observable 而言，有多个 Observer 的时候，它们各自的事件是独立的。

下面更加形象地说明 Cold 和 Hot 的区别：

Learning Reactive Programming with Java 8 的作者 Nickolay Tsvetinov 说过，把一个 Hot Observable 想象成一个广播电台，所有在此刻收听的听众都会听到同一首歌。

而 Cold Observable 是一张音乐 CD，人们可以独立购买并听取它。

2. Cold Observable

Observable 的 just、creat、range、fromXXX 等操作符都能生成 Cold Observable，在本书的第 3 章中会单独讲解各个操作符的使用。

```
Consumer<Long> subscriber1 = new Consumer<Long>() {
    @Override
    public void accept(@NonNull Long aLong) throws Exception {
        System.out.println("subscriber1: "+aLong);
    }
```

```
        };

        Consumer<Long> subscriber2 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("   subscriber2: "+aLong);
            }
        };

        Observable<Long> observable = Observable.create(new
ObservableOnSubscribe<Long>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Long> e) throws
Exception {
                Observable.interval(10,
TimeUnit.MILLISECONDS,Schedulers.computation())
                        .take(Integer.MAX_VALUE)
                        .subscribe(e::onNext);
            }
        }).observeOn(Schedulers.newThread());

        observable.subscribe(subscriber1);
        observable.subscribe(subscriber2);

        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果。

```
subscriber1: 0
   subscriber2: 0
subscriber1: 1
   subscriber2: 1
subscriber1: 2
   subscriber2: 2
   subscriber2: 3
subscriber1: 3
subscriber1: 4
   subscriber2: 4
```

```
   subscriber2: 5
subscriber1: 5
subscriber1: 6
   subscriber2: 6
subscriber1: 7
   subscriber2: 7
subscriber1: 8
   subscriber2: 8
subscriber1: 9
   subscriber2: 9
```

可以看出，subscriber1 和 subscriber2 的结果并不一定是相同的，它们二者是完全独立的。create 操作符创建的 Observable 是 Cold Observable。

尽管 Cold Observable 很好，但是对于某些事件不确定何时发生及不确定 Observable 发射的元素数量的情况，还需使用 Hot Observable。比如，UI 交互的事件、网络环境的变化、地理位置的变化、服务器推送消息的到达等。

3. Cold Observable 如何转换成 Hot Observable

（1）使用 publish，生成 ConnectableObservable。

使用 publish 操作符，可以让 Cold Observable 转换成 Hot Observable，它将原先的 Observable 转换成 ConnectableObservable。

来看看刚才的例子。

```
Consumer<Long> subscriber1 = new Consumer<Long>() {
    @Override
    public void accept(@NonNull Long aLong) throws Exception {
        System.out.println("subscriber1: "+aLong);
    }
};

Consumer<Long> subscriber2 = new Consumer<Long>() {
    @Override
    public void accept(@NonNull Long aLong) throws Exception {
        System.out.println("   subscriber2: "+aLong);
    }
};
```

```
        Consumer<Long> subscriber3 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("      subscriber3: "+aLong);
            }
        };

        ConnectableObservable<Long> observable = Observable.create(new
ObservableOnSubscribe<Long>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Long> e) throws
Exception {
                Observable.interval(10,
TimeUnit.MILLISECONDS,Schedulers.computation())
                        .take(Integer.MAX_VALUE)
                        .subscribe(e::onNext);
            }
        }).observeOn(Schedulers.newThread()).publish();
        observable.connect();

        observable.subscribe(subscriber1);
        observable.subscribe(subscriber2);

        try {
            Thread.sleep(20L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        observable.subscribe(subscriber3);

        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

注意，生成的 ConnectableObservable 需要调用 connect()才能真正执行。

执行结果：

```
subscriber1: 0
   subscriber2: 0
subscriber1: 1
   subscriber2: 1
subscriber1: 2
   subscriber2: 2
      subscriber3: 2
subscriber1: 3
   subscriber2: 3
      subscriber3: 3
subscriber1: 4
   subscriber2: 4
      subscriber3: 4
subscriber1: 5
   subscriber2: 5
      subscriber3: 5
subscriber1: 6
   subscriber2: 6
      subscriber3: 6
subscriber1: 7
   subscriber2: 7
      subscriber3: 7
subscriber1: 8
   subscriber2: 8
      subscriber3: 8
subscriber1: 9
   subscriber2: 9
      subscriber3: 9
subscriber1: 10
   subscriber2: 10
      subscriber3: 10
subscriber1: 11
   subscriber2: 11
      subscriber3: 11
```

可以看到，多个订阅的 subscriber（或者说观察者）共享同一事件。在这里，ConnectableObservable 是线程安全的。

（2）使用 Subject/Processor。

Subject 和 Processor 的作用相同。Processor 是 RxJava 2.x 新增的类，继承自 Flowable，

支持背压控制（Back Presure），而 Subject 则不支持背压控制。

```
        Consumer<Long> subscriber1 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("subscriber1: "+aLong);
            }
        };

        Consumer<Long> subscriber2 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("   subscriber2: "+aLong);
            }
        };

        Consumer<Long> subscriber3 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("      subscriber3: "+aLong);
            }
        };

        Observable<Long> observable = Observable.create(new
ObservableOnSubscribe<Long>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Long> e) throws
Exception {
                Observable.interval(10,
TimeUnit.MILLISECONDS,Schedulers.computation())
                        .take(Integer.MAX_VALUE)
                        .subscribe(e::onNext);
            }
        }).observeOn(Schedulers.newThread());

        PublishSubject<Long> subject = PublishSubject.create();
        observable.subscribe(subject);

        subject.subscribe(subscriber1);
        subject.subscribe(subscriber2);

        try {
```

```
            Thread.sleep(20L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        subject.subscribe(subscriber3);

        try {
            Thread.sleep(100L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果与上面使用 publish 操作符相同。

Subject 既是 Observable，又是 Observer（Subscriber）。这一点可以从 Subject 的源码上看到，继承自 Observable，实现 Observer。

```
import io.reactivex.*;
import io.reactivex.annotations.*;

/**
 * Represents an Observer and an Observable at the same time, allowing
 * multicasting events from a single source to multiple child Subscribers.
 * <p>All methods except the onSubscribe, onNext, onError and onComplete are
thread-safe.
 * Use {@link #toSerialized()} to make these methods thread-safe as well.
 *
 * @param <T> the item value type
 */
public abstract class Subject<T> extends Observable<T> implements Observer<T>
{
    /**
     * Returns true if the subject has any Observers.
     * <p>The method is thread-safe.
     * @return true if the subject has any Observers
     */
    public abstract boolean hasObservers();

    /**
```

```
     * Returns true if the subject has reached a terminal state through an error
event.
     * <p>The method is thread-safe.
     * @return true if the subject has reached a terminal state through an error
event
     * @see #getThrowable()
     * &see {@link #hasComplete()}
     */
    public abstract boolean hasThrowable();

    /**
     * Returns true if the subject has reached a terminal state through a complete
event.
     * <p>The method is thread-safe.
     * @return true if the subject has reached a terminal state through a complete
event
     * @see #hasThrowable()
     */
    public abstract boolean hasComplete();

    /**
     * Returns the error that caused the Subject to terminate or null if the Subject
     * hasn't terminated yet.
     * <p>The method is thread-safe.
     * @return the error that caused the Subject to terminate or null if the Subject
     * hasn't terminated yet
     */
    @Nullable
    public abstract Throwable getThrowable();

    /**
     * Wraps this Subject and serializes the calls to the onSubscribe, onNext,
onError and
     * onComplete methods, making them thread-safe.
     * <p>The method is thread-safe.
     * @return the wrapped and serialized subject
     */
    @NonNull
    public final Subject<T> toSerialized() {
        if (this instanceof SerializedSubject) {
            return this;
        }
```

```
        return new SerializedSubject<T>(this);
    }
}
```

Subject 作为观察者，可以订阅目标 Cold Observable，使对方开始发送事件。同时它又作为 Observable 转发或者发送新的事件，让 Cold Observable 借助 Subject 转换为 Hot Observable。

Subject 并不是线程安全的，如果想要其线程安全，则需要调用 `toSerialized()` 方法（在 RxJava 1.x 中还可以用 SerializedSubject 代替 Subject，但是在 RxJava 2.x 之后，SerializedSubject 不再是一个 public class）。

然而，很多基于 EventBus 改造的 RxBus 并没有这么做。这样的做法是非常危险的，会遇到并发的情况。

4. Hot Observable 如何转换成 Cold Observable

（1）ConnectableObservable 的 refCount 操作符

在 ReactiveX 官网的解释是：make a Connectable Observable behave like an ordinary Observable，如图 2-2 所示。

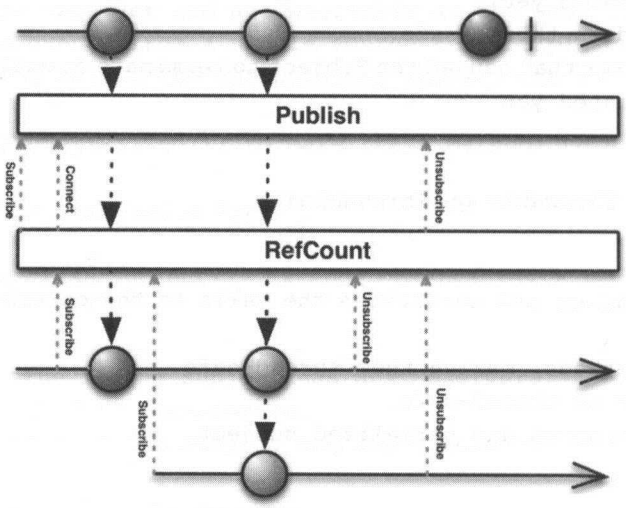


图 2-2

RefCount 操作符把从一个可连接的 Observable 连接和断开的过程自动化了。它操作一个

可连接的 Observable，返回一个普通的 Observable。当第一个订阅者/观察者订阅这个 Observable 时，RefCount 连接到下层的可连接 Observable。RefCount 跟踪有多少个观察者订阅它，直到最后一个观察者完成，才断开与下层可连接 Observable 的连接。

如果所有的订阅者/观察者都取消订阅了，则数据流停止；如果重新订阅，则重新开始数据流。

```
        Consumer<Long> subscriber1 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("subscriber1: "+aLong);
            }
        };

        Consumer<Long> subscriber2 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("   subscriber2: "+aLong);
            }
        };

        ConnectableObservable<Long> connectableObservable =
Observable.create(new ObservableOnSubscribe<Long>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Long> e) throws
Exception {
                Observable.interval(10,
TimeUnit.MILLISECONDS,Schedulers.computation())
                        .take(Integer.MAX_VALUE)
                        .subscribe(e::onNext);
            }
        }).observeOn(Schedulers.newThread()).publish();
        connectableObservable.connect();

        Observable<Long> observable = connectableObservable.refCount();

        Disposable disposable1 = observable.subscribe(subscriber1);
        Disposable disposable2 = observable.subscribe(subscriber2);

        try {
            Thread.sleep(20L);
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        disposable1.dispose();
        disposable2.dispose();

        System.out.println("重新开始数据流");

        disposable1 = observable.subscribe(subscriber1);
        disposable2 = observable.subscribe(subscriber2);

        try {
            Thread.sleep(20L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果：

```
subscriber1: 0
  subscriber2: 0
subscriber1: 1
  subscriber2: 1
```

重新开始数据流：

```
subscriber1: 0
  subscriber2: 0
subscriber1: 1
  subscriber2: 1
```

如果不是所有的订阅者/观察者都取消了订阅，而只是部分取消，则部分的订阅者/观察者重新开始订阅时，不会从头开始数据流。

```
        Consumer<Long> subscriber1 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("subscriber1: "+aLong);
            }
        };
```

```
        Consumer<Long> subscriber2 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("   subscriber2: "+aLong);
            }
        };

        Consumer<Long> subscriber3 = new Consumer<Long>() {
            @Override
            public void accept(@NonNull Long aLong) throws Exception {
                System.out.println("      subscriber3: "+aLong);
            }
        };

        ConnectableObservable<Long> connectableObservable =
Observable.create(new ObservableOnSubscribe<Long>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Long> e) throws
Exception {
                Observable.interval(10,
TimeUnit.MILLISECONDS,Schedulers.computation())
                        .take(Integer.MAX_VALUE)
                        .subscribe(e::onNext);
            }
        }).observeOn(Schedulers.newThread()).publish();
        connectableObservable.connect();

        Observable<Long> observable = connectableObservable.refCount();

        Disposable disposable1 = observable.subscribe(subscriber1);
        Disposable disposable2 = observable.subscribe(subscriber2);
        observable.subscribe(subscriber3);

        try {
            Thread.sleep(20L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        disposable1.dispose();
        disposable2.dispose();
```

```
        System.out.println("subscriber1、subscriber2 重新订阅");

        disposable1 = observable.subscribe(subscriber1);
        disposable2 = observable.subscribe(subscriber2);

        try {
            Thread.sleep(20L);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果：

```
subscriber1: 0
   subscriber2: 0
      subscriber3: 0
subscriber1: 1
   subscriber2: 1
      subscriber3: 1
subscriber1、subscriber2 重新订阅
      subscriber3: 2
subscriber1: 2
   subscriber2: 2
      subscriber3: 3
subscriber1: 3
   subscriber2: 3
      subscriber3: 4
subscriber1: 4
   subscriber2: 4
```

在这里，subscriber1 和 subscriber2 先取消了订阅，subscriber3 并没有取消订阅。之后，subscriber1 和 subscriber2 又重新订阅。最终 subscriber1、subscriber2、subscriber3 的值保持一致。

（2）Observable 的 share 操作符。

share 操作符封装了 publish().refCount()调用，可以看其源码。

```
    /**
     * Returns a new {@link ObservableSource} that multicasts (shares) the
original {@link ObservableSource}. As long as
     * there is at least one {@link Observer} this {@link ObservableSource} will
be subscribed and emitting data.
```

```
    * When all subscribers have disposed it will dispose the source {@link
ObservableSource}.
    * <p>
    * This is an alias for {@link #publish()}.{@link
ConnectableObservable#refCount()}.
    * <p>
    * ![](http://upload-images.jianshu.io/upload_images/2613397-81dcef165b6
9aca2.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)
    * <dl>
    *  <dt><b>Scheduler:</b></dt>
    *  <dd>{@code share} does not operate by default on a particular {@link
Scheduler}.</dd>
    * </dl>
    *
    * @return an {@code ObservableSource} that upon connection causes the source
{@code ObservableSource} to emit items
    *         to its {@link Observer}s
    * @see <a
href="http://reactivex.io/documentation/operators/refcount.html">ReactiveX
operators documentation: RefCount</a>
    */
   @CheckReturnValue
   @SchedulerSupport(SchedulerSupport.NONE)
   public final Observable<T> share() {
       return publish().refCount();
   }
```

2.3　Flowable

在 RxJava 2.x 中，Observable 不再支持背压（Back Pressure），而改由 Flowable 来支持非阻塞式的背压。Flowable 是 RxJava 2.x 新增的被观察者，Flowable 可以看成 Observable 新的实现，它支持背压，同时实现 Reactive Streams 的 Publisher 接口。Flowable 所有的操作符强制支持背压，不过幸运的是，Flowable 中的操作符大多与 Observable 类似。

虽然 Flowable 跟 Observable 很相似，但是两者在使用场景上还是有区别的。

使用 Observable 较好的场景如下：

◎　一般处理最大不超过 1000 条数据，并且几乎不会出现内存溢出；

◎ GUI 鼠标事件，基本不会背压（可以结合 sampling/debouncing 操作）；
◎ 处理同步流。

使用 Flowable 较好的场景如下：

◎ 处理以某种方式产生超过 10KB 的元素；
◎ 文件读取与分析；
◎ 读取数据库记录，也是一个阻塞的和基于拉取模式；
◎ 网络 I/O 流；
◎ 创建一个响应式非阻塞接口。

另外，在本书的第 8 章，会详细介绍背压，以及 RxJava 2 的背压策略。

2.4 Single、Completable 和 Maybe

通常情况下，如果想要使用 RxJava，首先会想到的是使用 Observable，如果考虑到背压的情况，则在 RxJava 2.x 下会使用 Flowable。除 Observable 和 Flowable 外，在 RxJava 2.x 中还有 3 种类型的被观察者：Single、Completable 和 Maybe。

1. Single

从 SingleEmitter 的源码可以看出，Single 只有 onSuccess 和 onError 事件。

```
/**
 * Copyright (c) 2016-present, RxJava Contributors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not
use this file except in
 * compliance with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software distributed
under the License is
 * distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
either express or implied. See
 * the License for the specific language governing permissions and limitations
under the License.
```

```
 */

package io.reactivex;

import io.reactivex.annotations.*;
import io.reactivex.disposables.Disposable;
import io.reactivex.functions.Cancellable;

/**
 * Abstraction over an RxJava {@link SingleObserver} that allows associating
 * a resource with it.
 * <p>
 * All methods are safe to call from multiple threads.
 * <p>
 * Calling onSuccess or onError multiple times has no effect.
 *
 * @param <T> the value type to emit
 */
public interface SingleEmitter<T> {

    /**
     * Signal a success value.
     * @param t the value, not null
     */
    void onSuccess(@NonNull T t);

    /**
     * Signal an exception.
     * @param t the exception, not null
     */
    void onError(@NonNull Throwable t);

    /**
     * Sets a Disposable on this emitter; any previous Disposable
     * or Cancellation will be unsubscribed/cancelled.
     * @param s the disposable, null is allowed
     */
    void setDisposable(@Nullable Disposable s);

    /**
     * Sets a Cancellable on this emitter; any previous Disposable
     * or Cancellation will be unsubscribed/cancelled.
```

```
     * @param c the cancellable resource, null is allowed
     */
    void setCancellable(@Nullable Cancellable c);

    /**
     * Returns true if the downstream cancelled the sequence.
     * @return true if the downstream cancelled the sequence
     */
    boolean isDisposed();
}
```

其中，onSuccess()用于发射数据（在 Observable/Flowable 中使用 onNext()来发射数据），而且只能发射一个数据，后面即使再发射数据也不会做任何处理。

Single 的 SingleObserver 中只有 onSuccess 和 onError，并没有 onComplete。这也是 Single 与其他 4 种被观察者之间的最大区别。

```
Single.create(new SingleOnSubscribe<String>() {

            @Override
            public void subscribe(@NonNull SingleEmitter<String> e) throws
Exception {

                e.onSuccess("test");
            }
        }).subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println(s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                throwable.printStackTrace();
            }
        });
```

上面的代码，由于 Observer 中有两个 Consumer，因而还可以进一步简化。

```
Single.create(new SingleOnSubscribe<String>() {
```

```
            @Override
            public void subscribe(@NonNull SingleEmitter<String> e) throws
Exception {

                e.onSuccess("test");
            }
        }).subscribe(new BiConsumer<String, Throwable>() {
            @Override
            public void accept(String s, Throwable throwable) throws Exception
{
                System.out.println(s);
            }
        });
```

Single 可以通过 toXXX 方法转换成 Observable、Flowable、Completable 及 Maybe。

2. Completable

Completable 在创建后，不会发射任何数据。从 CompletableEmitter 的源码中可以看到。

```
/**
 * Copyright (c) 2016-present, RxJava Contributors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License"); you may not
use this file except in
 * compliance with the License. You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software distributed
under the License is
 * distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND,
either express or implied. See
 * the License for the specific language governing permissions and limitations
under the License.
 */

package io.reactivex;

import io.reactivex.annotations.*;
import io.reactivex.disposables.Disposable;
```

```
import io.reactivex.functions.Cancellable;

/**
 * Abstraction over an RxJava {@link CompletableObserver} that allows associating
 * a resource with it.
 * <p>
 * All methods are safe to call from multiple threads.
 * <p>
 * Calling onComplete or onError multiple times has no effect.
 */
public interface CompletableEmitter {

    /**
     * Signal the completion.
     */
    void onComplete();

    /**
     * Signal an exception.
     * @param t the exception, not null
     */
    void onError(@NonNull Throwable t);

    /**
     * Sets a Disposable on this emitter; any previous Disposable
     * or Cancellation will be disposed/cancelled.
     * @param d the disposable, null is allowed
     */
    void setDisposable(@Nullable Disposable d);

    /**
     * Sets a Cancellable on this emitter; any previous Disposable
     * or Cancellation will be disposed/cancelled.
     * @param c the cancellable resource, null is allowed
     */
    void setCancellable(@Nullable Cancellable c);

    /**
     * Returns true if the downstream disposed the sequence.
     * @return true if the downstream disposed the sequence
     */
    boolean isDisposed();
```

```
}
```

Completable 只有 onComplete 和 onError 事件，同时 Completable 并没有 map、flatMap 等操作符，它的操作符比起 Observable/Flowable 要少得多。

我们可以通过 fromXXX 操作符来创建一个 Completable。这是一个 Completable 版本的 Hello World。

```
Completable.fromAction(new Action() {
            @Override
            public void run() throws Exception {

                System.out.println("Hello World");
            }
        }).subscribe();
```

Completable 经常结合 andThen 操作符使用。

```
Completable.create(new CompletableOnSubscribe() {
            @Override
            public void subscribe(@NonNull CompletableEmitter emitter) throws
Exception {

                try {
                    TimeUnit.SECONDS.sleep(1);
                    emitter.onComplete();
                } catch (InterruptedException e) {
                    emitter.onError(e);
                }
            }
        }).andThen(Observable.range(1, 10))
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(@NonNull Integer integer) throws Exception {
                System.out.println(integer);
            }
        });
```

在这里，emitter.onComplete()执行完成之后，表明 Completable 已经执行完毕，接下来执行 andThen 里的操作。

执行结果。

```
1
2
3
4
5
6
7
8
9
10
```

在 Completable 中，andThen 有多个重载的方法，正好对应了 5 种被观察者的类型。

```
Completable       andThen(CompletableSource next)
<T> Maybe<T>      andThen(MaybeSource<T> next)
<T> Observable<T> andThen(ObservableSource<T> next)
<T> Flowable<T>   andThen(Publisher<T> next)
<T> Single<T>     andThen(SingleSource<T> next)
```

Completable 也可以通过 toXXX 方法转换成 Observable、Flowable、Single 以及 Maybe。

在网络操作中，如果遇到更新的情况，也就是 Restful 架构中的 PUT 操作，一般要么返回原先的对象，要么只提示更新成功。下面两个接口使用了 Retrofit 框架，分别用于获取短信验证码和更新用户信息，其中更新用户信息如果用 PUT 会更符合 RESTful 的 API。

```
/**
 * 获取短信验证码
 * @param param
 * @return
 */
@POST("v1/user-auth")
Completable getVerificationCode(@Body VerificationCodeParam param);

/**
 * 用户信息更新接口
 * @param param
 * @return
 */
@POST("v1/user-update")
```

```
    Completable update(@Body UpdateParam param);
```

在 model 类中大致会这样写。

```
/**
 * Created by Tony Shen on 2017/7/24.
 */

public class VerificationCodeModel extends HttpResponse {

    /**
     * 获取验证码
     * @param activity
     * @param param
     * @return
     */
    public Completable getVerificationCode(AppCompatActivity activity,
VerificationCodeParam param) {

        return apiService
                .getVerificationCode(param)
                .compose(RxJavaUtils.<VerificationCodeModel>completableToMain
())
                .compose(RxLifecycle.bind(activity).<VerificationCodeModel>to
LifecycleTransformer());
    }
}
```

需特别注意的是，getVerificationCode 返回的是 Completable 而不是 Completable<t>。< p="" style="box-sizing: border-box;"></t>。<>

获取验证码成功则给出相应的 toast 提示，失败则应做出相应的处理。

```
VerificationCodeModel model = new VerificationCodeModel();
model.getVerificationCode(RegisterActivity.this,param)
          .subscribe(new Action() {
                    @Override
                    public void run() throws Exception {
                              showShort(RegisterActivity.this,"发送验证码成功");
                    }
          },new RxException<Throwable>(){
                    @Override
```

```
                public void accept(@NonNull Throwable throwable) throws
Exception {
                    throwable.printStackTrace();
                    ......
                }
        });
```

执行结果如图 2-3 所示。

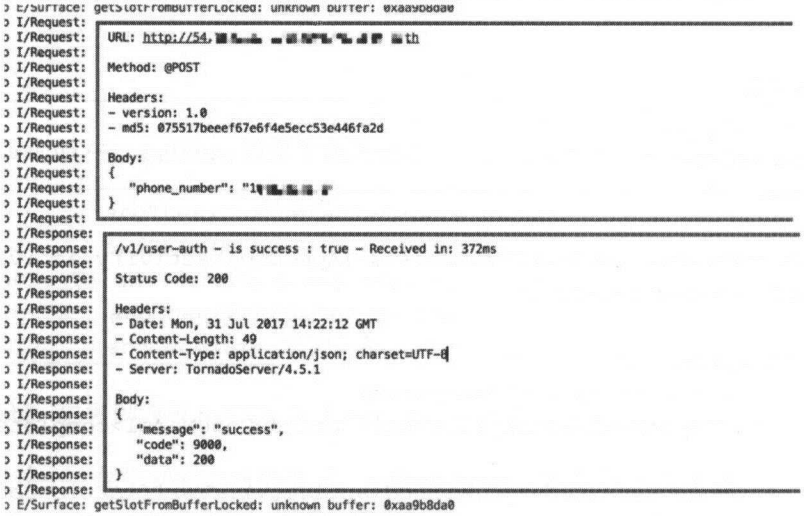


图 2-3

3. Maybe

Maybe 是 RxJava 2.x 之后才有的新类型，可以看成是 Single 和 Completable 的结合。

Maybe 创建之后，MaybeEmitter 和 SingleEmitter 一样，并没有 onNext()方法，同样需要通过 onSuccess()方法来发射数据。

```
Maybe.create(new MaybeOnSubscribe<String>() {

        @Override
        public void subscribe(@NonNull MaybeEmitter<String> e) throws
Exception {
            e.onSuccess("testA");
        }
    }).subscribe(new Consumer<String>() {
```

```
        @Override
        public void accept(@NonNull String s) throws Exception {

            System.out.println("s="+s);
        }
    });
```

打印出来的结果是：

```
s=testA
```

Maybe 也只能发射 0 或者 1 个数据，即使发射多个数据，后面发射的数据也不会处理。

```
Maybe.create(new MaybeOnSubscribe<String>() {

        @Override
        public void subscribe(@NonNull MaybeEmitter<String> e) throws
Exception {
            e.onSuccess("testA");
            e.onSuccess("testB");
        }
    }).subscribe(new Consumer<String>() {

        @Override
        public void accept(@NonNull String s) throws Exception {

            System.out.println("s="+s);
        }
    });
```

打印出来的结果仍然是：

```
s=testA
```

与第一次执行的结果相同。

如果 MaybeEmitter 先调用了 onComplete()，即使后面再调用 onSuccess()，也不会发射任何数据。

```
Maybe.create(new MaybeOnSubscribe<String>() {
```

```
            @Override
            public void subscribe(@NonNull MaybeEmitter<String> e) throws
Exception {
                e.onComplete();
                e.onSuccess("testA");
            }
        }).subscribe(new Consumer<String>() {

            @Override
            public void accept(@NonNull String s) throws Exception {

                System.out.println("s="+s);
            }
        });
```

这次就没有打印任何数据了。

我们对上面的代码再做一下修改，在 subscribe()中也加入 onComplete()，看看打印出来的结果会是怎样的？因为在 SingleObserver 中没有 onComplete()方法。

```
Maybe.create(new MaybeOnSubscribe<String>() {

            @Override
            public void subscribe(@NonNull MaybeEmitter<String> e) throws
Exception {
                e.onComplete();
                e.onSuccess("testA");
            }
        }).subscribe(new Consumer<String>() {

            @Override
            public void accept(@NonNull String s) throws Exception {

                System.out.println("s=" + s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{

            }
```

```
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Maybe onComplete");
            }
        });
```

这次打印的结果是：

```
Maybe onComplete
```

通过查看 Maybe 相关的源码：

```
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.NONE)
    public final Disposable subscribe(Consumer<? super T> onSuccess, Consumer<?
super Throwable> onError,
            Action onComplete) {
        ObjectHelper.requireNonNull(onSuccess, "onSuccess is null");
        ObjectHelper.requireNonNull(onError, "onError is null");
        ObjectHelper.requireNonNull(onComplete, "onComplete is null");
        return subscribeWith(new MaybeCallbackObserver<T>(onSuccess, onError,
onComplete));
    }
```

可以得到，Maybe 在没有数据发射时，subscribe 会调用 MaybeObserver 的 onComplete()。如果 Maybe 有数据发射或者调用了 onError()，则不会执行 MaybeObserver 的 onComplete()。

我们也可以将 Maybe 转换成 Observable、Flowable、Single，只需相应地调用 toObservable()、toFlowable()、toSingle()即可。

再来看看 Maybe 与 Retrofit 是怎样结合使用的？下面的网络请求，最初返回的类型是 Flowable，但是这个网络请求并不是一个连续事件流，我们只会发起一次 Post 请求返回数据，并且只收到一个事件。因此，可以考虑将 onComplete() 与 onNext() 合并。在这里，尝试将 Flowable 改成 Maybe。

```
    @POST("v1/contents")
    Maybe<ContentModel> loadContent(@Body ContentParam param);
```

在 model 类中，我们大致会这样写。

```
public class ContentModel extends HttpResponse {

    public List<ContentItem> data;

    /**
     * 获取内容
     * @param fragment
     * @param param
     * @param cacheKey
     * @return
     */
    public Maybe<ContentModel> getContent(Fragment fragment,ContentParam
param,String cacheKey) {

        return apiService.loadContent(param)
                .compose(RxLifecycle.bind(fragment).<ContentModel>toLifecycle
Transformer())
                .compose(RxJavaUtils.<ContentModel>maybeToMain())
                .compose(RxUtils.<ContentModel>toCacheTransformer(cacheKey,Ap
p.getInstance().cache));
    }

    ......
}
```

其中，maybeToMain()方法是用 Kotlin 编写的工具方法，这些工具方法用 Kotlin 来编写会显得比较简单且清晰，特别是 Lambda 表达式，更加直观。

```
@JvmStatic
    fun <T> maybeToMain(): MaybeTransformer<T, T> {

        return MaybeTransformer{
            upstream ->
            upstream.subscribeOn(Schedulers.io())
                    .observeOn(AndroidSchedulers.mainThread())
        }
    }
```

最后是真正地使用 model 类，如果网络请求成功则将数据展示到 Android 的 recyclerview 上，如果失败也会做出相应处理。

```
        model.getContent(this,param,cacheKey)
                .subscribe(new Consumer<ContentModel>() {
                    @Override
                    public void accept(@io.reactivex.annotations.NonNull
ContentModel model) throws Exception {

                        adapter = new NewsAdapter(mContext, model);
                        recyclerview.setAdapter(adapter);
                        spinKitView.setVisibility(View.GONE);
                    }
                }, new RxException<Throwable>() {
                    @Override
                    public void accept(@NonNull Throwable throwable) throws
Exception {
                        throwable.printStackTrace();
                        spinKitView.setVisibility(View.GONE);

                        ......
                    }
                });
```

发送网络请求的日志，如图 2-4 所示。

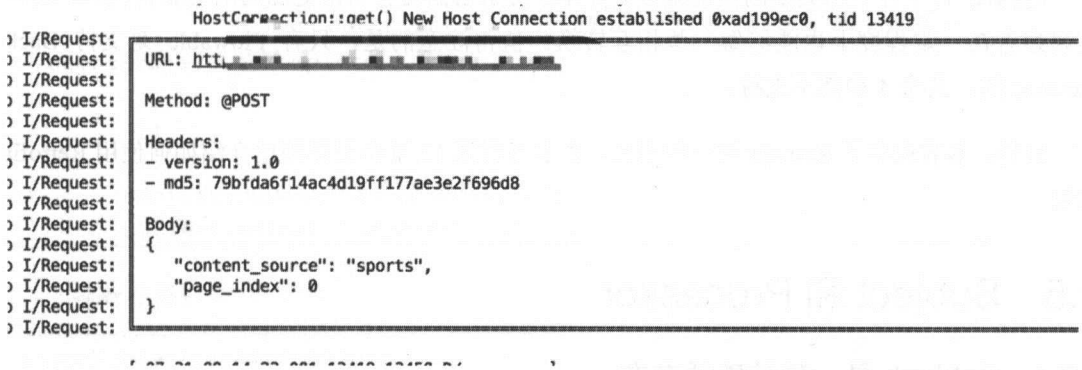


图 2-4

接受网络请求的日志，如图 2-5 所示。

图 2-5

4. 小结

RxJava 有 5 种不同类型的被观察者，合理地使用它们能够写出更加简洁优雅的代码。这些被观察者在一定程度上也能够做一些相互转换。值得注意的是，只有 Flowable 是支持 Back Pressure 的，其余 4 种都不支持。

另外，本节列举了 Retrofit 的一些用法，在本书的第 12 章会更详细地介绍如何使用 Retrofit 框架。

2.5 Subject 和 Processor

2.5.1 Subject 是一种特殊的存在

在 2.2 节中，我们曾介绍过 Subject 既是 Observable ，又是 Observer。官网称可以将 Subject 看作一个桥梁或者代理。

1. Subject 的分类

Subject 包含 4 种类型，分别是 AsyncSubject、BehaviorSubject、ReplaySubject 和 PublishSubject。

（1）AsyncSubject。

Observer 会接收 AsyncSubject 的 onComplete()之前的最后一个数据。

```
AsyncSubject<String> subject = AsyncSubject.create();
        subject.onNext("asyncSubject1");
        subject.onNext("asyncSubject2");
        subject.onComplete();
        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("asyncSubject:"+s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("asyncSubject onError");  //不输出
//（异常才会输出）
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("asyncSubject:complete");  //输出
//asyncSubject onComplete
            }
        });

        subject.onNext("asyncSubject3");
        subject.onNext("asyncSubject4");
```

执行结果：

```
asyncSubject:asyncSubject2
asyncSubject:complete
```

改一下代码，将 subject.onComplete()放在最后。

```
        AsyncSubject<String> subject = AsyncSubject.create();
        subject.onNext("asyncSubject1");
        subject.onNext("asyncSubject2");
```

```
        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("asyncSubject:"+s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("asyncSubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("asyncSubject:complete");  //输出
//asyncSubject onComplete
            }
        });

        subject.onNext("asyncSubject3");
        subject.onNext("asyncSubject4");
        subject.onComplete();
```

执行结果：

```
asyncSubject:asyncSubject4
asyncSubject:complete
```

注意，subject.onComplete()必须要调用才会开始发送数据，否则观察者将不接收任何数据。

（2）BehaviorSubject。

Observer 会先接收到 BehaviorSubject 被订阅之前的最后一个数据，再接收订阅之后发射过来的数据。如果 BehaviorSubject 被订阅之前没有发送任何数据，则会发送一个默认数据。

```
        BehaviorSubject<String> subject = BehaviorSubject.createDefault
("behaviorSubject1");

        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
```

```
                System.out.println("behaviorSubject:"+s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("behaviorSubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("behaviorSubject:complete");  //输出
//behaviorSubject onComplete
            }
        });

        subject.onNext("behaviorSubject2");
        subject.onNext("behaviorSubject3");
```

执行结果。

```
behaviorSubject:behaviorSubject1
behaviorSubject:behaviorSubject2
behaviorSubject:behaviorSubject3
```

在这里，behaviorSubject1 是默认值。因为执行了下面的代码。

```
BehaviorSubject<String> subject = BehaviorSubject.createDefault
("behaviorSubject1");
```

稍微改一下代码，在 subscribe()之前，再发射一个事件。

```
      BehaviorSubject<String> subject =
BehaviorSubject.createDefault("behaviorSubject1");
        subject.onNext("behaviorSubject2");

        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("behaviorSubject:"+s);  //输出
asyncSubject:asyncSubject3
```

```
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("behaviorSubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("behaviorSubject:complete");  //输出
//behaviorSubject onComplete
            }
        });

        subject.onNext("behaviorSubject3");
        subject.onNext("behaviorSubject4");
```

执行结果：

```
behaviorSubject:behaviorSubject2
behaviorSubject:behaviorSubject3
behaviorSubject:behaviorSubject4
```

这次丢弃了默认值，而发射了 behaviorSubject2。因为 BehaviorSubject 每次只会发射调用 subscribe()方法之前的最后一个事件和调用 subscribe()方法之后的事件。

BehaviorSubject 还可以缓存最近一次发出信息的数据。

（3）ReplaySubject。

ReplaySubject 会发射所有来自原始 Observable 的数据给观察者，无论它们是何时订阅的。

```
    ReplaySubject<String> subject = ReplaySubject.create();
     subject.onNext("replaySubject1");
     subject.onNext("replaySubject2");

     subject.subscribe(new Consumer<String>() {
         @Override
         public void accept(@NonNull String s) throws Exception {
             System.out.println("replaySubject:"+s);
```

```
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("replaySubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("replaySubject:complete");  //输出
//replaySubject onComplete
            }
        });

        subject.onNext("replaySubject3");
        subject.onNext("replaySubject4");
```

执行结果。

```
replaySubject:replaySubject1
replaySubject:replaySubject2
replaySubject:replaySubject3
replaySubject:replaySubject4
```

稍微改一下代码，将 create()改成 createWithSize(1)，表示只缓存订阅前最后发送的一条数据。

```
        ReplaySubject<String> subject = ReplaySubject.createWithSize(1);
        subject.onNext("replaySubject1");
        subject.onNext("replaySubject2");

        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("replaySubject:"+s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
```

```
                System.out.println("replaySubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("replaySubject:complete");  //输出
//replaySubject onComplete
            }
        });

        subject.onNext("replaySubject3");
        subject.onNext("replaySubject4");
```

执行结果：

```
replaySubject:replaySubject2
replaySubject:replaySubject3
replaySubject:replaySubject4
```

这个执行结果与 BehaviorSubject 的相同。但是从并发的角度来看，ReplaySubject 在处理并发 subscribe()和 onNext() 时会更加复杂。

ReplaySubject 除了可以限制缓存数据的数量，还能限制缓存的时间，使用 createWithTime() 即可。

（4）PublishSubject。

Observer 只接收 PublishSubject 被订阅之后发送的数据。

```
        PublishSubject<String> subject = PublishSubject.create();
        subject.onNext("publicSubject1");
        subject.onNext("publicSubject2");
        subject.onComplete();

        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("publicSubject:"+s);
            }
        }, new Consumer<Throwable>() {
```

```
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("publicSubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("publicSubject:complete");  //输出
//publicSubject onComplete
            }
        });

        subject.onNext("publicSubject3");
        subject.onNext("publicSubject4");
```

执行结果：

```
publicSubject:complete
```

因为 subject 在订阅之前已经执行了 onComplete()方法，所以无法发射数据。稍微改一下代码，将 onComplete()方法放在最后。

```
        PublishSubject<String> subject = PublishSubject.create();
        subject.onNext("publicSubject1");
        subject.onNext("publicSubject2");

        subject.subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                System.out.println("publicSubject:"+s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{
                System.out.println("publicSubject onError");  //不输出
//(异常才会输出)
            }
        }, new Action() {
            @Override
```

```
            public void run() throws Exception {
                System.out.println("publicSubject:complete");  //输出
//publicSubject onComplete
            }
        });

        subject.onNext("publicSubject3");
        subject.onNext("publicSubject4");
        subject.onComplete();
```

执行结果：

```
publicSubject:publicSubject3
publicSubject:publicSubject4
publicSubject:complete
```

最后，用表 2-4 总结这 4 个 Subject 的特性。

表 2-4

Subject	发 射 行 为
AsyncSubject	不论订阅发生在什么时候，只发射最后一个数据
BehaviorSubject	发送订阅之前的一个数据和订阅之后的全部数据
ReplaySubject	不论订阅发生在什么时候，都发射全部数据
PublishSubject	发送订阅之后全部数据

（5）可能错过的事件。

Subject作为一个Observable时，可以不停地调用onNext()来发送事件，直至遇到onComplete()才会结束。

```
PublishSubject<String> subject = PublishSubject.create();
        subject.subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        System.out.println(s);
                    }
                }, new Consumer<Throwable>() {

                    @Override
```

```
                public void accept(@NonNull Throwable throwable) throws
Exception {

                }
            },new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("completed");
                }
            });
        subject.onNext("Foo");
        subject.onNext("Bar");
        subject.onComplete();
```

执行结果：

```
Foo
Bar
completed
```

如果使用 subsribeOn 操作符将 subject 切换到 I/O 线程，则可以使用 Thread.sleep(2000) 让主线程休眠 2s。

```
PublishSubject<String> subject = PublishSubject.create();
        subject.subscribeOn(Schedulers.io())
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        System.out.println(s);
                    }
                }, new Consumer<Throwable>() {

                    @Override
                    public void accept(@NonNull Throwable throwable) throws
Exception {

                    }
                },new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("completed");
                    }
```

```
        });
        subject.onNext("Foo");
        subject.onNext("Bar");
        subject.onComplete();
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

这时，其执行的结果变为：

```
completed
```

为何会没有打印 Foo 和 Bar?

因为subject发射元素的线程被指派到了I/O线程，此时I/O线程正在初始化还没起来，subject发射前，Foo 和 Bar 这两个元素还在主线程中，而在从主线程往 I/O 线程转发的过程中，由于I/O 线程还没有起来，所以就被丢弃了。此时，无论 Thread 睡了多少秒，Foo、Bar 都不会被打印出来。

其实，解决方法很简单，使用 Observable.create()来替代 subject，它允许为每个订阅者精确控制事件的发送，这样就不会少打印 Foo 和 Bar 了。

（6）使用 PublishSubject 实现简化的 RxBus。

下面的代码实现了一个简化版本的 Android EventBus，在这里使用了 PublishSubject。因为事件总线是基于 发布/订阅模式 实现的，如果某一事件在多个 Activity/Fragment 中被订阅，则在 App 的任意地方一旦发布该事件，多个订阅的地方均能收到这一事件（在这里，订阅事件的 Activity/Fragment 不能被损坏，一旦被损坏就不能收到事件），这很符合 Hot Observable 的特性。所以，我们使用 PublishSubject，考虑到多线程的情况，还需要使用 Subject 的 toSerialized() 方法。

```
import io.reactivex.Observable;
import io.reactivex.subjects.PublishSubject;
import io.reactivex.subjects.Subject;

public class RxBus {
```

```
    private final Subject<Object> mBus;

    private RxBus() {
        mBus = PublishSubject.create().toSerialized();
    }

    public static RxBus get() {
        return Holder.BUS;
    }

    public void post(Object obj) {
        mBus.onNext(obj);
    }

    public <T> Observable<T> toObservable(Class<T> tClass) {
        return mBus.ofType(tClass);
    }

    public Observable<Object> toObservable() {
        return mBus;
    }

    public boolean hasObservers() {
        return mBus.hasObservers();
    }

    private static class Holder {
        private static final RxBus BUS = new RxBus();
    }
}
```

在这里，Subject 的 toSerialized()，使用 SerializedSubject 包装了原先的 Subject。

```
    /**
     * Wraps this Subject and serializes the calls to the onSubscribe, onNext,
onError and
     * onComplete methods, making them thread-safe.
     * <p>The method is thread-safe.
     * @return the wrapped and serialized subject
     */
    @NonNull
    public final Subject<T> toSerialized() {
```

```
        if (this instanceof SerializedSubject) {
            return this;
        }
        return new SerializedSubject<T>(this);
    }
```

这个版本的 EventBus 比较简单，并没有考虑背压的情况，因为在 RxJava 2.x 中，Subject 已经不再支持背压了。如果要增加背压的处理，可以使用 Processor，我们需要将 PublishSubject 改成 PublishProcessor，对应的 Observable 也需要改成 Flowable。在第 15 章里会详细介绍如何开发一个 RxBus。

（7）使用 BehaviorSubject 实现预加载。

预加载可以很好地提高程序的用户体验。每当用户处于弱网络时，打开一个 App 很可能会出现一片空白或者一直在 loading，此时用户一定很烦躁。而如果能够预先加载一些数据，例如上一次打开 App 时保存的数据，那么一定会有效提升 App 的用户体验。

下面是借助 BehaviorSubject 的特性来实现一个简单的预加载类 RxPreLoader。

```
import io.reactivex.disposables.Disposable;
import io.reactivex.functions.Consumer;
import io.reactivex.subjects.BehaviorSubject;

/**
 * Created by Tony Shen on 2017/6/2.
 */

public class RxPreLoader<T> {

    //能够缓存订阅之前的最新数据
    private  BehaviorSubject<T> mData;
    private Disposable disposable;

    public RxPreLoader(T defaultValue) {

        mData = BehaviorSubject.createDefault(defaultValue);
    }

    /**
     * 发送事件
```

```
     * @param object
     */
    public void publish(T object) {
        mData.onNext(object);
    }

    /**
     * 订阅事件
     * @param onNext
     * @return
     */
    public  Disposable subscribe(Consumer onNext) {
        disposable = mData.subscribe(onNext);
        return disposable;
    }

    /**
     * 取消订阅
     *
     */
    public void dispose() {
        if (disposable != null && !disposable.isDisposed()) {
            disposable.dispose();
            disposable = null;
        }
    }

    /**
     * 获取缓存数据的 Subject
     *
     * @return
     */
    public BehaviorSubject<T> getCacheDataSubject() {
        return mData;
    }

    /**
     * 直接获取最近的一个数据
     *
     * @return
     */
    public T getLastCacheData() {
```

```
        return mData.getValue();
    }
}
```

可以考虑在 App 基类的 Activity/Fragment 中也实现一个类似的 RxPreLoader。

2. Processor

Processor 和 Subject 的作用相同。Processor 是 RxJava 2.0 新增的功能，它是一个接口，继承自 Subscriber、Publisher，能够支持背压（Back Pressure）控制。这是 Processor 和 Subject 的最大区别。

其实，Publisher 和 Processor 都是 Reactive Streams 的接口，Reactive Streams 项目提供了一个非阻塞异步流处理的背压标准。RxJava 2.0 已经基于 Reactive Streams 库进行重写，包括 Single、Completable，都是基于 Reactive Streams 规范重写的，Flowable 实现了 Reactive Streams 的 Publisher 接口等。

Reactive Streams 的目标是管制流数据在跨越异步边界进行流数据交换，可以认为是将元素传递到另一个线程或线程池，同时确保在接收端不是被迫缓冲任意数量的数据。换句话说，背压是该模型的重要组成部分，通过设置协调线程之间的队列大小进行限制。当各异步模型之间采用同步通信时会削弱异步带来的好处，因此必须采取谨慎措施，强制完全无阻塞反应流能在系统的各个方面都做到异步实施。

Reactive Streams 规范的主要目标：

- ◎ 通过异步边界（Asynchronous Boundary）来解耦系统组件。解耦的先决条件，分离事件/数据流的发送方和接收方的资源使用。
- ◎ 为背压处理定义一种模型。流处理的理想范式是将数据从发布者推送到订阅者，这样发布者就可以快速发布数据，同时通过压力处理来确保速度更快的发布者不会对速度较慢的订阅者造成过载。背压处理通过使用流控制来确保操作的稳定性并能实现优雅降级，从而提供弹性能力。

Reactive Streams JVM 接口由以下四个接口组成。

- ◎ Publisher：消息发布者。
- ◎ Subscriber：消息订阅者。
- ◎ Subscription：一个订阅。

◎　Processor：Publisher + Subscriber 的结合体。

RxJava 2.0 中使用 Processor 来处理背压。同时，在新发布的 Java 9 中也已经引入 Reactive Streams 的思想，具体来说是构建在 java.util.concurrent.Flow 容器中，包含了四个接口类。

小结

RxJava 的 Subject 是一种特殊的存在，它的灵活性在使用时也会伴随着风险，若是没有用好则可能会错过事件。注意，Subject 不是线程安全的。当然很多开源框架都在使用 Subject，例如大名鼎鼎的 RxLifecycle 就使用了 BehaviorSubject。

2.6　小结

本章详细介绍了 RxJava 2.0 的 5 种观察者模式，以及 Subject 和 Processor。5 种观察者模式是 RxJava 2.0 的核心内容，正是有了它们，才构成了 RxJava 2.0。

本章还详细介绍了 Hot Observable 和 Cold Observable，以及它们的用途和相互转换。

在本章的内容中，讲得比较少的是 Flowable，它是 RxJava 2.0 新增的被观察者，支持背压。关于背压的内容会在第 8 章详细介绍。

第3章

创建操作符

从本书的第 3 章开始讲解 RxJava 的操作符，操作符是 RxJava 的重要组成部分。RxJava 的操作符大致可以按照图 3-1 来进行分类。

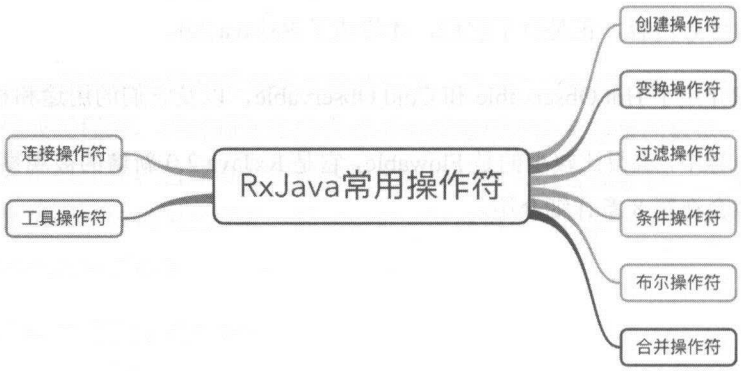


图 3-1

RxJava 的创建操作符主要包括如下内容。

- ◎ just()：将一个或多个对象转换成发射这个或这些对象的一个 Observable。
- ◎ from()：将一个 Iterable、一个 Future 或者一个数组转换成一个 Observable。
- ◎ create()：使用一个函数从头创建一个 Observable。
- ◎ defer()：只有当订阅者订阅才创建 Observable，为每个订阅创建一个新的 Observable。
- ◎ range()：创建一个发射指定范围的整数序列的 Observable。
- ◎ interval()：创建一个按照给定的时间间隔发射整数序列的 Observable。
- ◎ timer()：创建一个在给定的延时之后发射单个数据的 Observable。
- ◎ empty()：创建一个什么都不做直接通知完成的 Observable。

◎ error()：创建一个什么都不做直接通知错误的 Observable。

◎ never() — 创建一个不发射任何数据的 Observable。

3.1 create、just 和 from

1. create

使用一个函数从头开始创建一个 Observable，如图 3-2 所示。

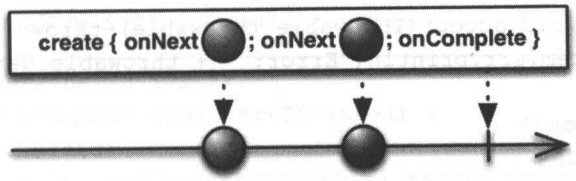


图 3-2

我们可以使用 create 操作符从头开始创建一个 Observable，给这个操作符传递一个接受观察者作为参数的函数，编写这个函数让它的行为表现为一个 Observable——恰当地调用观察者的 onNext、onError 和 onComplete 方法。一个形式正确的有限 Observable 必须尝试调用观察者的 onComplete()一次或者它的 onError()一次，而且此后不能再调用观察者的任何其他方法。

RxJava 建议我们在传递给 create 方法的函数时，先检查一下观察者的 isDisposed 状态，以便在没有观察者的时候，让我们的 Observable 停止发射数据，防止运行昂贵的运算。

示例代码：

```
    Observable.create(new ObservableOnSubscribe<Integer>() {

        @Override
        public void subscribe(ObservableEmitter<Integer> emitter) throws
Exception {
            try {
                if (!emitter.isDisposed()) {
                    for (int i = 0; i < 10; i++) {
                        emitter.onNext(i);
                    }
                    emitter.onComplete();
                }
```

```
            } catch (Exception e) {
                emitter.onError(e);
            }
        }
    }).subscribe(new Consumer<Integer>() {
        @Override
        public void accept(Integer integer) throws Exception {
            System.out.println("Next: " + integer);
        }
    }, new Consumer<Throwable>() {
        @Override
        public void accept(Throwable throwable) throws Exception {
            System.err.println("Error: " + throwable.getMessage());
        }
    }, new Action() {
        @Override
        public void run() throws Exception {
            System.out.println("Sequence complete.");
        }
    });
```

执行结果：

```
Next: 0
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Next: 6
Next: 7
Next: 8
Next: 9
Sequence complete.
```

2. just

创建一个发射指定值的 Observable，如图 3-3 所示。

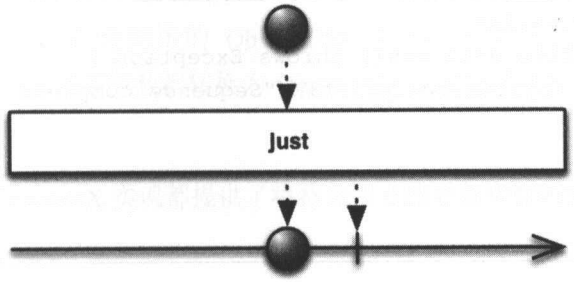


图 3-3

just 将单个数据转换为发射这个单个数据的 Observable。

```
Observable.just("hello just")
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                System.out.println(s);
            }
        });
```

执行结果：

```
hello just
```

just 类似于 from，但是 from 会将数组或 Iterable 的数据取出然后逐个发射，而 just 只是简单地原样发射，将数组或 Iterable 当作单个数据。

它可以接受一至十个参数，返回一个按参数列表顺序发射这些数据的 Observable。

```
Observable.just(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
```

```
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Next: 6
Next: 7
Next: 8
Next: 9
Next: 10
Sequence complete.
```

在 RxJava 2.0 中，如果在 just()中传入 null，则会抛出一个空指针异常。

```
java.lang.NullPointerException: The item is null
```

3. from

from 可以将其他种类的对象和数据类型转换为 Observable，如图 3-4 所示。

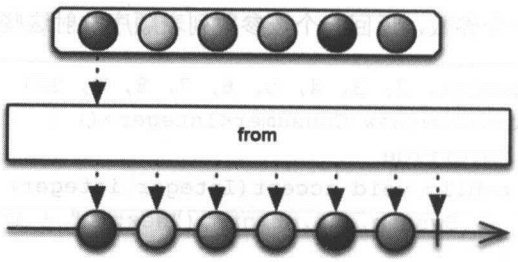


图 3-4

当我们使用 Observable 时，如果要处理的数据都可以转换成 Observables，而不是需要混合使用 Observables 和其他类型的数据，会非常方便。这让我们在数据流的整个生命周期中，可以使用一组统一的操作符来管理它们。

例如，Iterable 可以看成同步的 Observable；Future 可以看成总是只发射单个数据的 Observable。通过显式地将那些数据转换为 Observables，我们可以像使用 Observable 一样与它们交互。

因此，大部分 ReactiveX 实现都提供了将特定语言的对象和数据结构转换为 Observables 的方法。

在 RxJava 中，from 操作符可以将 Future、Iterable 和数组转换成 Obseruable。对于 Iterable 和数组，产生的 Observable 会发射 Iterable 或数组的每一项数据。

示例代码：

```
Observable.fromArray("hello","from")
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                System.out.println(s);
            }
        });
```

执行结果：

```
hello
from
```

示例代码：

```
List<Integer> items = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    items.add(i);
}

Observable.fromIterable(items)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
```

```
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 0
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Next: 6
Next: 7
Next: 8
Next: 9
Sequence complete.
```

对于 Future，它会发射 Future.get()方法返回的单个数据。

```
public static void main(String[] args) {

    ExecutorService executorService = Executors.newCachedThreadPool();
    Future<String> future = executorService.submit(new MyCallable());

    Observable.fromFuture(future)
            .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });

}

static class MyCallable implements Callable<String> {
    @Override
```

```
    public String call() throws Exception {
        System.out.println("模拟一些耗时的任务...");
        Thread.sleep(5000);
        return "OK";
    }
}
```

执行结果：

```
模拟一些耗时的任务...
OK
```

from 方法有一个可接受两个可选参数的版本，分别指定超时时长和时间单位。如果过了指定的时长，Future 还没有返回一个值，那么这个 Observable 就会发射错误通知并终止。 下面的代码，把超时时间设置为 4s。

```
public static void main(String[] args) {

    ExecutorService executorService = Executors.newCachedThreadPool();
    Future<String> future = executorService.submit(new MyCallable());

    Observable.fromFuture(future,4, TimeUnit.SECONDS)
            .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });

}

static class MyCallable implements Callable<String> {
    @Override
    public String call() throws Exception {
        System.out.println("模拟一些耗时的任务...");
        Thread.sleep(5000);
        return "OK";
    }
}
```

执行结果：

```
模拟一些耗时的任务...
io.reactivex.exceptions.OnErrorNotImplementedException
......
```

执行结果报错，说明超时的设置起了作用。

3.2 repeat

创建一个发射特定数据重复多次的 Observable，如图 3-5 所示。

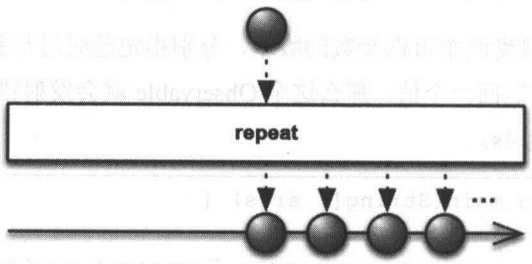


图 3-5

repeat 会重复地发射数据。某些实现允许我们重复发射某个数据序列，还有一些允许我们限制重复的次数。

repeat 不是创建一个 Observable，而是重复发射原始 Observable 的数据序列，这个序列或者是无限的，或者是通过 repeat(n)指定的重复次数。

示例代码：

```
Observable
        .just("hello repeat")
        .repeat(3)
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                System.out.println("s=" + s);
            }
        });
```

执行结果：

```
s=hello repeat
s=hello repeat
s=hello repeat
```

执行结果：

```
repeatWhen
```

在 RxJava 2.x 中还有两个 repeat 相关的操作符：repeatWhen 和 repeatUntil。

1. repeatWhen

repeatWhen 不是缓存和重放原始 Observable 的数据序列，而是有条件地重新订阅和发射原来的 Observable。

将原始 Observable 的终止通知（完成或错误）当作一个 void 数据传递给一个通知处理器，以此来决定是否要重新订阅和发射原来的 Observable。这个通知处理器就像一个 Observable 操作符，接受一个发射 void 通知的 Observable 作为输入，返回一个发射 void 数据（意思是，重新订阅和发射原始 Observable）或者直接终止（即使用 repeatWhen 终止发射数据）的 Observable。

示例代码：

```
        Observable.range(0, 9).repeatWhen(new Function<Observable<Object>,
ObservableSource<?>>() {
            @Override
            public ObservableSource<?> apply(Observable<Object>
objectObservable) throws Exception {
                return Observable.timer(10, TimeUnit.SECONDS);
            }
        }).subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println(integer);
            }
        });

        try {
            Thread.sleep(12000);
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
    }
```

执行结果：

```
0
1
2
3
4
5
6
7
8
0
1
2
3
4
5
6
7
8
```

在这里，我们会先发射 0 到 8 这 9 个数据，由于使用了 repeatWhen 操作符，因此在 10s 之后还会再发射一次这些数据。

2. repeatUntil

repeatUntil 是 RxJava 2.x 新增的操作符，表示直到某个条件就不再重复发射数据。当 BooleanSupplier 的 getAsBoolean()返回 false 时，表示重复发射上游的 Observable；当 getAsBoolean()为 true 时，表示中止重复发射上游的 Observable。

示例代码：

```
final long startTimeMillis = System.currentTimeMillis();
 Observable.interval(500, TimeUnit.MILLISECONDS)
        .take(5)
        .repeatUntil(new BooleanSupplier() {
            @Override
            public boolean getAsBoolean() throws Exception {
```

```
                return System.currentTimeMillis() - startTimeMillis >
5000;
            }
        })
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {
                System.out.println(aLong);
            }
        });

    try {
        Thread.sleep(6000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

执行结果：

```
0
1
2
3
4
0
1
2
3
4
```

可以发现执行的结果里打印了两遍 0 到 4，之所以不再打印第三遍是因为符合了 System.currentTimeMillis() - startTimeMillis > 5000 这个条件。

3.3　defer、interval 和 timer

1. defer

直到有观察者订阅时才创建 Observable，并且为每个观察者创建一个全新的 Observable，如图 3-6 所示。

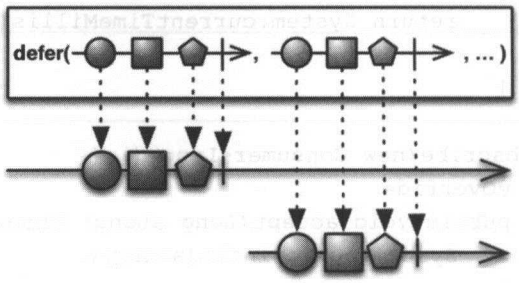


图 3-6

defer 操作符会一直等待直到有观察者订阅它，然后它使用 Observable 工厂方法生成一个 Observable。它对每个观察者都这样做，因此尽管每个订阅者都以为自己订阅的是同一个 Observable，但事实上每个订阅者获取的是它们自己单独的数据序列。

在某些情况下，直到最后一分钟（订阅发生时）才生成 Observable，以确保 Observable 包含最新的数据。

示例代码：

```
        Observable observable = Observable.defer(new Callable<ObservableSource<?
extends String>>() {
            @Override
            public ObservableSource<? extends String> call() throws Exception
{
                return Observable.just("hello defer");
            }
        });

        observable.subscribe(new Consumer<String>() {
            @Override
            public void accept(String str) throws Exception {
                System.out.println(str);
            }
        });
```

执行结果：

```
hello defer
```

2. interval

创建一个按固定时间间隔发射整数序列的 Observable，如图 3-7 所示。

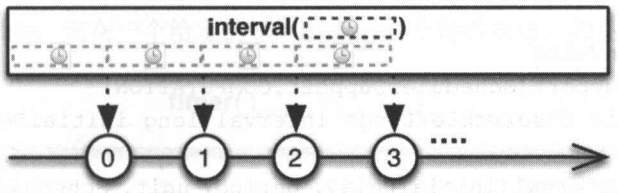


图 3-7

interval 操作符返回一个 Observable，它按固定的时间间隔发射一个无限递增的整数序列。

interval 接受一个表示时间间隔的参数和一个表示时间单位的参数。interval 默认在 computation 调度器上执行。

```
    /**
     * Returns an Observable that emits a {@code 0L} after the {@code initialDelay}
and ever increasing numbers
     * after each {@code period} of time thereafter.
     * <p>
     * <img width="640" height="200"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/timer
.p.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>{@code interval} operates by default on the {@code computation}
{@link Scheduler}.</dd>
     * </dl>
     *
     * @param initialDelay
     *            the initial delay time to wait before emitting the first value
of 0L
     * @param period
     *            the period of time between emissions of the subsequent numbers
     * @param unit
     *            the time unit for both {@code initialDelay} and {@code period}
     * @return an Observable that emits a 0L after the {@code initialDelay} and
ever increasing numbers after
     *         each {@code period} of time thereafter
```

```
     * @see <a
href="http://reactivex.io/documentation/operators/interval.html">ReactiveX
operators documentation: Interval</a>
     * @since 1.0.12
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.COMPUTATION)
    public static Observable<Long> interval(long initialDelay, long period,
TimeUnit unit) {
        return interval(initialDelay, period, unit, Schedulers.computation());
    }
```

示例代码：

```
Observable.interval(1, TimeUnit.SECONDS)
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {

                System.out.println(aLong);
            }
        });

try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

执行结果：

```
0
1
2
3
4
5
6
7
8
9
```

上述代码每隔 1 秒打印一个数字直到 10 秒后结束，最终打印了从 0 到 9 这十个数。

3. timer

创建一个 Observable，它在一个给定的延迟后发射一个特殊的值，如图 3-8 所示。

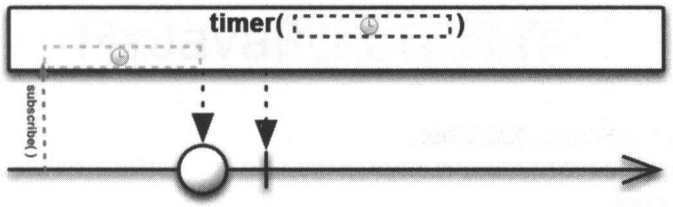


图 3-8

timer 操作符创建一个在给定的时间段之后返回一个特殊值的 Observable。

timer 返回一个 Observable，它在延迟一段给定的时间后发射一个简单的数字 0。timer 操作符默认在 computation 调度器上执行。

```
   /**
     * Returns an Observable that emits {@code 0L} after a specified delay, and
then completes.
     * <p>
     * <img width="640" height="200"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/timer
.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>{@code timer} operates by default on the {@code computation} {@link
Scheduler}.</dd>
     * </dl>
     *
     * @param delay
     *            the initial delay before emitting a single {@code 0L}
     * @param unit
     *            time units to use for {@code delay}
     * @return an Observable that {@code 0L} after a specified delay, and then
completes
     * @see <a
href="http://reactivex.io/documentation/operators/timer.html">ReactiveX
operators documentation: Timer</a>
```

```
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.COMPUTATION)
    public static Observable<Long> timer(long delay, TimeUnit unit) {
        return timer(delay, unit, Schedulers.computation());
    }
```

示例代码：

```
Observable
        .timer(2,TimeUnit.SECONDS)
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {

                System.out.println("hello timer"); // 2 秒后打印 hello timer
            }
        });

try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

执行结果：

```
hello timer
```

上述代码执行之后，会延迟 2s 打印“hello timer”。

3.4 小结

本章介绍了多个创建操作符，它们用于创建被观察者，所以创建操作符是 RxJava 比较基础的操作符。

本章介绍的创建操作符不仅 Observable 可以使用，Flowable 等也都可以使用。

第4章
RxJava的线程操作

4.1 调度器（Scheduler）种类

1. RxJava 线程介绍

RxJava 是一个为异步编程而实现的库，异步是其重要特色，合理地利用异步编程能够提高系统的处理速度。但是异步也会带来线程的安全问题，而且异步并不等于并发，与异步概念相对应的是同步。

在默认情况下，RxJava 只在当前线程中运行，它是单线程的。此时 Observable 用于发射数据流，Observer 用于接收和响应数据流，各种操作符（Operators）用于加工数据流，它们都在同一个线程中运行，实现出来的是一个同步的函数响应式。然而，函数响应式的实际应用是大部分操作都在后台处理，前台响应的一个过程。所以需要对刚才的流程做一下修改，改成 Observable 生成发射数据流，Operators 加工数据流在后台线程中进行，Observer 在前台线程中接收并响应数据。此时会涉及使用多线程来操作 RxJava，我们可以使用 RxJava 的调度器（Scheduler）来实现。

2. Scheduler

Scheduler 是 RxJava 对线程控制器的一个抽象，RxJava 内置了多个 Scheduler 的实现，它们基本满足绝大多数使用场景，如表 4-1 所示。

表 4-1

Scheduler	作　用
single	使用定长为 1 的线程池（new Scheduled Thread Pool(1)），重复利用这个线程
newThread	每次都启用新线程，并在新线程中执行操作
computation	使用的固定的线程池（Fixed Scheduler Pool），大小为 CPU 核数，适用于 CPU 密集型计算
io	适合 I/O 操作（读写文件、读写数据库、网络信息交互等）所使用的 Scheduler。行为模式和 newThread() 差不多，区别在于 io() 的内部实现是用一个无数量上限的线程池，可以重用空闲的线程，因此多数情况下，io() 比 newThread() 更有效率
trampoline	直接在当前线程运行，如果当前线程有其他任务正在执行，则会先暂停其他任务
Schedulers.from	将 java.util.concurrent.Executor 转换成一个调度器实例，即可以自定义一个 Executor 来作为调度器

如果内置的 Scheduler 不能满足业务需求，那么可以使用自定义的 Executor 作为调度器，以满足个性化需求。

下面来看一个简单的例子，看看如何使用 Scheduler。

```
Observable.create(new ObservableOnSubscribe<String>() {

        @Override
        public void subscribe(ObservableEmitter<String> e) throws Exception
{
            e.onNext("hello");
            e.onNext("world");
        }
    }).observeOn(Schedulers.newThread())
    .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });
```

这里的 Observable 发射完数据之后，切换到 newThread。后面的两次打印都是在 newThread 中进行的。通过这个例子可以看到，与传统的开发方式相比，RxJava 切换线程要简单方便得多。接下来会详细介绍 RxJava 的线程模型，包括各个 Scheduler 原理、线程调度等。

4.2　RxJava 线程模型

RxJava 的被观察者们在使用操作符时可以利用线程调度器——Scheduler 来切换线程，例如

```
Observable.just("aaa","bbb")
        .observeOn(Schedulers.newThread())
        .map(new Function<String, String>() {
            @Override
            public String apply(@NonNull String s) throws Exception {

                return s.toUpperCase();
            }
        })
        .subscribeOn(Schedulers.single())
        .observeOn(Schedulers.io())
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {

                System.out.println(s);
            }
        });
```

不同的箭头颜色表示不同的线程，如图 4-1 所示。

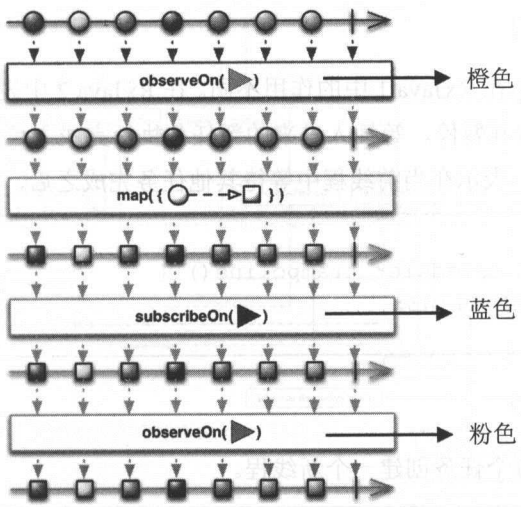


图 4-1

其中，蓝色表示主线程、橙色表示 newThread、粉色表示 I/O 线程。

1. 线程调度器

Schedulers 是一个静态工厂类，通过分析 Schedulers 的源码可以看到它有多种不同类型的 Scheduler。下面是 Schedulers 的各个工厂方法。

◎ computation()

computation()用于 CPU 密集型的计算任务，但并不适合 I/O 操作。

```
@NonNull
 public static Scheduler computation() {
     return RxJavaPlugins.onComputationScheduler(COMPUTATION);
 }
```

◎ io()

io()用于 I/O 密集型任务，支持异步阻塞 I/O 操作，这个调度器的线程池会根据需要增长。对于普通的计算任务，请使用 Schedulers.computation()。

```
@NonNull
 public static Scheduler io() {
     return RxJavaPlugins.onIoScheduler(IO);
 }
```

◎ trampoline()

在 RxJava 2 中与在 RxJava 1 中的作用不同。在 RxJava 2 中表示立即执行，如果当前线程有任务在执行，则会将其暂停，等插入进来的新任务执行完成之后，再接着执行原先未完成的任务。在 RxJava 1 中，表示在当前线程中等待其他任务完成之后，再执行新的任务。

```
@NonNull
 public static Scheduler trampoline() {
     return TRAMPOLINE;
 }
```

◎ newThread()

newThread()为每个任务创建一个新线程。

```
@NonNull
```

```
public static Scheduler newThread() {
    return RxJavaPlugins.onNewThreadScheduler(NEW_THREAD);
}
```

◎ single()

single()拥有一个线程单例，所有的任务都在这一个线程中执行。当此线程中有任务执行时，它的任务将会按照先进先出的顺序依次执行。

```
@NonNull
public static Scheduler single() {
    return RxJavaPlugins.onSingleScheduler(SINGLE);
}
```

除此之外，还支持自定义的 Executor 来作为调度器。

```
@NonNull
public static Scheduler from(@NonNull Executor executor) {
    return new ExecutorScheduler(executor);
}
```

如图 4-2 所示，Scheduler 是 RxJava 的线程任务调度器，Worker 是线程任务的具体执行者。从 Scheduler 源码可以看到，Scheduler 在 scheduleDirect()、schedulePeriodicallyDirect()方法中创建了 Worker，然后会分别调用 worker 的 schedule()、schedulePeriodically()来执行任务。

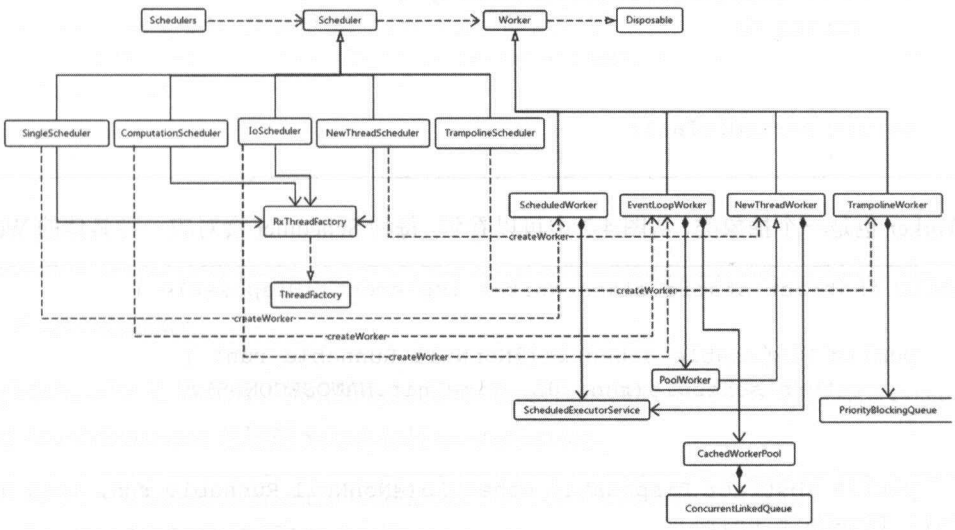


图 4-2

```
    public Disposable scheduleDirect(@NonNull Runnable run, long delay, @NonNull
TimeUnit unit) {
        final Worker w = createWorker();

        final Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

        DisposeTask task = new DisposeTask(decoratedRun, w);

        w.schedule(task, delay, unit);

        return task;
    }

    public Disposable schedulePeriodicallyDirect(@NonNull Runnable run, long
initialDelay, long period, @NonNull TimeUnit unit) {
        final Worker w = createWorker();

        final Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

        PeriodicDirectTask periodicTask = new PeriodicDirectTask(decoratedRun,
w);

        Disposable d = w.schedulePeriodically(periodicTask, initialDelay,
period, unit);
        if (d == EmptyDisposable.INSTANCE) {
            return d;
        }

        return periodicTask;
    }
```

Worker 也是一个抽象类，从图 4-2 中可以看到，每种 Scheduler 会对应一种具体的 Worker。

```
    public abstract static class Worker implements Disposable {

        public Disposable schedule(@NonNull Runnable run) {
            return schedule(run, 0L, TimeUnit.NANOSECONDS);
        }

        public abstract Disposable schedule(@NonNull Runnable run, long delay,
@NonNull TimeUnit unit);
```

```
        public Disposable schedulePeriodically(@NonNull Runnable run, final long
initialDelay, final long period, @NonNull final TimeUnit unit) {
            final SequentialDisposable first = new SequentialDisposable();

            final SequentialDisposable sd = new SequentialDisposable(first);

            final Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

            final long periodInNanoseconds = unit.toNanos(period);
            final long firstNowNanoseconds = now(TimeUnit.NANOSECONDS);
            final long firstStartInNanoseconds = firstNowNanoseconds +
unit.toNanos(initialDelay);

            Disposable d = schedule(new PeriodicTask(firstStartInNanoseconds,
decoratedRun, firstNowNanoseconds, sd,
                    periodInNanoseconds), initialDelay, unit);

            if (d == EmptyDisposable.INSTANCE) {
                return d;
            }
            first.replace(d);

            return sd;
        }

        public long now(@NonNull TimeUnit unit) {
            return unit.convert(System.currentTimeMillis(),
TimeUnit.MILLISECONDS);
        }

        ...

    }
```

（1）SingleScheduler

SingleScheduler 是 RxJava 2 新增的 Scheduler。SingleScheduler 中有一个属性叫作 executor，它是使用 AtomicReference 包装的 ScheduledExecutorService。

```
final AtomicReference<ScheduledExecutorService> executor = new
AtomicReference<ScheduledExecutorService>();
```

在 SingleScheduler 构造函数中，executor 会调用 lazySet()。

```
public SingleScheduler(ThreadFactory threadFactory) {
    this.threadFactory = threadFactory;
    executor.lazySet(createExecutor(threadFactory));
}
```

它的 createExecutor()用于创建工作线程，可以看到通过 SchedulerPoolFactory 来创建 ScheduledExecutorService。

```
static ScheduledExecutorService createExecutor(ThreadFactory threadFactory)
{
    return SchedulerPoolFactory.create(threadFactory);
}
```

在 SchedulerPoolFactory 类的 create(ThreadFactory factory) 中，使用 newScheduledThreadPool 线程池定义定时器，最大允许线程数为 1。

```
public static ScheduledExecutorService create(ThreadFactory factory) {
    final ScheduledExecutorService exec =
Executors.newScheduledThreadPool(1, factory);
    if (exec instanceof ScheduledThreadPoolExecutor) {
        ScheduledThreadPoolExecutor e = (ScheduledThreadPoolExecutor) exec;
        POOLS.put(e, exec);
    }
    return exec;
}
```

在 SingleScheduler 中，每次使用 ScheduledExecutorService 时，其实是使用 executor.get()。所以说，single 拥有一个线程单例。

SingleScheduler 会创建一个 ScheduledWorker，ScheduledWorker 使用 JDK 的 ScheduledExecutorService 作为 executor。

下面是 ScheduledWorker 的 schedule()方法，使用 ScheduledExecutorService 的 submit()或 schedule()来执行 runnable。

```
    @NonNull
    @Override
```

```
        public Disposable schedule(@NonNull Runnable run, long delay, @NonNull
TimeUnit unit) {
            if (disposed) {
                return EmptyDisposable.INSTANCE;
            }

            Runnable decoratedRun = RxJavaPlugins.onSchedule(run);

            ScheduledRunnable sr = new ScheduledRunnable(decoratedRun, tasks);
            tasks.add(sr);

            try {
                Future<?> f;
                if (delay <= 0L) {
                    f = executor.submit((Callable<Object>)sr);
                } else {
                    f = executor.schedule((Callable<Object>)sr, delay, unit);
                }

                sr.setFuture(f);
            } catch (RejectedExecutionException ex) {
                dispose();
                RxJavaPlugins.onError(ex);
                return EmptyDisposable.INSTANCE;
            }

            return sr;
        }
```

（2）ComputationScheduler

ComputationScheduler 使用 FixedSchedulerPool 作为线程池，并且 FixedSchedulerPool 被 AtomicReference 包装了一下。

从 ComputationScheduler 的源码中可以看出，MAX_THREADS 是 CPU 的数目。FixedSchedulerPool 可以理解为拥有固定数量的线程池，数量为 MAX_THREADS。

```
static {
    MAX_THREADS = cap(Runtime.getRuntime().availableProcessors(),
Integer.getInteger(KEY_MAX_THREADS, 0));
    ......
```

```
}

static int cap(int cpuCount, int paramThreads) {
    return paramThreads <= 0 || paramThreads > cpuCount ? cpuCount :
paramThreads;
}
```

ComputationScheduler 会创建一个 EventLoopWorker。

```
@NonNull
@Override
public Worker createWorker() {
    return new EventLoopWorker(pool.get().getEventLoop());
}
```

其中，getEventLoop()是 FixedSchedulerPool 中的方法，返回了 FixedSchedulerPool 中的一个 PoolWorker。

```
public PoolWorker getEventLoop() {
    int c = cores;
    if (c == 0) {
        return SHUTDOWN_WORKER;
    }
    // simple round robin, improvements to come
    return eventLoops[(int)(n++ % c)];
}
```

PoolWorker 继承自 NewThreadWorker，也是线程数为 1 的 ScheduledExecutorService。

（3）IoScheduler

IoScheduler 使用 CachedWorkerPool 作为线程池，并且 CachedWorkerPool 也被 AtomicReference 包装了一下。

CachedWorkerPool 是基于 RxThreadFactory 这个 ThreadFactory 来创建的。

```
static {
    ......
    WORKER_THREAD_FACTORY = new RxThreadFactory(WORKER_THREAD_NAME_PREFIX,
priority);
    ......
```

```
        NONE = new CachedWorkerPool(0, null, WORKER_THREAD_FACTORY);
        ......
}
```

在 RxThreadFactory 中，由 prefix 和 incrementAndGet() 来创建新线程的名称。

```
    @Override
    public Thread newThread(Runnable r) {
        StringBuilder nameBuilder = new
StringBuilder(prefix).append('-').append(incrementAndGet());

        String name = nameBuilder.toString();
        Thread t = nonBlocking ? new RxCustomThread(r, name) : new Thread(r, name);
        t.setPriority(priority);
        t.setDaemon(true);
        return t;
    }
```

IoScheduler 创建的线程数是不固定的，可以通过 IoScheduler 的 size() 来获得当前的线程数。一般情况下，ComputationScheduler 的线程数等于 CPU 的数目。

```
    public int size() {
        return pool.get().allWorkers.size();
    }
```

需要特别注意的是，ComputationScheduler 和 IoScheduler 都是依赖线程池来维护线程的，区别就是 IoScheduler 线程池中的个数是无限的，由 prefix 和 incrementAndGet()产生的递增值来决定线程的名字。而 ComputationScheduler 中则是一个固定线程数量的线程池，数据为 CPU 的数目，并且不要把 I/O 操作放在 computation()中，否则 I/O 操作的等待时间会浪费 CPU。

同样，IoScheduler 也会创建 EventLoopWorker。

```
    @NonNull
    @Override
    public Worker createWorker() {
        return new EventLoopWorker(pool.get());
    }
```

但这个 EventLoopWorker 是 IoScheduler 的内部类，与 ComputationScheduler 创建的 EventLoopWorker 不同，只是二者的名称相同罢了。

（4）NewThreadScheduler

NewThreadScheduler 会创建 NewThreadWorker，NewThreadWorker 的构造函数使用的也是 SchedulerPoolFactory。

```
public NewThreadWorker(ThreadFactory threadFactory) {
    executor = SchedulerPoolFactory.create(threadFactory);
}
```

与 SingleScheduler 不同的是，SingleScheduler 的 executor 是使用 AtomicReference 包装的 ScheduledExecutorService。每次使用时，都会调用 executor.get()。

然而，NewThreadScheduler 每次都会创建一个新的线程。

（5）TrampolineScheduler

TrampolineScheduler 会创建 TrampolineWorker，在 TrampolineWorker 内部维护着一个 PriorityBlockingQueue。任务进入该队列之前，会先用 TimedRunnable 封装一下。

```
static final class TimedRunnable implements Comparable<TimedRunnable> {
    final Runnable run;
    final long execTime;
    final int count;

    volatile boolean disposed;

    TimedRunnable(Runnable run, Long execTime, int count) {
        this.run = run;
        this.execTime = execTime;
        this.count = count;
    }

    @Override
    public int compareTo(TimedRunnable that) {
        int result = ObjectHelper.compare(execTime, that.execTime);
        if (result == 0) {
            return ObjectHelper.compare(count, that.count);
        }
        return result;
    }
}
```

可以看到 TimedRunnable 实现了 Comparable 接口，会比较任务的 execTime 和 count。

任务在进入 queue 之前，count 每次都会+1。

```
final TimedRunnable timedRunnable = new TimedRunnable(action, execTime,
counter.incrementAndGet());
queue.add(timedRunnable);
```

所以，在使用 TrampolineScheduler 时，新的任务总是会优先执行。

2. 线程调度

默认情况下不做任何线程处理，Observable 和 Observer 处于同一线程中。如果想要切换线程，则可以使用 subscribeOn()和 observeOn()。

（1）subscribeOn

subscribeOn 通过接收一个 Scheduler 参数，来指定对数据的处理运行在特定的线程调度器 Scheduler 上。

若多次执行 subscribeOn，则只有一次起作用。

单击 subscribeOn() 的源码可以看到，每次调用 subscribeOn() 都会创建一个 ObservableSubscribeOn 对象。

```
public final Observable<T> subscribeOn(Scheduler scheduler) {
    ObjectHelper.requireNonNull(scheduler, "scheduler is null");
    return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this,
scheduler));
}
```

ObservableSubscribeOn 真正发生订阅的方法是 subscribeActual(Observer<? super T> observer)。

```
@Override
public void subscribeActual(final Observer<? super T> s) {
    final SubscribeOnObserver<T> parent = new SubscribeOnObserver<T>(s);

    s.onSubscribe(parent);
```

```
        parent.setDisposable(scheduler.scheduleDirect(new
SubscribeTask(parent)));
    }
```

其中，SubscribeOnObserver 是下游的 Observer 通过装饰器模式生成的，它实现了 Observer、Disposable 接口。

接下来，在上游的线程中执行下游 Observer 的 onSubscribe(Disposable disposabel)方法。

```
s.onSubscribe(parent);
```

然后，将子线程的操作加入 Disposable 管理中，加入 Disposable 后可以方便上下游的统一管理。

```
parent.setDisposable(scheduler.scheduleDirect(new SubscribeTask(parent)));
```

在这里，已经调用了对应 scheduler 的 scheduleDirect()方法。scheduleDirect() 传入的是一个 Runnable，也就是下面的 SubscribeTask。

```
final class SubscribeTask implements Runnable {
    private final SubscribeOnObserver<T> parent;

    SubscribeTask(SubscribeOnObserver<T> parent) {
        this.parent = parent;
    }

    @Override
    public void run() {
        source.subscribe(parent);
    }
}
```

SubscribeTask 会执行 run()对上游的 Observable，从而进行订阅。

此时，已经在对应的 Scheduler 线程中运行了：

```
source.subscribe(parent);
```

在 RxJava 的链式操作中，数据的处理是自下而上的，这点与数据发射正好相反。如果多次调用 subscribeOn，则最上面的线程切换最晚执行，所以就变成了只有第一次切换线程才有效。

（2）observeOn

observeOn 同样接收一个 Scheduler 参数，用来指定下游操作运行在特定的线程调度器 Scheduler 上。

若多次执行 observeOn，则每次都起作用，线程会一直切换。

单击 observeOn()的源码可以看到，每次调用 observeOn()都会创建一个 ObservableObserveOn 对象。

```
public final Observable<T> observeOn(Scheduler scheduler) {
    return observeOn(scheduler, false, bufferSize());
}

public final Observable<T> observeOn(Scheduler scheduler, boolean
delayError, int bufferSize) {
    ObjectHelper.requireNonNull(scheduler, "scheduler is null");
    ObjectHelper.verifyPositive(bufferSize, "bufferSize");
    return RxJavaPlugins.onAssembly(new ObservableObserveOn<T>(this,
scheduler, delayError, bufferSize));
}
```

ObservableObserveOn 真正发生订阅的方法是 subscribeActual(Observer<? super T> observer)。

```
@Override
protected void subscribeActual(Observer<? super T> observer) {
    if (scheduler instanceof TrampolineScheduler) {
        source.subscribe(observer);
    } else {
        Scheduler.Worker w = scheduler.createWorker();

        source.subscribe(new ObserveOnObserver<T>(observer, w, delayError,
bufferSize));
    }
}
```

如果 scheduler 是 TrampolineScheduler，则上游事件和下游事件会立即产生订阅。

如果不是 TrampolineScheduler，则 scheduler 会创建自己的 Worker，然后上游事件和下游事件产生订阅，生成一个 ObserveOnObserver 对象，封装了下游真正的 Observer。

ObserveOnObserver 是 ObservableObserveOn 的内部类，实现了 Observer、Runnable 接口。与 SubscribeOnObserver 不同的是，SubscribeOnObserver 实现了 Observer、Disposable 接口。

在 ObserveOnObserver 的 onNext()中，schedule()执行了具体调度的方法。

```
@Override
public void onNext(T t) {
    if (done) {
        return;
    }

    if (sourceMode != QueueDisposable.ASYNC) {
        queue.offer(t);
    }
    schedule();
}

void schedule() {
    if (getAndIncrement() == 0) {
        worker.schedule(this);
    }
}
```

其中，worker 是当前 scheduler 创建的 Worker，this 指的是当前的 ObserveOnObserver 对象，this 实现了 Runnable 接口。

然后，再来看看 Runnable 接口的实现方法 run()，这个方法是在 Worker 对应的线程里执行的。drainNormal()会取出 ObserveOnObserver 的 queue 里的数据进行发送。

```
@Override
public void run() {
    if (outputFused) {
        drainFused();
    } else {
        drainNormal();
    }
}
```

若下游多次调用 observeOn()，则线程会一直切换。每次切换线程，都会把对应的 Observer 对象的各个方法的处理执行在指定的线程中。

3. 示例

（1）单独使用 subscribeOn

对前面的例子稍微做一下修改，将原先的 observeOn 改成 subscribeOn

```
    Observable.create(new ObservableOnSubscribe<String>() {

        @Override
        public void subscribe(ObservableEmitter<String> e) throws Exception
{
            e.onNext("hello");
            e.onNext("world");
        }
    }).subscribeOn(Schedulers.newThread())
    .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });
```

此时，所有的操作都是在 newThread 中运行的，包括发射数据。

（2）多次切换线程

最后，举一个多次调用 subscribeOn 和 observeOn 的例子。

```
    Observable.just("HELLO WORLD")
            .subscribeOn(Schedulers.single())
            .map(new Function<String, String>() {
                @Override
                public String apply(@NonNull String s) throws Exception {

                    s = s.toLowerCase();
                    L.i("map1",s);
                    return s;
                }
            })
            .observeOn(Schedulers.io())
            .map(new Function<String, String>() {
```

```
        @Override
        public String apply(String s) throws Exception {

            s = s+" tony.";
            L.i("map2",s);
            return s;
        }
    })
    .subscribeOn(Schedulers.computation())
    .map(new Function<String, String>() {

        @Override
        public String apply(String s) throws Exception {

            s = s+"it is a test.";
            L.i("map3",s);
            return s;
        }
    })
    .observeOn(Schedulers.newThread())
    .subscribe(new Consumer<String>() {
        @Override
        public void accept(@NonNull String s) throws Exception {

            L.i("subscribe",s);
            System.out.println(s);
        }
    });
```

从图 4-3 中可以看出这里所做的线程切换。

这个例子是在 Android 中运行的，在这里用到了笔者个人编写的 Android 日志框架，GitHub 地址是 https://github.com/fengzhizi715/SAF-Kotlin-log。它是完全基于 Kotlin 开发的框架，提供极简的 API。打印出来的日志风格是第一行显示线程名，第二行显示类中打印的行数，第三行显示打印的具体内容。除此之外，它默认支持 JSON 字符串、集合、Map、Bundle、Intent、Reference、Throwable、URI 等类型的打印，并分别做了特别的格式化处理。

图 4-3

了解 RxJava 的线程模型、线程调度器和线程调度是非常有意义的，它们能够帮助我们更合理地使用 RxJava。另外，RxJava 的线程切换结合链式调用非常方便，比起传统的 Java 线程操作，实在是简单太多了。

4.3　Scheduler 的测试

TestScheduler 是专门用于测试的调度器，与其他调度器的区别是，TestScheduler 只有被调用了时间才会继续。TestScheduler 是一种特殊的、非线程安全的调度器，用于测试一些不引入真实并发性、允许手动推进虚拟时间的调度器。

在 RxJava 2.x 中，原先 RxJava 1.x 的 Schedulers.test()被去掉了。要想获得 TestScheduler 对象，则可以通过直接 new TestScheduler()的方式来实现。

TestScheduler 所包含的方法并不多，下面罗列几个关键的方法。

（1）advanceTimeTo

将调度器的时钟移动到某个特定时刻。

例如，时钟移动到 10ms。

```
scheduler.advanceTimeTo(10, TimeUnit.MILLISECONDS);
```

时钟移动到 20ms。

```
scheduler.advanceTimeBy(20, TimeUnit.MILLISECONDS);
```

下面的例子展示了 0s、20s、40s 各会打印什么结果。

```
        TestScheduler scheduler = new TestScheduler();

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("immediate");
            }
        });

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("20s");
            }
        },20, TimeUnit.SECONDS);

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("40s");
            }
        },40, TimeUnit.SECONDS);

        scheduler.advanceTimeTo(1, TimeUnit.MILLISECONDS);
        System.out.println("virtual time: " +
scheduler.now(TimeUnit.MILLISECONDS));

        scheduler.advanceTimeTo(20, TimeUnit.SECONDS);
        System.out.println("virtual time: " +
scheduler.now(TimeUnit.SECONDS));

        scheduler.advanceTimeTo(40, TimeUnit.SECONDS);
        System.out.println("virtual time: " +
scheduler.now(TimeUnit.SECONDS));
```

执行结果：

```
immediate
virtual time: 1
20s
virtual time: 20
40s
virtual time: 40
```

可以看到使用 advanceTimeTo 之后，移动不同的时间点会打印不同的内容。

当然，advanceTimeTo()也可以传负数，表示回到过去的时间点，但是一般不推荐这种用法。

（2）advanceTimeBy

将调度程序的时钟按指定的时间向前移动。

例如，时钟移动了 10ms。

```
scheduler.advanceTimeBy(10, TimeUnit.MILLISECONDS);
```

再次调用刚才的方法，时钟又会移动 10ms。此时，时钟移动到 20ms，这是一个累加的过程。

```
scheduler.advanceTimeBy(10, TimeUnit.MILLISECONDS);
```

下面的例子使用了 timer 操作符，timer 是按照指定时间延迟发送的操作符，timer()并不会按周期地执行。该例子展示了 2s 后 atomicLong 会自动加 1。

```
        TestScheduler scheduler = new TestScheduler();

        final AtomicLong atomicLong = new AtomicLong();
        Observable.timer(2, TimeUnit.SECONDS, scheduler).subscribe(new
Consumer<Long>() {
            @Override
            public void accept(final Long value) throws Exception {
                atomicLong.incrementAndGet();
            }
        });
```

```
        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));

        scheduler.advanceTimeBy(1, TimeUnit.SECONDS);

        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));

        scheduler.advanceTimeBy(1, TimeUnit.SECONDS);

        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));
```

执行结果：

```
atomicLong's value=0, virtual time:0
atomicLong's value=0, virtual time:1
atomicLong's value=1, virtual time:2
```

这个结果符合预期，最初 atomicLong 为 0，时钟移动到 1s 时它的值仍然为 0；时钟再移动 1s，即相当于时钟移动到 2s，所以它的值变为 1。

当然，advanceTimeBy()也可以传负数，表示回到过去。

```
        TestScheduler scheduler = new TestScheduler();

        final AtomicLong atomicLong = new AtomicLong();
        Observable.timer(2, TimeUnit.SECONDS, scheduler).subscribe(new
Consumer<Long>() {
            @Override
            public void accept(final Long value) throws Exception {
                atomicLong.incrementAndGet();
            }
        });

        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));

        scheduler.advanceTimeBy(1, TimeUnit.SECONDS);

        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));
```

```
        scheduler.advanceTimeBy(-1, TimeUnit.SECONDS);

        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));

        scheduler.advanceTimeBy(2, TimeUnit.SECONDS);

        System.out.println("atomicLong's value="+atomicLong.get() + ", virtual
time:" + scheduler.now(TimeUnit.SECONDS));
```

执行结果：

```
atomicLong's value=0, virtual time:0
atomicLong's value=0, virtual time:1
atomicLong's value=0, virtual time:0
atomicLong's value=1, virtual time:2
```

（3）triggerActions

triggerActions 不会修改时间，它执行计划中的但是未启动的任务，已经执行过的任务不会再启动。

```
      TestScheduler scheduler = new TestScheduler();

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("immediate");
            }
        });

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("20s");
            }
        },20, TimeUnit.SECONDS);

        System.out.println("virtual time: " +
scheduler.now(TimeUnit.SECONDS));
```

执行结果：

```
virtual time: 0
```

稍微改一下代码，增加 scheduler.triggerActions()：

```
        TestScheduler scheduler = new TestScheduler();

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("immediate");
            }
        });

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("20s");
            }
        },20, TimeUnit.SECONDS);

        scheduler.triggerActions();
        System.out.println("virtual time: " +
scheduler.now(TimeUnit.SECONDS));
```

执行结果：

```
immediate
virtual time: 0
```

此时由于执行了 triggerActions()，所以打印了 immediate。

再改一下，增加 advanceTimeBy()。

```
        TestScheduler scheduler = new TestScheduler();

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("immediate");
            }
```

```
        });

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("20s");
            }
        },20, TimeUnit.SECONDS);

        scheduler.triggerActions();
        System.out.println("virtual time: " +
scheduler.now(TimeUnit.SECONDS));

        scheduler.advanceTimeBy(20,TimeUnit.SECONDS);
```

执行结果：

```
immediate
virtual time: 0
20s
```

如果将 triggerActions()放在最后，看看效果。

```
    TestScheduler scheduler = new TestScheduler();

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("immediate");
            }
        });

        scheduler.createWorker().schedule(new Runnable() {
            @Override
            public void run() {
                System.out.println("20s");
            }
        },20, TimeUnit.SECONDS);


        System.out.println("virtual time: " +
scheduler.now(TimeUnit.SECONDS));
```

```
        scheduler.advanceTimeBy(20,TimeUnit.SECONDS);
        scheduler.triggerActions();
```

执行结果：

```
virtual time: 0
immediate
20s
```

因为已经使用了 advanceTimeBy()，所以即使再调用 triggerActions()，也不会执行已经启动过的任务。

对于测试一些需要精确时间的任务，TestScheduler 还是很有用处的，可以节省很多等待的时间。

4.4 小结

本章介绍了 RxJava 如何使用多线程、RxJava 的线程模型以及 RxJava 如何实现并发操作。

在 RxJava 中有多个内置的调度器（Scheduler），每个都适用于不同的场景。如果想在 Android 中使用 RxJava，则可以结合 RxAndroid 的调度器，这些内容会在第 11 章里讲述，感兴趣的读者也可以先翻阅第 11 章的内容，加深对 Scheduler 的印象。

另外，在本书的第 10 章会讲解 RxJava 的并行操作。

第 5 章
变换操作符和过滤操作符

RxJava 的变换操作符主要包括以下几种。

- map( )：对序列的每一项都用一个函数来变换 Observable 发射的数据序列。
- flatMap( )、concatMap( )和 flatMapIterable( )：将 Observable 发射的数据集合变换为 Observables 集合，然后将这些 Observable 发射的数据平坦化地放进一个单独的 Observable 中。
- switchMap( )：将 Observable 发射的数据集合变换为 Observables 集合，然后只发射这些 Observables 最近发射过的数据。
- scan( )：对 Observable 发射的每一项数据应用一个函数，然后按顺序依次发射每一个值。
- groupBy( )：将 Observable 拆分为 Observable 集合，将原始 Observable 发射的数据按 Key 分组，每一个 Observable 发射过一组不同的数据。
- buffer( )：定期从 Observable 收集数据到一个集合，然后把这些数据集合打包发射，而不是一次发射一个。
- window( )：定期将来自 Observable 的数据拆分成一些 Observable 窗口，然后发射这些窗口，而不是每次发射一项。
- cast( )：在发射之前强制将 Observable 发射的所有数据转换为指定类型。

RxJava 的过滤操作符主要包括以下几种。

- filter( )：过滤数据。
- takeLast( )：只发射最后的 *N* 项数据。
- last( )：只发射最后一项数据。

- lastOrDefault( )：只发射最后一项数据，如果 Observable 为空，就发射默认值。
- takeLastBuffer( )：将最后的 *N* 项数据当作单个数据发射。
- skip( )：跳过开始的 *N* 项数据。
- skipLast( )：跳过最后的 *N* 项数据。
- take( )：只发射开始的 *N* 项数据。
- first( ) and takeFirst( )：只发射第一项数据，或者满足某种条件的第一项数据。
- firstOrDefault( )：只发射第一项数据，如果 Observable 为空，就发射默认值。
- elementAt( )：发射第 *N* 项数据。
- elementAtOrDefault( )：发射第 *N* 项数据，如果 Observable 数据少于 *N* 项，就发射默认值。
- sample( ) or throttleLast( )：定期发射 Observable 最近的数据。
- throttleFirst( )：定期发射 Observable 发射的第一项数据。
- throttleWithTimeout( ) or debounce( )：只有当 Observable 在指定的时间段后还没有发射数据时，才发射一个数据。
- timeout( )：如果在一个指定的时间段后还没发射数据，就发射一个异常。
- distinct( )：过滤掉重复的数据。
- distinctUntilChanged( )：过滤掉连续重复的数据。
- ofType( )：只发射指定类型的数据。
- ignoreElements( )：丢弃所有的正常数据，只发射错误或完成通知。

5.1 map 和 flatMap

1. map 操作符

对 Observable 发射的每一项数据应用一个函数，执行变换操作，如图 5-1 所示。

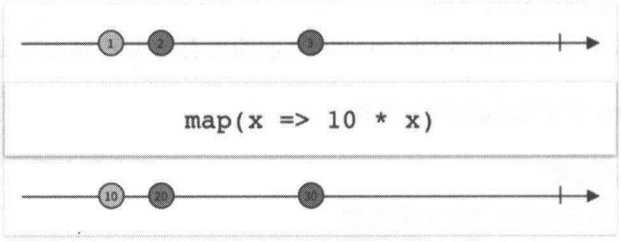


图 5-1

map 操作符对原始 Observable 发射的每一项数据应用一个你选择的函数，然后返回一个发射这些结果的 Observable。

RxJava 将这个操作符实现为 map 函数，这个操作符默认不在任何特定的调度器上执行。

示例代码：

```
Observable.just("HELLO")
        .map(new Function<String, String>() {

            @Override
            public String apply(String s) throws Exception {
                return s.toLowerCase();
            }
        })
        .map(new Function<String, String>() {

            @Override
            public String apply(String s) throws Exception {
                return s+" world";
            }
        })
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                System.out.println(s);
            }
        });
```

执行结果：

```
hello world
```

示例代码调用了两次 map 函数，表示它做了两次转换。第一次转换，将字符串“HELLO”转换成全是小写字母的字符串“hello”。第二次转换，在字符串“hello”后面添加新的字符串“ world”，它们组成了新的字符串也就是执行结果。

2. flatmap 操作符

flatMap 将一个发射数据的 Observable 变换为多个 Observables，然后将它们发射的数据合

并后放进一个单独的 Observable，如图 5-2 所示。

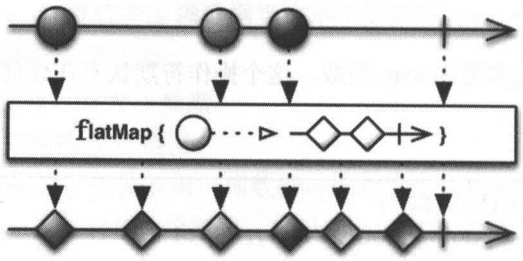


图 5-2

flatMap 操作符使用一个指定的函数对原始 Observable 发射的每一项数据执行变换操作，这个函数返回一个本身也发射数据的 Observable，然后 flatMap 合并这些 Observables 发射的数据，最后将合并后的结果当作它自己的数据序列发射。

下面看一个例子。先定义一个用户对象，包含用户名和地址，由于地址可能会包括生活、工作等地方，所以使用一个 List 对象来表示用户的地址。

示例代码：

```
import java.util.List;

/**
 * Created by tony on 2017/12/12.
 */
public class User {

    public String userName;
    public List<Address> addresses;

    public static class Address {

        public String street;
        public String city;
    }
}
```

如果想打印出某个用户所有的地址，那么可以借助 map 操作符返回一个地址的列表。

```
        User user = new User();
```

```
        user.userName = "tony";
        user.addresses = new ArrayList<>();
        User.Address address1 = new User.Address();
        address1.street = "ren ming road";
        address1.city = "Su zhou";
        user.addresses.add(address1);

        User.Address address2 = new User.Address();
        address2.street = "dong wu bei road";
        address2.city = "Su Zhou";
        user.addresses.add(address2);

        Observable.just(user)
                .map(new Function<User, List<User.Address>>() {

                    @Override
                    public List<User.Address> apply(User user) throws Exception
{
                        return user.addresses;
                    }
                })
                .subscribe(new Consumer<List<User.Address>>() {
                    @Override
                    public void accept(List<User.Address> addresses) throws
Exception {
                        for(User.Address address:addresses) {
                            System.out.println(address.street);
                        }
                    }
                });
```

执行结果：

```
ren ming road
dong wu bei road
```

换成 flatMap 操作符之后，flatMap 内部将用户的地址列表转换成一个 Observable。

```
        Observable.just(user)
                .flatMap(new Function<User, ObservableSource<User.Address>>() {
                    @Override
```

```
                public Observable<User.Address> apply(User user) throws
Exception {
                    return Observable.fromIterable(user.addresses);
                }
            })
            .subscribe(new Consumer<User.Address>() {
                @Override
                public void accept(User.Address address) throws Exception {
                    System.out.println(address.street);
                }
            });
```

执行结果与上面的相同。

flatMap 对这些 Observables 发射的数据做的是合并（merge）操作，因此它们可能是交错的。还有一个操作符不会让变换后的 Observables 发射的数据交错，它严格按照顺序发射这些数据，这个操作符就是 concatMap。

map 和 flatMap 是 RxJava 中使用率非常高的操作符，熟练掌握它们非常有必要，在笔者看来它们的重要性仅次于第 3 章的创建操作符。

5.2 groupBy

groupBy 操作符将一个 Observable 拆分为一些 Observables 集合，它们中的每一个都发射原始 Observable 的一个子序列，如图 5-3 所示。

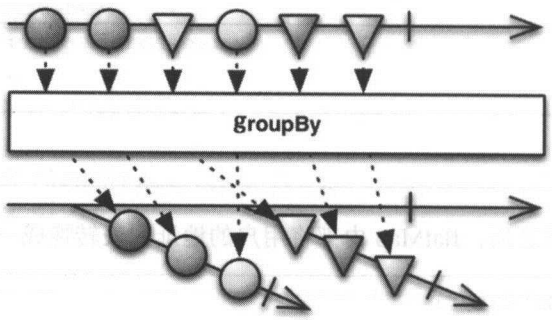


图 5-3

哪个数据项由哪一个Observable发射是由一个函数判定的，这个函数给每一项指定一个Key，Key 相同的数据会被同一个 Observable 发射。

最终返回的是 Observable 的一个特殊子类 GroupedObservable。它是一个抽象类。getKey() 方法是 GroupedObservable 的方法，这个 Key 用于将数据分组到指定的 Observable。

示例代码：

```
        Observable.range(1, 8).groupBy(new Function<Integer, String>() {
            @Override
            public String apply(Integer integer) throws Exception {
                return (integer % 2 == 0) ? "偶数组" : "奇数组";
            }
        }).subscribe(new Consumer<GroupedObservable<String, Integer>>() {
            @Override
            public void accept(GroupedObservable<String, Integer>
stringIntegerGroupedObservable) throws Exception {

                System.out.println("group name:" +
stringIntegerGroupedObservable.getKey());
            }
        });
```

执行结果：

```
group name:奇数组
group name:偶数组
```

对上述代码做一些修改，对 GroupedObservable 使用 getKey()方法，从而能够选出奇数组的 GroupedObservable，最后打印出该 GroupedObservable 下的全部成员。

```
        Observable.range(1, 8).groupBy(new Function<Integer, String>() {
            @Override
            public String apply(Integer integer) throws Exception {
                return (integer % 2 == 0) ? "偶数组" : "奇数组";
            }
        }).subscribe(new Consumer<GroupedObservable<String, Integer>>() {
            @Override
            public void accept(GroupedObservable<String, Integer>
stringIntegerGroupedObservable) throws Exception {
```

```
                if
(stringIntegerGroupedObservable.getKey().equalsIgnoreCase("奇数组")) {

                    stringIntegerGroupedObservable.subscribe(new
Consumer<Integer>() {
                        @Override
                        public void accept(Integer integer) {
                            System.out.println(stringIntegerGroupedObservable.
getKey() + "member: " + integer);
                        }
                    });
                }
            }
        });
```

执行结果：

```
奇数组 member: 1
奇数组 member: 3
奇数组 member: 5
奇数组 member: 7
```

5.3 buffer 和 window

1. buffer 操作符

buffer 会定期收集 Observable 数据并放进一个数据包裹，然后发射这些数据包裹，而不是一次发射一个值，如图 5-4 所示。

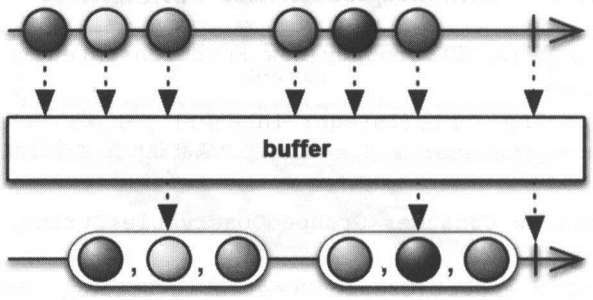


图 5-4

buffer 操作符将一个 Observable 变换为另一个，原来的 Observable 正常发射数据，由变换产生的 Observable 发射这些数据的缓存集合。

示例代码：

```
    Observable.range(1,10)
            .buffer(2)
            .subscribe(new Consumer<List<Integer>>() {
                @Override
                public void accept(List<Integer> integers) throws Exception
{
                    System.out.println("onNext:" + integers);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println("onError:");
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("onComplete:");
                }
            });
```

执行结果：

```
onNext:[1, 2]
onNext:[3, 4]
onNext:[5, 6]
onNext:[7, 8]
onNext:[9, 10]
onComplete:
```

上述代码发射了从 1 到 10 这 10 个数字，由于使用了 buffer 操作符，它会将原先的 Observable 转换成新的 Observable，而新的 Observable 每次可发射两个数字，发射完毕后调用 onComplete() 方法。

查看 buffer 操作符的源码，可以看到使用 buffer 操作符之后转换成 Observable<list<T>>。

```
</list\
```

```
    /**
     * Returns an Observable that emits buffers of items it collects from the
source ObservableSource. The resulting
     * ObservableSource emits connected, non-overlapping buffers, each
containing {@code count} items. When the source
     * ObservableSource completes or encounters an error, the resulting
ObservableSource emits the current buffer and
     * propagates the notification from the source ObservableSource.
     * <p>
     * <img width="640" height="320"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/buffe
r3.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>This version of {@code buffer} does not operate by default on a
particular {@link Scheduler}.</dd>
     * </dl>
     *
     * @param count
     *            the maximum number of items in each buffer before it should be
emitted
     * @return an Observable that emits connected, non-overlapping buffers, each
containing at most
     *         {@code count} items from the source ObservableSource
     * @see <a
href="http://reactivex.io/documentation/operators/buffer.html">ReactiveX
operators documentation: Buffer</a>
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.NONE)
    public final Observable<List<T>> buffer(int count) {
        return buffer(count, count);
    }
```

对上述代码做一些改动，将发射的数据变成 11。

```
        Observable.range(1,11)
                .buffer(2)
                .subscribe(new Consumer<List<Integer>>() {
                    @Override
                    public void accept(List<Integer> integers) throws Exception
{
```

```
                System.out.println("onNext:" + integers);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.out.println("onError:");
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("onComplete:");
            }
        });
```

执行结果：

```
onNext:[1, 2]
onNext:[3, 4]
onNext:[5, 6]
onNext:[7, 8]
onNext:[9, 10]
onNext:[11]
onComplete:
```

由于 11 不是偶数，所以最后一个数字被单独发射出来。

对上述代码再做一些改动，缓存 5 个数字。

```
    Observable.range(1,11)
            .buffer(5)
            .subscribe(new Consumer<List<Integer>>() {
                @Override
                public void accept(List<Integer> integers) throws Exception
{
                    System.out.println("onNext:" + integers);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println("onError:");
                }
            }, new Action() {
```

```
                @Override
                public void run() throws Exception {
                    System.out.println("onComplete:");
                }
            });
```

执行结果：

```
onNext:[1, 2, 3, 4, 5]
onNext:[6, 7, 8, 9, 10]
onNext:[11]
onComplete:
```

在 RxJava 中有许多 buffer 的重载方法，例如比较常用的 buffer(count, skip)。

buffer(count, skip)从原始 Observable 的第一项数据开始创建新的缓存，此后每当收到 skip 项数据，就用 count 项数据填充缓存：开头的一项和后续的 count–1 项。它以列表（List）的形式发射缓存，这些缓存可能会有重叠部分（比如 skip < count 时），也可能会有间隙（比如 skip > count 时），取决于 count 和 skip 的值。

示例代码：

```
    Observable.range(1,11)
            .buffer(5,1)
            .subscribe(new Consumer<List<Integer>>() {
                @Override
                public void accept(List<Integer> integers) throws Exception
{
                    System.out.println("onNext:" + integers);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println("onError:");
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("onComplete:");
                }
            });
```

执行结果：

```
onNext:[1, 2, 3, 4, 5]
onNext:[2, 3, 4, 5, 6]
onNext:[3, 4, 5, 6, 7]
onNext:[4, 5, 6, 7, 8]
onNext:[5, 6, 7, 8, 9]
onNext:[6, 7, 8, 9, 10]
onNext:[7, 8, 9, 10, 11]
onNext:[8, 9, 10, 11]
onNext:[9, 10, 11]
onNext:[10, 11]
onNext:[11]
onComplete:
```

如果原来的 Observable 发射了一个 onError 通知，那么 buffer 会立即传递这个通知，而不是首先发射缓存的数据，即使在这之前缓存中包含了原始 Observable 发射的数据。

window 操作符与 buffer 类似，但它在发射之前是把收集到的数据放进单独的 Observable，而不是放进一个数据结构。

2. window 操作符

定期将来自原始 Observable 的数据分解为一个 Observable 窗口，发射这些窗口，而不是每次发射一项数据，如图 5-5 所示。

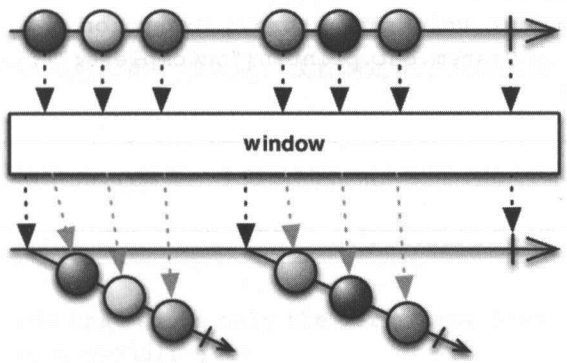


图 5-5

window 发射的不是原始 Observable 的数据包，而是 Observables，这些 Observables 中的每

一个都发射原始 Observable 数据的一个子集，最后发射一个 onComplete 通知。

示例代码：

```
        Observable.range(1,10)
                .window(2)
                .subscribe(new Consumer<Observable<Integer>>() {
                    @Override
                    public void accept(Observable<Integer> observable) throws
Exception {

                        System.out.println("onNext:");
                        observable.subscribe(new Consumer<Integer>() {
                            @Override
                            public void accept(Integer integer) throws Exception
{
                                System.out.println("accept:" + integer);
                            }
                        });
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        System.out.println("onError:");
                    }
                }, new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("onComplete:");
                    }
                });
```

执行结果：

```
onNext:
accept:1
accept:2
onNext:
accept:3
accept:4
onNext:
accept:5
```

```
accept:6
onNext:
accept:7
accept:8
onNext:
accept:9
accept:10
onComplete:
```

5.4　first 和 last

1. first 操作符

只发射第一项（或者满足某个条件的第一项）数据，如图 5-6 所示。

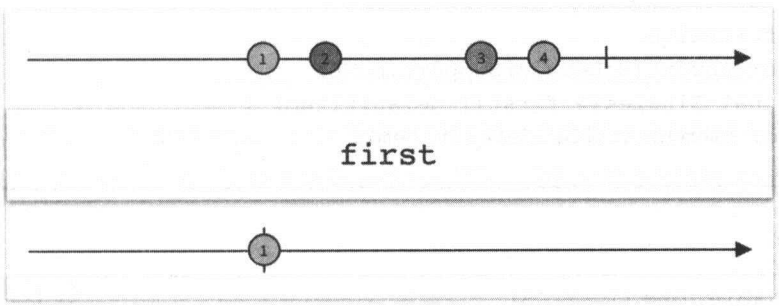


图 5-6

如果只对 Observable 发射的第一项数据，或者满足某个条件的第一项数据感兴趣，那么就可以使用 first 操作符。

在 RxJava 2.x 中，使用 first()需要一个默认的 Item，对于 Observable 而言，使用了 first()会返回 Single 类型。

```
/**
 * Returns a Single that emits only the very first item emitted by the source
ObservableSource, or a default item
 * if the source ObservableSource completes without emitting any items.
 * <p>
```

```
     * <img width="640" height="286"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/first
.s.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>{@code first} does not operate by default on a particular {@link
Scheduler}.</dd>
     * </dl>
     *
     * @param defaultItem
     *            the default item to emit if the source ObservableSource doesn't
emit anything
     * @return the new Single instance
     * @see <a
href="http://reactivex.io/documentation/operators/first.html">ReactiveX
operators documentation: First</a>
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.NONE)
    public final Single<T> first(T defaultItem) {
        return elementAt(0L, defaultItem);
    }
```

示例代码：

```
    Observable.just(1, 2, 3)
            .first(1)
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            });
```

执行结果：

```
Next: 1
```

如果 Observable 不发射任何数据，那么 first 操作符的默认值就起了作用。

```
Observable.<Integer>empty()
        .first(1)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        });
```

执行结果：

```
Next: 1
```

在 RxJava 2.x 中，还有 firstElement 操作符表示只取第一个数据，没有默认值。firstOrError 操作符表示要么能取到第一个数据，要么执行 onError 方法，它们分别返回 Maybe 类型和 Single 类型。

2. last 操作符

只发射最后一项（或者满足某个条件的最后一项）数据，如图 5-7 所示。

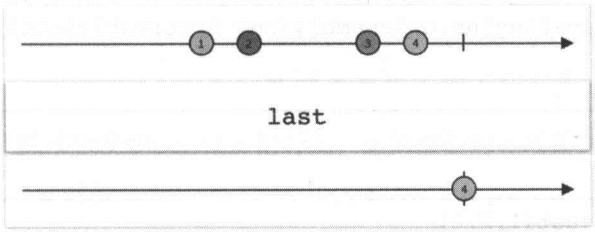


图 5-7

如果只对 Observable 发射的最后一项数据，或者满足某个条件的最后一项数据感兴趣，那么就可以使用 last 操作符。

last 操作符跟 first 操作符类似，需要一个默认的 Item，也是返回 Single 类型。

```
    /**
     * Returns a Single that emits only the last item emitted by this Observable,
or a default item
     * if this Observable completes without emitting any items.
     * <p>
     * <img width="640" height="305"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/last.
2.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>{@code last} does not operate by default on a particular {@link
Scheduler}.</dd>
     * </dl>
     *
     * @param defaultItem
     *            the default item to emit if the source ObservableSource is empty
     * @return an Observable that emits only the last item emitted by the source
ObservableSource, or a default item
     *         if the source ObservableSource is empty
     * @see <a
href="http://reactivex.io/documentation/operators/last.html">ReactiveX
operators documentation: Last</a>
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.NONE)
    public final Single<T> last(T defaultItem) {
        ObjectHelper.requireNonNull(defaultItem, "defaultItem is null");
        return RxJavaPlugins.onAssembly(new ObservableLastSingle<T>(this,
defaultItem));
    }
```

示例代码：

```
        Observable.just(1,2,3)
                .last(3)
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println("Next: " + integer);
                    }
```

```
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        });
```

执行结果：

```
Next: 3
```

在 RxJava 2.x 中，有 lastElement 操作符和 lastOrError 操作符。

5.5　take 和 takeLast

1. take 操作符

只发射前面的 *n* 项数据，如图 5-8 所示。

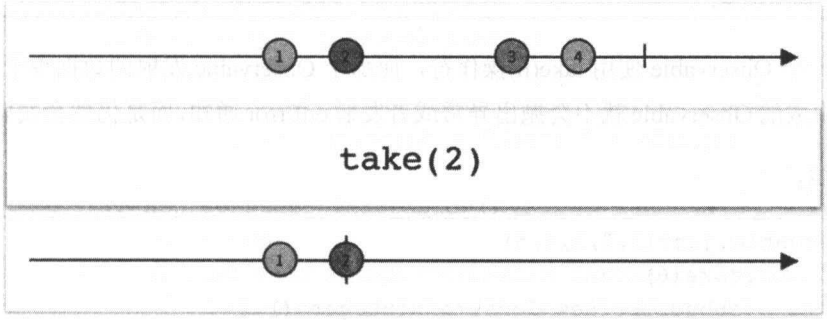


图 5-8

使用 take 操作符可以只修改 Observable 的行为，返回前面的 *n* 项数据，发射完成通知，忽略剩余的数据。

示例代码：

```
Observable.just(1,2,3,4,5)
        .take(3)
        .subscribe(new Consumer<Integer>() {
            @Override
```

```
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 1
Next: 2
Next: 3
Sequence complete.
```

如果对一个 Observable 使用 take(n)操作符，而那个 Observable 发射的数据少于 *n* 项，那么 take 操作符生成的 Observable 就不会抛出异常或者发射 onError 通知，而是仍然会发射那些数据。

示例代码：

```
    Observable.just(1,2,3,4,5)
            .take(6)
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
```

```
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Sequence complete.
```

take 有一个重载方法能够接受一个时长而不是数量参数。它会丢掉发射 Observable 开始的那段时间发射的数据，时长和时间单位通过参数指定。

示例代码：

```
Observable
        .intervalRange(0,10,1,1, TimeUnit.SECONDS)
        .take(3, TimeUnit.SECONDS)
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {
                System.out.println("Next: " + aLong);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });

try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
```

```
        }
```

执行结果：

```
Next: 0
Next: 1
Next: 2
Sequence complete.
```

上述代码使用了 intervalRange 操作符表示每隔 1s 会发射 1 个数据，它们从 0 开始到 9 结束，发射 10 个数据。由于在这里使用了 take 操作符，最后只打印前 3 个数据。

take 的这个重载方法默认在 computation 调度器上执行，也可以使用参数来指定其他调度器。

2. takeLast 操作符

发射 Observable 发射的最后 *n* 项数据，如图 5-9 所示。

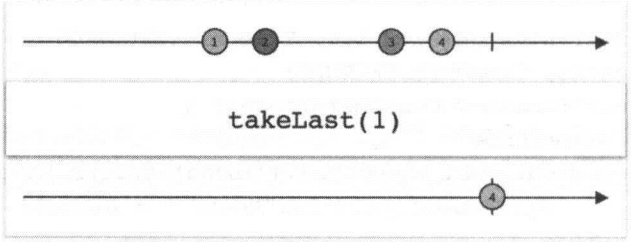


图 5-9

使用 takeLast 操作符修改原始 Observable，我们可以只发射 Observable 发射的最后 *n* 项数据，忽略前面的数据。

```
    Observable.just(1,2,3,4,5)
            .takeLast(3)
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
```

```
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 3
Next: 4
Next: 5
Sequence complete.
```

同样，如果对一个 Observable 使用 takeLast(n)操作符，而那个 Observable 发射的数据少于 *n* 项，那么 takeLast 操作符生成的 Observable 不会抛出异常或者发射 onError 通知，而是仍然发射那些数据。

```
Observable.just(1,2,3,4,5)
        .takeLast(6)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 1
```

```
Next: 2
Next: 3
Next: 4
Next: 5
Sequence complete.
```

takeLast 也有一个重载方法能够接受一个时长而不是数量参数。它会发射在原始 Observable 生命周期内最后一段时间发射的数据，时长和时间单位通过参数指定。

```
Observable
        .intervalRange(0,10,1,1, TimeUnit.SECONDS)
        .takeLast(3, TimeUnit.SECONDS)
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {
                System.out.println("Next: " + aLong);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });

try {
    Thread.sleep(10000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

执行结果：

```
Next: 7
Next: 8
Next: 9
Sequence complete.
```

takeLast 的这个重载方法默认在 computation 调度器上执行，也可以使用参数来指定其他的调度器。

5.6　skip 和 skipLast

skip 操作者

抑制 Observable 发射的前 *n* 项数据，如图 5-10 所示。

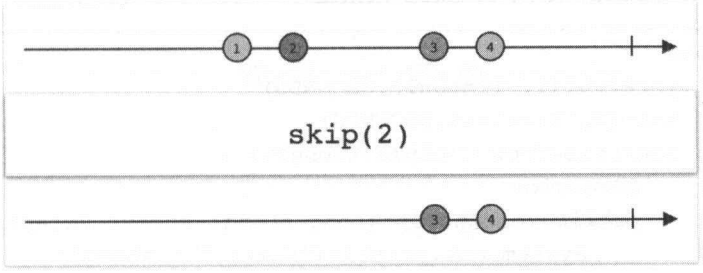


图 5-10

使用 skip 操作符，可以忽略 Observable 发射的前 *n* 项数据，只保留之后的数据。

示例代码：

```
Observable.just(1,2,3,4,5)
        .skip(3)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
```

```
            }
        });
```

执行结果：

```
Next: 4
Next: 5
Sequence complete.
```

skip 有一个重载方法能够接受一个时长而不是数量参数。它会丢弃原始 Observable 开始那段时间发射的数据，时长和时间单位通过参数指定。

```
    Observable
            .interval(1, TimeUnit.SECONDS)
            .skip(3, TimeUnit.SECONDS)
            .subscribe(new Consumer<Long>() {
                @Override
                public void accept(Long aLong) throws Exception {
                    System.out.println("Next: " + aLong);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

执行结果：

```
Next: 3
Next: 4
```

```
Next: 5
Next: 6
Next: 7
Next: 8
Next: 9
```

这个重载方法默认在 computation 调度器上执行，也可以使用参数来指定其他的调度器。

2. skipLast 操作符

抑制 Observable 发射的后 *n* 项数据，如图 5-11 所示。

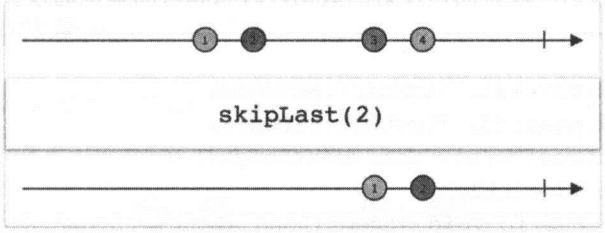


图 5-11

使用 skipLast 操作符修改原始 Observable，可以忽略 Observable 发射的后 *n* 项数据，只保留前面的数据。

示例代码：

```
Observable.just(1,2,3,4,5)
        .skipLast(3)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
```

```
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 1
Next: 2
Sequence complete.
```

同样，skipLast 也有一个重载方法接受一个时长而不是数量参数。它会丢弃在原始 Observable 生命周期最后一段时间内发射的数据，时长和时间单位通过参数指定。

```
    Observable
            .interval(1, TimeUnit.SECONDS)
            .skipLast(3, TimeUnit.SECONDS)
            .subscribe(new Consumer<Long>() {
                @Override
                public void accept(Long aLong) throws Exception {
                    System.out.println("Next: " + aLong);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });

    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

执行结果：

```
Next: 0
```

```
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Next: 6
```

这个重载方法默认在 computation 调度器上执行，也可以使用参数来指定其他调度器。

5.7　elementAt 和 ignoreElements

1. elementAt 操作符

只发射第 *n* 项数据，如图 5-12 所示。

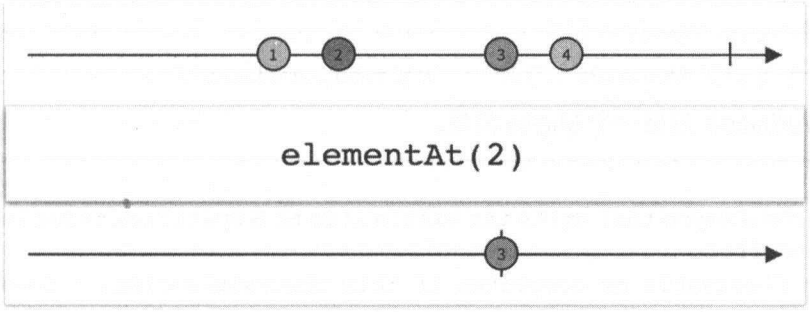


图 5-12

elementAt 操作符获取原始 Observable 发射的数据序列指定索引位置的数据项，然后当作自己的唯一数据发射。

它传递一个基于 0 的索引值，发射原始 Observable 数据序列对应索引位置的值，如果传递给 elementAt 的值为 5，那么它会发射第 6 项数据。如果传递的是一个负数，则将会抛出一个 IndexOutOfBoundsException 异常。

示例代码：

```
    Observable.just(1,2,3,4,5)
            .elementAt(2)
            .subscribe(new Consumer<Integer>() {
```

```
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 3
```

elementAt(index) 返回一个 Maybe 类型。

```
   /**
     * Returns a Maybe that emits the single item at a specified index in a sequence
of emissions from
     * this Observable or completes if this Observable signals fewer elements
than index.
     * <p>
     * <img width="640" height="310"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/eleme
ntAt.2m.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>{@code elementAt} does not operate by default on a particular {@link
Scheduler}.</dd>
     * </dl>
     *
     * @param index
     *            the zero-based index of the item to retrieve
     * @return a Maybe that emits a single item: the item at the specified position
in the sequence of
     *         those emitted by the source ObservableSource
```

```
     * @throws IndexOutOfBoundsException
     *             if {@code index} is less than 0
     * @see <a
href="http://reactivex.io/documentation/operators/elementat.html">ReactiveX
operators documentation: ElementAt</a>
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.NONE)
    public final Maybe<T> elementAt(long index) {
        if (index < 0) {
            throw new IndexOutOfBoundsException("index >= 0 required but it was
" + index);
        }
        return RxJavaPlugins.onAssembly(new ObservableElementAtMaybe<T>(this,
index));
    }
```

如果原始 Observable 的数据项数小于 index+1，那么会调用 onComplete()方法（在 RxJava 1.x 中也会抛出一个 IndexOutOfBoundsException 异常）。所以，elementAt 还提供了一个带默认值的方法，它返回一个 Single 类型。

```
        Observable.just(1,2,3,4,5)
                .elementAt(10,0)
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println("Next: " + integer);
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        System.err.println("Error: " + throwable.getMessage());
                    }
                });
```

执行结果：

```
Next: 0
```

如果 index 超出了索引范围，那么取默认值 0。

跟 first、last 操作符类似，element 还有一个 elementAtOrError 操作符，它表示要么能取到指定索引位置的数据，要么执行 onError 方法，也是返回 Single 类型。

2. ignoreElements

不发射任何数据，只发射 Observable 的终止通知，如图 5-13 所示。

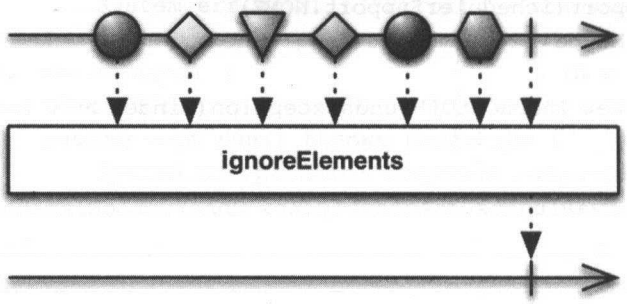


图 5-13

ignoreElements 操作符抑制原始 Observable 发射的所有数据，只允许它的终止通知（onError 或 onComplete）通过。它返回一个 Completable 类型。

如果我们不关心一个 Observable 发射的数据，但是希望在它完成时或遇到错误终止时收到通知，那么就可以对 Observable 使用 ignoreElements 操作符，它将确保永远不会调用观察者的 onNext()方法。

示例代码：

```
Observable.just(1,2,3,4,5)
        .ignoreElements()
        .subscribe(new Action() {
            @Override
            public void run() throws Exception {

                System.out.println("Sequence complete.");
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {

                System.err.println("Error: " + throwable.getMessage());
```

```
                }
            });
```

执行结果：

```
Sequence complete.
```

5.8　distinct 和 filter

1. distinct 操作符

过滤掉重复的数据项，如图 5-14 所示。

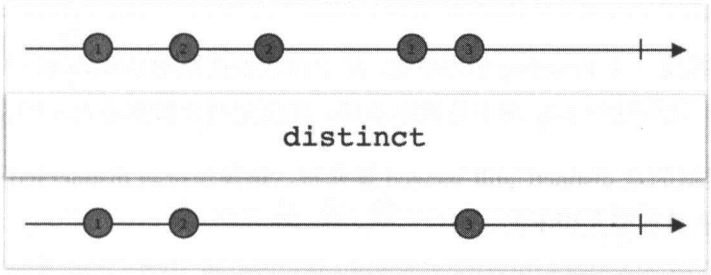


图 5-14

distinct 的过滤规则是：只允许还没有发射过的数据项通过。

示例代码：

```
Observable.just(1, 2, 1,2,3, 4, 5,5,6)
        .distinct()
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
```

```
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Next: 6
Sequence complete.
```

distinct 还能接受一个 Function 作为参数，这个函数根据原始 Observable 发射的数据项产生一个 Key，然后，比较这些 Key 而不是数据本身，来判定两个数据是否不同。

与 distinct 类似的是 distinctUntilChanged 操作符，该操作符与 distinct 的区别是，它只判定一个数据和它的直接前驱是否不同。

```
    Observable.just(1, 2, 1,2,3, 4, 5,5,6)
            .distinctUntilChanged()
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 1
Next: 2
Next: 1
Next: 2
Next: 3
Next: 4
Next: 5
Next: 6
Sequence complete.
```

在这段代码中，发射的数据里有两个 5 是彼此相邻的，通过 distinctUntilChanged 操作符过滤了其中一个。

2. filter 操作符

只发射通过谓词测试的数据项，如图 5-15 所示。

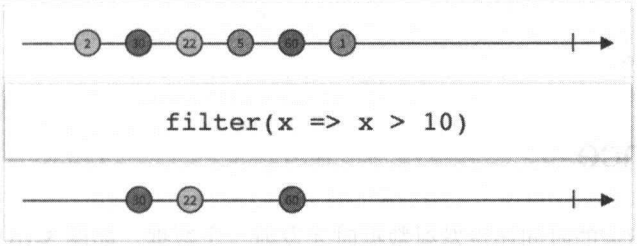


图 5-15

filter 操作符使用你指定的一个谓词函数测试数据项，只有通过测试的数据才会被发射。

示例代码：

```
Observable.just(2,30,22,5,60,1)
        .filter(new Predicate<Integer>() {
            @Override
            public boolean test(Integer integer) throws Exception {
                return integer>10;
            }
        })
        .subscribe(new Consumer<Integer>() {
```

```
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 30
Next: 22
Next: 60
Sequence complete.
```

5.9 debounce

仅在过了一段指定的时间还没发射数据时才发射一个数据，如图 5-16 所示。

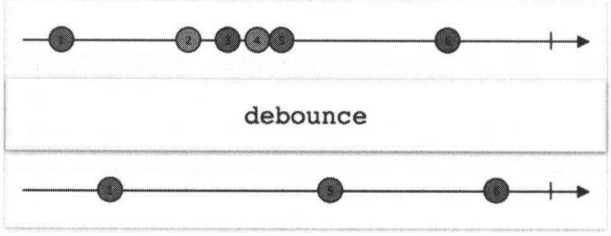


图 5-16

debounce 操作符会过滤掉发射速率过快的数据项。

示例代码：

```
        Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
                if (e.isDisposed()) return;
                try {
                    for (int i = 1; i <= 10; i++) {
                        e.onNext(i);   //发射数据
                        Thread.sleep(i * 100);
                    }
                    e.onComplete();
                } catch (Exception ex) {
                    e.onError(ex);
                }
            }
        }).debounce(500, TimeUnit.MILLISECONDS)
//如果发射数据间隔少于 400ms，就过滤拦截
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println("Next: " + integer);
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        System.err.println("Error: " + throwable.getMessage());
                    }
                }, new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("Sequence complete.");
                    }
                });

        try {
            Thread.sleep(10000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果：

```
Next: 6
Next: 7
Next: 8
Next: 9
Next: 10
Sequence complete.
```

debounce 还有另外一种形式，即使用一个 Function 函数来限制发送的数据。

跟 debounce 类似的是 throttleWithTimeout 操作符，它与只使用时间参数来限流的 debounce 的功能相同。

5.10 小结

变换操作符的作用是对被观察者发射的数据按照一定规则做一些变换操作，然后将变换后的数据发射出去。

过滤操作符的作用是过滤掉被观察者发射的一些数据，不让它们发射出去，也就是忽略并丢弃。至于需要过滤哪些数据，则需要按照不同的规则来进行。

第6章
条件操作符和布尔操作符

RxJava 的条件操作符主要包括以下几个。

- amb()：给定多个 Observable，只让第一个发射数据的 Observable 发射全部数据。
- defaultIfEmpty()：发射来自原始 Observable 的数据，如果原始 Observable 没有发射数据，则发射一个默认数据。
- skipUntil()：丢弃原始 Observable 发射的数据，直到第二个 Observable 发射了一个数据，然后发射原始 Observable 的剩余数据。
- skipWhile()：丢弃原始 Observable 发射的数据，直到一个特定的条件为假，然后发射原始 Observable 剩余的数据。
- takeUntil()：发射来自原始 Observable 的数据，直到第二个 Observable 发射了一个数据或一个通知。
- takeWhile() and takeWhileWithIndex()：发射原始 Observable 的数据，直到一个特定的条件为真，然后跳过剩余的数据。

RxJava 的布尔操作符主要包括：

- all()：判断是否所有的数据项都满足某个条件。
- contains()：判断 Observable 是否会发射一个指定的值。
- exists() and isEmpty()：判断 Observable 是否发射了一个值。
- sequenceEqual()：判断两个 Observables 发射的序列是否相等。

6.1 all、contains 和 amb

1. all 操作符

判定 Observable 发射的所有数据是否都满足某个条件，如图 6-1 所示。

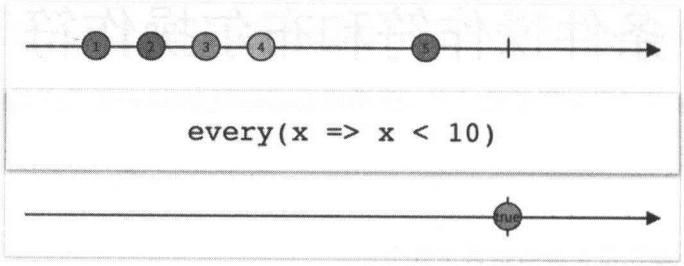


图 6-1

传递一个谓词函数给 all 操作符，这个函数接受原始 Observable 发射的数据，根据计算返回一个布尔值。all 返回一个只发射单个布尔值的 Observable，如果原始 Observable 正常终止并且每一项数据都满足条件，就返回 true；如果原始 Observable 的任意一项数据不满足条件，就返回 false。

示例代码：

```
Observable.just(1,2,3,4,5)
        .all(new Predicate<Integer>() {
            @Override
            public boolean test(Integer integer) throws Exception {
                return integer<10;
            }
        })
        .subscribe(new Consumer<Boolean>() {
            @Override
            public void accept(Boolean aBoolean) throws Exception {

                System.out.println(aBoolean);
            }
        });
```

执行结果：

```
true
```

对条件做一下修改，判断 Observable 发射的所有数据是否都大于 3。

```
Observable.just(1,2,3,4,5)
        .all(new Predicate<Integer>() {
            @Override
            public boolean test(Integer integer) throws Exception {
                return integer>3;
            }
        })
        .subscribe(new Consumer<Boolean>() {
            @Override
            public void accept(Boolean aBoolean) throws Exception {

                System.out.println(aBoolean);
            }
        });
```

执行结果：

```
false
```

all 操作符默认不在任何特定的调度器上执行。

2. contains 操作符

判定一个 Observable 是否发射了一个特定的值，如图 6-2 所示。

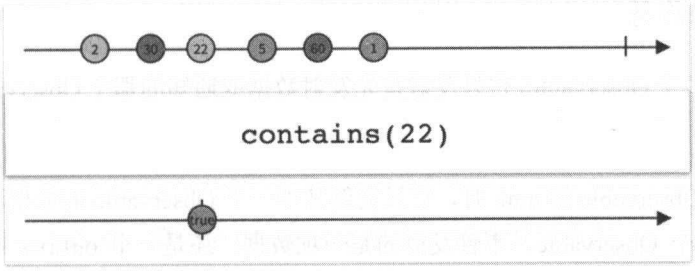


图 6-2

给 contains 传一个指定的值，如果原始 Observable 发射了那个值，那么返回的 Observable 将发射 true，否则发射 false。与它相关的一个操作符是 isEmpty，用于判定原始 Observable 是否

未发射任何数据。

示例代码：

```
Observable.just(2,30,22,5,60,1)
        .contains(22)
        .subscribe(new Consumer<Boolean>() {
            @Override
            public void accept(Boolean aBoolean) throws Exception {
                System.out.println("contains(22):"+aBoolean);
            }
        });

Observable.just(2,30,22,5,60,1)
        .isEmpty()
        .subscribe(new Consumer<Boolean>() {
            @Override
            public void accept(Boolean aBoolean) throws Exception {
                System.out.println("isEmpty():"+aBoolean);
            }
        });
```

执行结果：

```
contains(22):true
isEmpty():false
```

contains 默认不在任何特定的调度器上执行。

2. amb 操作符

给定两个或多个 Observable，它只发射首先发射数据或通知的那个 Observable 的所有数据，如图 6-3 所示。

当传递多个 Observable 给 amb 时，它只发射其中一个 Observable 的数据和通知：首先发送通知给 amb 的那个 Observable，不管发射的是一项数据，还是一个 onError 或是 onCompleted 通知。amb 将忽略和丢弃其他所有 Observables 的发射物。

在 RxJava 中，amb 还有一个类似的操作符 ambWith。例如，Observable.amb(o1,o2)和 o1.ambWith(o2)是等价的。

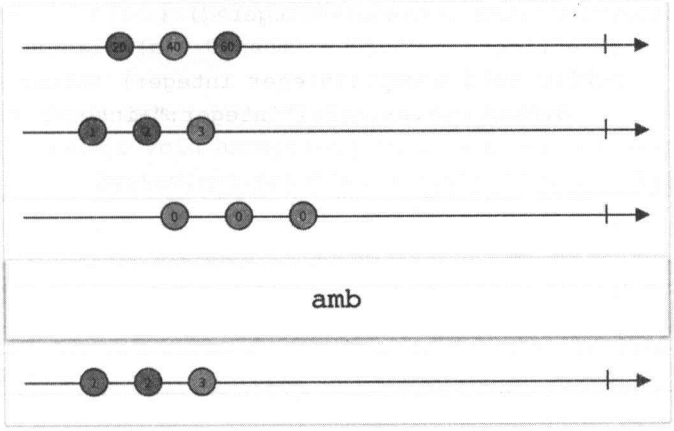


图 6-3

在 RxJava 2.x 中，amb 需要传递一个 Iterable 对象，或者使用 ambArray 来传递可变参数。

示例代码：

```
Observable.ambArray(
        Observable.just(1,2,3),
        Observable.just(4,5,6))
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("integer:"+integer);
            }
        });
```

执行结果：

```
integer:1
integer:2
integer:3
```

对代码稍微做一下改动，第一个 Observable 延迟 1s 后再发射数据

```
Observable.ambArray(
        //第一个 Observable 延迟 1 秒发射数据
        Observable.just(1,2,3).delay(1,TimeUnit.SECONDS),
        Observable.just(4,5,6))
```

```
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println("integer:"+integer);
                    }
                });
```

执行结果：

```
integer:4
integer:5
integer:6
```

由于第一个 Observable 延迟发射，因此我们只消费了第二个 Observable 的数据，第一个 Observable 发射的数据就不再处理了。

6.2 defaultIfEmpty

发射来自原始 Observable 的值，如果原始 Observable 没有发射任何值，就发射一个默认值，如图 6-4 所示。

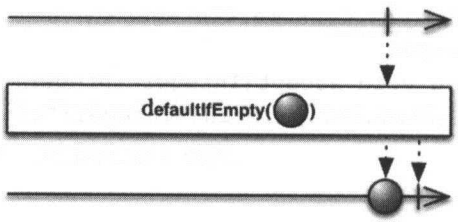


图 6-4

defaultIfEmpty 简单精确地发射原始 Observable 的值，如果原始 Observable 没有发射任何数据，就正常终止（以 onComplete 的形式了），那么 defaultIfEmpty 返回的 Observable 就发射一个我们提供的默认值。

defaultIfEmpty 默认不在任何特定的调度器上执行。

示例代码：

```
    Observable.empty()
```

```
                .defaultIfEmpty(8)
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(Object o) throws Exception {
                        System.out.println("defaultIfEmpty():"+o);
                    }
                });
```

执行结果：

```
defaultIfEmpty():8
```

在 defaultIfEmpty 方法内部，其实调用的是 switchIfEmpty 操作符，源码如下：

```
    /**
     * Returns an Observable that emits the items emitted by the source
ObservableSource or a specified default item
     * if the source ObservableSource is empty.
     * <p>
     * <img width="640" height="305"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/defau
ltIfEmpty.png" alt="">
     * <dl>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>{@code defaultIfEmpty} does not operate by default on a particular
{@link Scheduler}.</dd>
     * </dl>
     *
     * @param defaultItem
     *            the item to emit if the source ObservableSource emits no items
     * @return an Observable that emits either the specified default item if the
source ObservableSource emits no
     *         items, or the items emitted by the source ObservableSource
     * @see <a
href="http://reactivex.io/documentation/operators/defaultifempty.html">Reac
tiveX operators documentation: DefaultIfEmpty</a>
     */
    @CheckReturnValue
    @SchedulerSupport(SchedulerSupport.NONE)
    public final Observable<T> defaultIfEmpty(T defaultItem) {
        ObjectHelper.requireNonNull(defaultItem, "defaultItem is null");
```

```
        return switchIfEmpty(just(defaultItem));
    }
```

defaultIfEmpty 和 switchIfEmpty 的区别是，defaultIfEmpty 操作符只能在被观察者不发送数据时发送一个默认的数据，如果想要发送更多数据，则可以使用 switchIfEmpty 操作符，发送自定义的被观察者作为替代。

```
    Observable.empty()
            .switchIfEmpty(Observable.just(1, 2, 3))
            .subscribe(new Consumer<Object>() {

                @Override
                public void accept(Object o) throws Exception {
                    System.out.println("switchIfEmpty():"+o);
                }
            });
```

执行结果：

```
switchIfEmpty():1
switchIfEmpty():2
switchIfEmpty():3
```

6.3 sequenceEqual

判定两个 Observable 是否发射相同的数据序列，如图 6-5 所示。

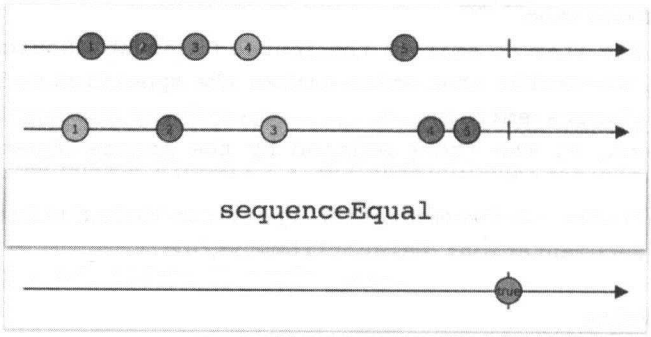


图 6-5

传递两个 Observable 给 sequenceEqual 操作符时，它会比较两个 Observable 的发射物，如果两个序列相同（相同的数据，相同的顺序，相同的终止状态），则发射 true，否则发射 false。

示例代码：

```
    Observable.sequenceEqual(
            Observable.just(1,2,3,4,5),
            Observable.just(1,2,3,4,5))
            .subscribe(new Consumer<Boolean>() {
                @Override
                public void accept(Boolean aBoolean) throws Exception {

                    System.out.println("sequenceEqual:"+aBoolean);
                }
            });
```

执行结果：

```
sequenceEqual:true
```

对代码稍微做一下改动，将两个 Observable 改成不一致。

```
    Observable.sequenceEqual(
            Observable.just(1,2,3,4,5),
            Observable.just(1,2,3,4,5,6))
            .subscribe(new Consumer<Boolean>() {
                @Override
                public void accept(Boolean aBoolean) throws Exception {

                    System.out.println("sequenceEqual:"+aBoolean);
                }
            });
```

执行结果：

```
sequenceEqual:false
```

sequenceEqual 还有一个版本接受第三个参数，可以传递一个函数用于比较两个数据项是否相同。对于复杂对象的比较，用三个参数的版本更为合适。

```
    Observable.sequenceEqual(
            Observable.just(4, 5, 6),
```

```
                Observable.just(4, 5, 6),
                new BiPredicate<Integer, Integer>() {
                    @Override
                    public boolean test(Integer integer, Integer integer2) throws
Exception {
                        return integer == integer2;
                    }
                })
                .subscribe(new Consumer<Boolean>() {
                    @Override
                    public void accept(Boolean aBoolean) throws Exception {

                        System.out.println("sequenceEqual:"+aBoolean);
                    }
                });
```

执行结果：

```
sequenceEqual:true
```

sequenceEqual 这个操作符默认不在任何特定的调度器上执行。

6.4 skipUntil 和 skipWhile

1. skipUntil 操作符

丢弃原始 Observable 发射的数据，直到第二个 Observable 发射了一项数据，如图 6-6 所示。

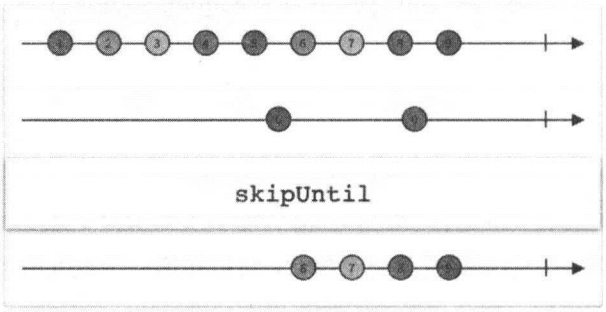


图 6-6

skipUntil 订阅原始的 Observable，但是忽略它的发射物，直到第二个 Observable 发射一项数据那一刻，它才开始发射原始 Observable。 skipUntil 默认不在任何特定的调度器上执行。

示例代码：

```
Observable.intervalRange(1, 9, 0, 1, TimeUnit.MILLISECONDS)
        .skipUntil(Observable.timer(4, TimeUnit.MILLISECONDS))
        .subscribe(new Consumer<Long>() {
            @Override
            public void accept(Long aLong) throws Exception {
                System.out.println(String.valueOf(aLong));
            }
        });

try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

执行结果：

```
6
7
8
9
```

上述代码中，原始 Observable 发射 1 到 9 这 9 个数字，初始延迟时间是 0，每个间隔 1ms。由于使用了 skipUntil 操作符，因此它会发射原始 Observable 在 3ms 之后的数据。

2. skipWhile 操作符

丢弃 Observable 发射的数据，直到一个指定的条件不成立，如图 6-7 所示。

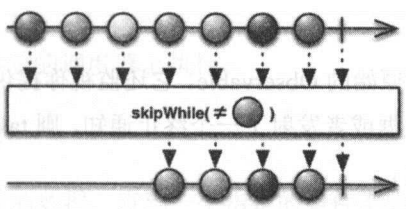


图 6-7

skipWhile 订阅原始的 Observable，但是忽略它的发射物，直到指定的某个条件变为 false，它才开始发射原始的 Observable。 skipWhile 默认不在任何特定的调度器上执行。

示例代码：

```
Observable.just(1,2,3,4,5)
        .skipWhile(new Predicate<Integer>() {
            @Override
            public boolean test(Integer integer) throws Exception {
                return integer<=2;
            }
        })
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println(String.valueOf(integer));
            }
        });
```

执行结果：

```
3
4
5
```

对于小于等于 2 的数字就被忽略了，所以打印的结果是 3、4、5。

6.5 takeUntil 和 takeWhile

1. takeUntil 操作符

当第二个 Observable 发射了一项数据或者终止时，丢弃原始 Observable 发射的任何数据，如图 6-8 所示。

takeUntil 订阅并开始发射原始的 Observable，它还监视你提供的第二个 Observable。如果第二个 Observable 发射了一项数据或者发射了一个终止通知，则 takeUntil 返回的 Observable 会停止发射原始 Observable 并终止。

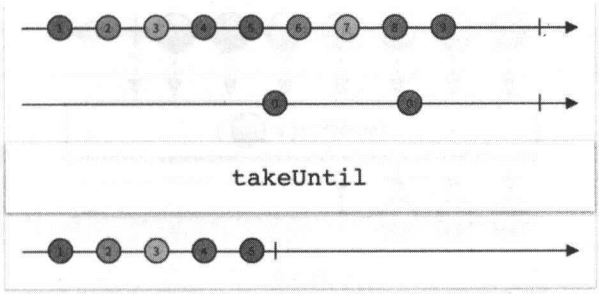


图 6-8

示例代码：

```
Observable.just(1,2,3,4,5,6,7,9)
        .takeUntil(new Predicate<Integer>(){

            @Override
            public boolean test(Integer i) throws Exception {
                return i==5;
            }
        })
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println(String.valueOf(integer));
            }
        });
```

执行结果：

```
1
2
3
4
5
```

takeUntil 默认不在任何特定的调度器上执行。

2. takeWhile 操作符

发射原始 Observable 发射的数据，直到一个指定的条件不成立，如图 6-9 所示。

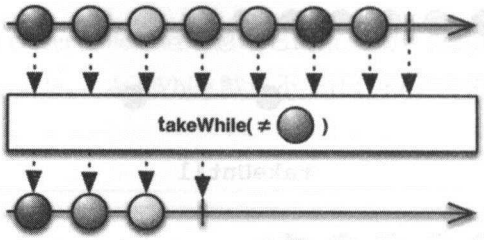


图 6-9

takeWhile 发射原始的 Observable，直到某个指定的条件不成立，它会立即停止发射原始 Observable，并终止自己的 Observable。

RxJava 中的 takeWhile 操作符返回一个原始 Observable 行为的 Observable，直到某项数据我们指定的函数返回 false，这个新的 Observable 就会发射 onComplete 终止通知。

示例代码：

```
Observable.just(1,2,3,4,5)
        .takeWhile(new Predicate<Integer>() {
            @Override
            public boolean test(Integer integer) throws Exception {
                return integer<=2;
            }
        })
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println(String.valueOf(integer));
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.out.println(throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("onComplete");
            }
        });
```

执行结果：

```
1
2
onComplete
```

takeWhile 默认不在任何特定的调度器上执行。

6.4 节和 6.5 节介绍的操作符在功能上是相互对应的，所以在使用上非常相似。

6.6　小结

在 RxJava 中，布尔操作符返回的结果全部为 boolean 值，而条件操作符会根据条件进行数据发射或变换被观察者。

第7章
合并操作符与连接操作符

RxJava 的合并操作符主要包括如下几个。

- ◎ startWith( )：在数据序列的开头增加一项数据。
- ◎ merge( )：将多个 Observable 合并为一个。
- ◎ mergeDelayError( )：合并多个 Observable，让没有错误的 Observable 都完成后再发射错误通知。
- ◎ zip( )：使用一个函数组合多个 Observable 发射的数据集合，然后再发射这个结果。
- ◎ combineLatest( )：当两个 Observable 中的任何一个发射了一个数据时，通过一个指定的函数组合每个 Observable 发射的最新数据（一共两个数据），然后发射这个函数的结果。
- ◎ join( ) and groupJoin( )：无论何时，如果一个 Observable 发射了一个数据项，就需要在另一个 Observable 发射的数据项定义的时间窗口内，将两个 Observable 发射的数据合并发射。
- ◎ switchOnNext( )：将一个发射 Observable 的 Observable 转换成另一个 Observable，后者发射这些 Observable 最近发射的数据。

RxJava 的连接操作符，主要是 ConnectableObservable 所使用的操作符和 Observable 所使用的操作符。

- ◎ ConnectableObservable.connect( )：指示一个可连接的 Observable 开始发射数据。
- ◎ Observable.publish( )：将一个 Observable 转换为一个可连接的 Observable。
- ◎ Observable.replay( )：确保所有的订阅者看到相同的数据序列，即使它们在 Observable 开始发射数据之后才订阅。

◎ ConnectableObservable.refCount()：让一个可连接的 Observable 表现得像一个普通的 Observable。

7.1 merge 和 zip

1. merge 操作符

合并多个 Observable 的发射物，如图 7-1 所示。

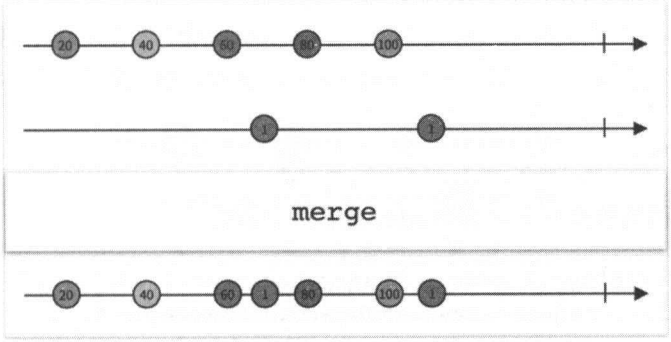


图 7-1

merge 操作符可以将多个 Observable 的输出合并，使得它们就像是单个的 Observable 一样。

示例代码：

```
Observable<Integer> odds = Observable.just(1, 3, 5);
Observable<Integer> evens = Observable.just(2, 4, 6);

Observable.merge(odds, evens)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
```

```
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 1
Next: 3
Next: 5
Next: 2
Next: 4
Next: 6
Sequence complete.
```

对上面的代码稍微修改一下，其中 odds 多了两个数据。

```
Observable<Integer> odds = Observable.just(1, 3, 5,7,9);
Observable<Integer> evens = Observable.just(2, 4, 6);

Observable.merge(odds, evens)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println("Next: " + integer);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                System.err.println("Error: " + throwable.getMessage());
            }
        }, new Action() {
            @Override
            public void run() throws Exception {
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 1
Next: 3
Next: 5
Next: 7
Next: 9
Next: 2
Next: 4
Next: 6
Sequence complete.
```

merge 是按照时间线并行的。如果传递给 merge 的任何一个 Observable 发射了 onError 通知终止，则 merge 操作符生成的 Observable 也会立即以 onError 通知终止。如果想让它继续发射数据，直到最后才报告错误，则可以使用 mergeDelayError 操作符，如图 7-2 所示。

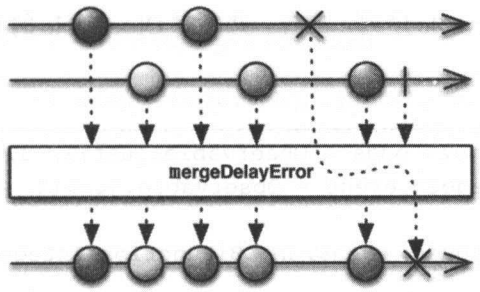


图 7-2

如果只是两个被观察者合并，则还可以使用 mergeWith 操作符，Observable.merge(odds,evens) 等价于 odds.mergeWith(evens)。

merge 操作符最多只能合并 4 个被观察者，如果需要合并更多个被观察者，则可以使用 mergeArray 操作符。

2. zip

通过一个函数将多个 Observable 的发射物结合到一起，基于这个函数的结果为每个结合体发射单个数据项，如图 7-3 所示。

zip 操作符返回一个 Obversable，它使用这个函数按顺序结合两个或多个 Observable 发射的数据项，然后发射这个函数返回的结果。它按照严格的顺序应用这个函数，只发射与发射数据项最少的那个 Observable 一样多的数据。

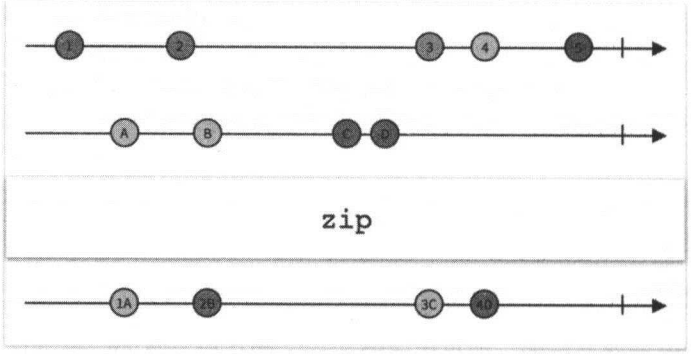


图 7-3

zip 的最后一个参数接收每个 Observable 发射的一项数据，返回被压缩后的数据，它可以接收 1~9 个参数：一个 Observable 序列，或者一些发射 Observable 的 Observable。

示例代码：

```
        Observable<Integer> odds = Observable.just(1, 3, 5);
        Observable<Integer> evens = Observable.just(2, 4, 6);

        Observable.zip(odds, evens, new BiFunction<Integer, Integer, Integer>()
{


            @Override
            public Integer apply(Integer integer1, Integer integer2) throws
Exception {
                return integer1 + integer2;
            }
        }).subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println("Next: " + integer);
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        System.err.println("Error: " + throwable.getMessage());
                    }
                }, new Action() {
```

```
                @Override
                public void run() throws Exception {
                    System.out.println("Sequence complete.");
                }
            });
```

执行结果：

```
Next: 3
Next: 7
Next: 11
Sequence complete.
```

可以看到这段代码是对两个 Observable 发射的数据进行相加，然后再打印出来。

对上面的代码稍微修改一下，其中 odds 多了两个数据。

```
    Observable<Integer> odds = Observable.just(1, 3, 5,7,9);
     Observable<Integer> evens = Observable.just(2, 4, 6);

     Observable.zip(odds, evens, new BiFunction<Integer, Integer, Integer>()
{


        @Override
        public Integer apply(Integer integer1, Integer integer2) throws
Exception {
            return integer1 + integer2;
        }
    }).subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println("Next: " + integer);
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.err.println("Error: " + throwable.getMessage());
                }
            }, new Action() {
                @Override
                public void run() throws Exception {
```

```
                System.out.println("Sequence complete.");
            }
        });
```

执行结果：

```
Next: 3
Next: 7
Next: 11
Sequence complete.
```

细心的读者会看到，代码改动之后执行结果并没有变。因为 zip 只发射与发射数据项最少的那个 Observable 一样多的数据。zip 操作符相对于 merge 操作符，除发射数据外，还会进行合并操作，而且 zip 发射的数据与数据项最少的 Observable 有关。

还有一点需要注意的是，这里 BiFunction 相当于一个合并函数，并不一定要返回 Integer 类型，可以根据业务需要返回合适的类型。BiFunction 的源码如下：

```
import io.reactivex.annotations.NonNull;

/**
 * A functional interface (callback) that computes a value based on multiple
input values.
 * @param <T1> the first value type
 * @param <T2> the second value type
 * @param <R> the result type
 */
public interface BiFunction<T1, T2, R> {

    /**
     * Calculate a value based on the input values.
     * @param t1 the first value
     * @param t2 the second value
     * @return the result value
     * @throws Exception on error
     */
    @NonNull
    R apply(@NonNull T1 t1, @NonNull T2 t2) throws Exception;
}
```

RxJava 2.x 中 FuncN 遵循 Java 8 的命名规则。相对于 RxJava1.x，Func 改名成 Function，Func2

改名成 BiFunction，Func3 ~ Func9 改名成 Function3 ~ Function9，FuncN 由 Function 取代。

7.2　combineLatest 和 join

1. combineLatest 操作符

combineLatest 操作符的行为类似于 zip，但是只有当原始的 Observable 中的每一个都发射了一条数据时 zip 才发射数据，而 combineLatest 则是当原始的 Observable 中任意一个发射了数据时就发射一条数据。当原始 Observables 的任何一个发射了一条数据时，combineLatest 使用一个函数结合它们最近发射的数据，然后发射这个函数的返回值。

```
        Observable<Integer> odds = Observable.just(1, 3, 5);
        Observable<Integer> evens = Observable.just(2, 4, 6);

        Observable.combineLatest(odds, evens, new BiFunction<Integer, Integer,
Integer>() {


            @Override
            public Integer apply(Integer integer1, Integer integer2) throws
Exception {
                return integer1 + integer2;
            }
        }).subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println("Next: " + integer);
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(Throwable throwable) throws Exception {
                        System.err.println("Error: " + throwable.getMessage());
                    }
                }, new Action() {
                    @Override
                    public void run() throws Exception {
                        System.out.println("Sequence complete.");
                    }
                });
```

执行结果：

```
Next: 7
Next: 9
Next: 11
Sequence complete.
```

除此之外，本书的 11.2 节介绍了如何使用 combineLatest 来做表单验证。

2. join 操作符

join 操作符结合两个 Observable 发射的数据，基于时间窗口（针对每条数据特定的原则）选择待集合的数据项。将这些时间窗口实现为一些 Observables，它们的生命周期从任何一条 Observable 发射的每一条数据开始。当这个定义时间窗口的 Observable 发射了一条数据或者完成时，与这条数据关联的窗口也会关闭。只要这条数据的窗口是打开的，它就继续结合其他 Observable 发射的任何数据项。

示例代码：

```
        Observable<Integer> o1 = Observable.just(1, 2, 3);
        Observable<Integer> o2 = Observable.just(4, 5, 6);

        o1.join(o2, new Function<Integer, ObservableSource<String>>() {

            @Override
            public ObservableSource<String> apply(@NonNull Integer integer)
throws Exception {
                return Observable.just(String.valueOf(integer)).delay(200,
TimeUnit.MILLISECONDS);
            }
        }, new Function<Integer, ObservableSource<String>>() {

            @Override
            public ObservableSource<String> apply(@NonNull Integer integer)
throws Exception {
                return Observable.just(String.valueOf(integer)).delay(200,
TimeUnit.MILLISECONDS);
            }
        }, new BiFunction<Integer, Integer, String>() {
```

```
        @Override
        public String apply(@NonNull Integer integer, @NonNull Integer
integer2) throws Exception {
            return integer + ":" + integer2;
        }
    }).subscribe(new Consumer<String>() {
        @Override
        public void accept(String s) throws Exception {
            System.out.println("onNext " + s);
        }
    });

    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

执行结果：

```
onNext 1:4
onNext 2:4
onNext 3:4
onNext 1:5
onNext 2:5
onNext 3:5
onNext 1:6
onNext 2:6
onNext 3:6
```

join(Observable, Function, Function, BiFunction) 有四个参数，下面分别解释它们的用途。

- ◎ Observable：源 Observable 需要组合的 Observable，这里可以称之为目标 Observable。
- ◎ Function：接收从源 Observable 发射来的数据，并返回一个 Observable，这个 Observable 的生命周期决定了源 Observable 发射数据的有效期。
- ◎ Function：接收目标 Observable 发射的数据，并返回一个 Observable，这个 Observable 的生命周期决定了目标 Observable 发射数据的有效期。
- ◎ BiFunction：接收从源 Observable 和目标 Observable 发射的数据，并将这两个数据组合后返回。

join 操作符的效果类似于排列组合，把第一个数据源 A 作为基座窗口，它根据自己的节奏不断发射数据元素；第二个数据源 B，每发射一个数据，我们都把它和第一个数据源 A 中已经发射的数据进行一对一匹配。举例来说，如果某一时刻 B 发射了一个数据“B”，此时 A 已经发射了 a,b,c,d 共 4 个数据，那么合并操作就是把“B”依次与 a,b,c,d 配对，得到 4 组数据：[a, B]、[b, B]、[c, B]、[d, B]。

对上面的代码做一下修改：

```
        Observable<Integer> o1 = Observable.just(1, 2, 3).delay(200,
TimeUnit.MILLISECONDS);
        Observable<Integer> o2 = Observable.just(4, 5, 6);

        o1.join(o2, new Function<Integer, ObservableSource<String>>() {

            @Override
            public ObservableSource<String> apply(@NonNull Integer integer)
throws Exception {
                return Observable.just(String.valueOf(integer)).delay(200,
TimeUnit.MILLISECONDS);
            }
        }, new Function<Integer, ObservableSource<String>>() {

            @Override
            public ObservableSource<String> apply(@NonNull Integer integer)
throws Exception {
                return Observable.just(String.valueOf(integer)).delay(200,
TimeUnit.MILLISECONDS);
            }
        }, new BiFunction<Integer, Integer, String>() {

            @Override
            public String apply(@NonNull Integer integer, @NonNull Integer
integer2) throws Exception {
                return integer + ":" + integer2;
            }
        }).subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                System.out.println("onNext " + s);
            }
```

```
        });

        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果：

```
onNext 1:4
onNext 1:5
onNext 1:6
onNext 2:4
onNext 2:5
onNext 2:6
onNext 3:4
onNext 3:5
onNext 3:6
```

7.3 startWith

在数据序列的开头插入一条指定的项，如图 7-4 所示。

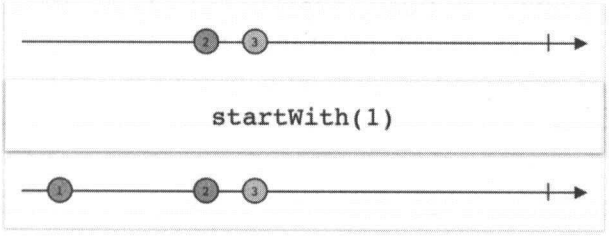


图 7-4

如果想让一个Observable在发射数据之前先发射一个指定的数据序列，则可以使用startWith操作符。如果想在一个 Observable 发射数据的末尾追加一个数据序列，则可以使用 concat 操作符。

示例代码：

```
    Observable.just("Hello Java", "Hello Kotlin","Hello Scala")
```

```
                .startWith("Hello Rx")
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String s) throws Exception {
                        System.out.println(s);
                    }
                });
```

执行结果：

```
Hello Rx
Hello Java
Hello Kotlin
Hello Scala
```

可以看到由于调用了 startWith 操作符，因此 Hello Rx 在执行结果中排在了最前面。

startWith 操作符支持传递 Iterable，同时还有一个 startWithArray 的操作符。

示例代码：

```
        Observable.just("Hello Java", "Hello Kotlin","Hello Scala")
                .startWithArray("Hello Groovy","Hello Clojure")
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String s) throws Exception {
                        System.out.println(s);
                    }
                });
```

执行结果：

```
Hello Groovy
Hello Clojure
Hello Java
Hello Kotlin
Hello Scala
```

使用了 startWithArray 操作符之后，可以再使用 startWith 操作符。

```
        Observable.just("Hello Java", "Hello Kotlin","Hello Scala")
                .startWithArray("Hello Groovy","Hello Clojure")
```

```
            .startWith("Hello Rx")
            .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });
```

执行结果：

```
Hello Rx
Hello Groovy
Hello Clojure
Hello Java
Hello Kotlin
Hello Scala
```

可以看到即使多次使用 startWith 操作符，最后发射的数据也永远排在最前面。

startWith 还可以传递一个 Observable 对象，它会将那个 Observable 的发射物插在原始 Observable 发射的数据序列之前，然后把这个当作自己的发射物集合。

```
    Observable.just("Hello Java", "Hello Kotlin","Hello Scala")
            .startWithArray("Hello Groovy","Hello Clojure")
            .startWith(Observable.just("Hello Rx"))
            .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });
```

执行结果：

```
Hello Rx
Hello Groovy
Hello Clojure
Hello Java
Hello Kotlin
Hello Scala
```

7.4 connect、push 和 refCount

在本书 2.2 小节，曾经讲解过 connect、push、refCount 这三个操作符。其中，connect 和 refCount 是 ConnectableObservable 所使用的操作符。

connectableObservable 继承自 Observable，然而它并不是在调用 subscribe()的时候发射数据，而是只有对其使用 connect 操作符时它才会发射数据，所以可以用来更灵活地控制数据发射的时机。另外，connectableObservable 是 Hot Observable。

push 操作符是将普通的 Observable 转换成 connectableObservable。

connect 操作符是用来触发 connectableObservable 发射数据的。我们可以等所有的观察者们都订阅了 connectableObservable 之后再发射数据，如图 7-5 所示。

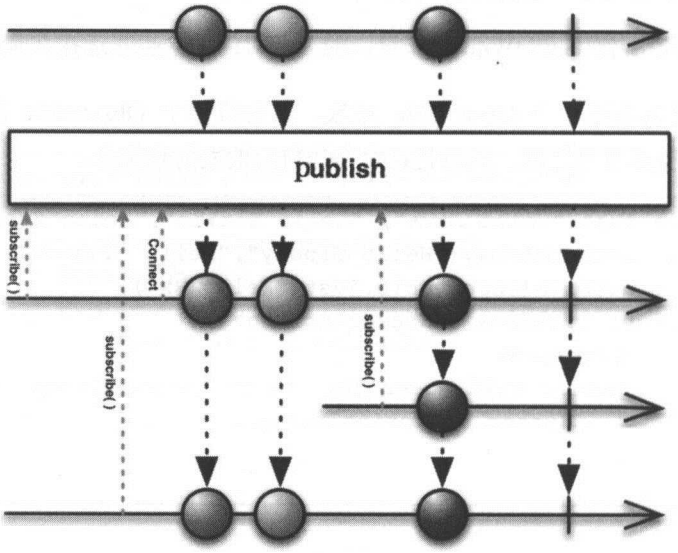


图 7-5

示例代码：

```
SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");

Observable<Long> obs = Observable.interval(1,
TimeUnit.SECONDS).take(6);
```

```
        ConnectableObservable<Long> connectableObservable = obs.publish();

        connectableObservable.subscribe(new Observer<Long>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(Long aLong) {
                System.out.println("subscriber1: onNext:"+aLong+"->time:"+
sdf.format(new Date()));
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("subscriber1: onError");
            }

            @Override
            public void onComplete() {
                System.out.println("subscriber1: onComplete");
            }
        });
        connectableObservable.delaySubscription(3, TimeUnit.SECONDS)
                .subscribe(new Observer<Long>() {
                    @Override
                    public void onSubscribe(Disposable d) {

                    }

                    @Override
                    public void onNext(Long aLong) {
                        System.out.println("subscriber2:
onNext:"+aLong+"->time:"+ sdf.format(new Date()));
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.out.println("subscriber2: onError");
                    }
```

```
                @Override
                public void onComplete() {
                    System.out.println("subscriber2: onComplete");
                }
            });

    connectableObservable.connect();

    try {
        Thread.sleep(15000);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
```

执行结果：

```
subscriber1: onNext:0->time:04:47:57
subscriber1: onNext:1->time:04:47:58
subscriber1: onNext:2->time:04:47:59
subscriber2: onNext:2->time:04:47:59
subscriber1: onNext:3->time:04:48:00
subscriber2: onNext:3->time:04:48:00
subscriber1: onNext:4->time:04:48:01
subscriber2: onNext:4->time:04:48:01
subscriber1: onNext:5->time:04:48:02
subscriber2: onNext:5->time:04:48:02
subscriber1: onComplete
subscriber2: onComplete
```

refCount 操作符是将 connectableObservable 转换成普通的 Observable，同时又保持了 Hot Observable 的特性。当出现第一个订阅者时，refCount 会调用 connect()。每个订阅者每次都会接收到同样的数据，但是当所有订阅者都取消订阅（dispose）时，refCount 会自动 dipose 上游的 Observable，如图 7-6 所示。

所有的订阅者都取消订阅后，则数据流停止。如果重新订阅则数据流重新开始。如果不是所有的订阅者都取消了订阅，而是只取消了部分。则部分订阅者/观察者重新开始订阅时，不会从头开始数据流。

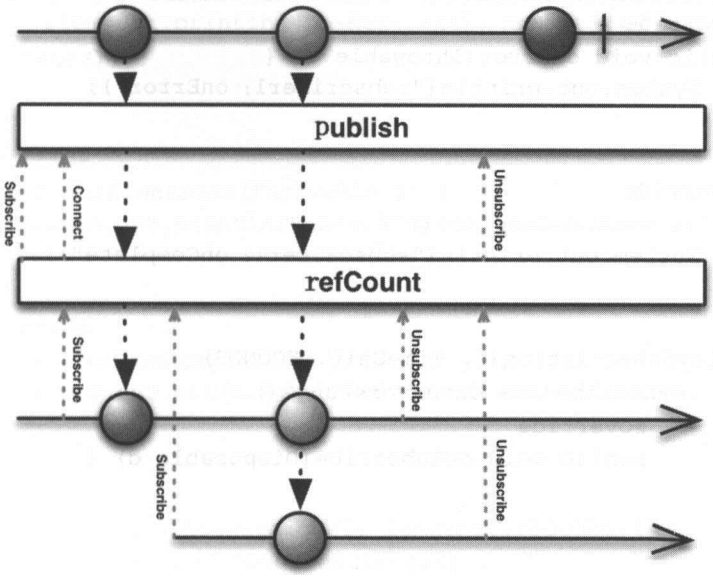


图 7-6

示例代码：

```
        SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");

        Observable<Long> obs = Observable.interval(1,
TimeUnit.SECONDS).take(6);

        ConnectableObservable<Long> connectableObservable = obs.publish();
        Observable obsRefCount = connectableObservable.refCount();

        obs.subscribe(new Observer<Long>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(Long aLong) {
                System.out.println("subscriber1: onNext:"+aLong+"->time:"+
sdf.format(new Date()));
            }
```

```
            @Override
            public void onError(Throwable e) {
                System.out.println("subscriber1: onError");
            }

            @Override
            public void onComplete() {
                System.out.println("subscriber1: onComplete");
            }
        });
        obs.delaySubscription(3, TimeUnit.SECONDS)
                .subscribe(new Observer<Long>() {
                    @Override
                    public void onSubscribe(Disposable d) {

                    }

                    @Override
                    public void onNext(Long aLong) {
                        System.out.println("subscriber2:
onNext:"+aLong+"->time:"+ sdf.format(new Date()));
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.out.println("subscriber2: onError");
                    }

                    @Override
                    public void onComplete() {
                        System.out.println("subscriber2: onComplete");
                    }
                });

        obsRefCount.subscribe(new Observer<Long>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(Long aLong) {
```

```
                System.out.println("obsRefCount1: onNext:"+aLong+"->time:"+
sdf.format(new Date()));
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("obsRefCount1: onError");
            }

            @Override
            public void onComplete() {
                System.out.println("obsRefCount1: onComplete");
            }
        });

        obsRefCount.delaySubscription(3, TimeUnit.SECONDS)
                .subscribe(new Observer<Long>() {
                    @Override
                    public void onSubscribe(Disposable d) {

                    }

                    @Override
                    public void onNext(Long aLong) {
                        System.out.println("obsRefCount2:
onNext:"+aLong+"->time:"+ sdf.format(new Date()));
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.out.println("obsRefCount2: onError");
                    }

                    @Override
                    public void onComplete() {
                        System.out.println("obsRefCount2: onComplete");
                    }
                });

        try {
            Thread.sleep(15000);
        } catch (InterruptedException e) {
```

```
            e.printStackTrace();
        }
```

执行结果：

```
subscriber1: onNext:0->time:04:57:20
obsRefCount1: onNext:0->time:04:57:20
subscriber1: onNext:1->time:04:57:21
obsRefCount1: onNext:1->time:04:57:21
subscriber1: onNext:2->time:04:57:22
obsRefCount1: onNext:2->time:04:57:22
subscriber1: onNext:3->time:04:57:23
obsRefCount1: onNext:3->time:04:57:23
subscriber2: onNext:0->time:04:57:23
obsRefCount2: onNext:3->time:04:57:23
subscriber1: onNext:4->time:04:57:24
obsRefCount1: onNext:4->time:04:57:24
subscriber2: onNext:1->time:04:57:24
obsRefCount2: onNext:4->time:04:57:24
subscriber1: onNext:5->time:04:57:25
subscriber1: onComplete
subscriber2: onNext:2->time:04:57:25
obsRefCount1: onNext:5->time:04:57:25
obsRefCount2: onNext:5->time:04:57:25
obsRefCount1: onComplete
obsRefCount2: onComplete
subscriber2: onNext:3->time:04:57:26
subscriber2: onNext:4->time:04:57:27
subscriber2: onNext:5->time:04:57:28
subscriber2: onComplete
```

7.5 replay

保证所有的观察者收到相同的数据序列，即使它们在 Observable 开始发射数据之后才订阅，如图 7-7 所示。

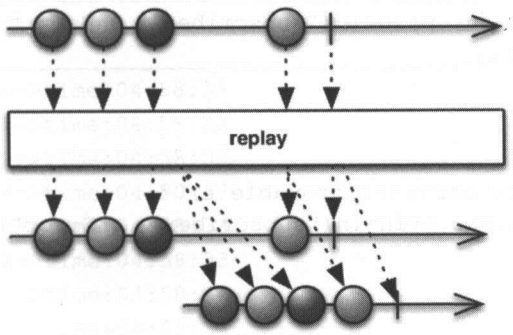


图 7-7

replay 操作符返回一个 connectableObservable 对象，并且可以缓存发射过的数据，这样即使有订阅者在发射数据之后进行订阅，也能收到之前发射过的数据。不过使用 replay 操作符最好还是先限定缓存的大小，否则缓存的数据太多时会占用很大一块内存。对缓存的控制可以从空间和时间两个方面来实现。

replay 操作符生成的 connectableObservable，使得观察者无论什么时候开始订阅，都能收到 Observable 发送的所有数据。

示例代码：

```
    SimpleDateFormat sdf = new SimpleDateFormat("HH:mm:ss");

    Observable<Long> obs = Observable.interval(1,
TimeUnit.SECONDS).take(6);

    ConnectableObservable<Long> connectableObservable = obs.replay();

    connectableObservable.connect();

    connectableObservable.subscribe(new Observer<Long>() {
        @Override
        public void onSubscribe(Disposable d) {

        }

        @Override
        public void onNext(Long aLong) {
```

```
                System.out.println("subscriber1: onNext:"+aLong+"->time:"+
sdf.format(new Date()));
            }

            @Override
            public void onError(Throwable e) {
                System.out.println("subscriber1: onError");
            }

            @Override
            public void onComplete() {
                System.out.println("subscriber1: onComplete");
            }
        });
        connectableObservable.delaySubscription(3, TimeUnit.SECONDS)
                .subscribe(new Observer<Long>() {
                    @Override
                    public void onSubscribe(Disposable d) {

                    }

                    @Override
                    public void onNext(Long aLong) {
                        System.out.println("subscriber2:
onNext:"+aLong+"->time:"+ sdf.format(new Date()));
                    }

                    @Override
                    public void onError(Throwable e) {
                        System.out.println("subscriber2: onError");
                    }

                    @Override
                    public void onComplete() {
                        System.out.println("subscriber2: onComplete");
                    }
                });
        try {
            Thread.sleep(15000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
```

执行结果：

```
subscriber1: onNext:0->time:04:58:31
subscriber1: onNext:1->time:04:58:32
subscriber1: onNext:2->time:04:58:33
subscriber2: onNext:0->time:04:58:33
subscriber2: onNext:1->time:04:58:33
subscriber2: onNext:2->time:04:58:33
subscriber1: onNext:3->time:04:58:34
subscriber2: onNext:3->time:04:58:34
subscriber1: onNext:4->time:04:58:35
subscriber2: onNext:4->time:04:58:35
subscriber1: onNext:5->time:04:58:36
subscriber2: onNext:5->time:04:58:36
subscriber1: onComplete
subscriber2: onComplete
```

这段代码中，connect()无须在观察者订阅之后调用也能执行。

replay 有多个接收不同参数的重载方法，有的可以指定 replay 的最大缓存数量，有的可以指定调度器。

connectableObservable 的线程切换只能通过 replay 操作符实现，普通 Observable 的 subscribeOn()和 observerOn()在 ConnectableObservable 中不起作用。replay 操作符可以通过指定线程的方式来切换线程。

7.6　小结

RxJava 的合并操作能够同时处理多个被观察者，并发送相应的事件。

对于连接操作符，有一个很重要的概念 connectableObservable。可连接的 Observable 在被订阅时并不发射数据，只有在它的 connect()被调用时才开始发射数据。

第8章 RxJava的背压

8.1 背压

在RxJava中，会遇到被观察者发送消息太快以至于它的操作符或者订阅者不能及时处理相关的消息，这就是典型的背压（Back Pressure）场景。

在RxJava官方的维基百科中关于Back Pressure是这样描述的：

In RxJava it is not difficult to get into a situation in which an Observable is emitting items more rapidly than an operator or subscriber can consume them. This presents the problem of what to do with such a growing backlog of unconsumed items.

Back Pressure经常被翻译为背压，背压的字面意思比较晦涩，难以理解。它是指在异步场景下，被观察者发送事件速度远快于观察者处理的速度，从而导致下游的buffer溢出，这种现象叫作背压。

首先，背压必须是在异步的场景下才会出现，即被观察者和观察者处于不同的线程中。

其次，RxJava是基于Push模型的。对于Pull模型而言，当消费者请求数据的时候，如果生产者比较慢，则消费者会阻塞等待。如果生产者比较快，则生产者会等待消费者处理完后再生产新的数据，所以不会出现背压的情况。然而在RxJava中，只要生产者数据准备好了就会发射出去。如果生产者比较慢，则消费者会等待新的数据到来。如果生产者比较快，则会有很多数据发射给消费者，而不管消费者当前有没有能力处理数据，这样就会导致背压。

最后，在 RxJava 2.x 中，只有新增的 Flowable 类型是支持背压的，并且 Flowable 很多操作符内部都使用了背压策略，从而避免过多的数据填满内部的队列。

在 RxJava 1.x 中，有很多事件因为不能被正确地背压，从而抛出 MissingBackpressureException。在 RxJava 1.x 中的 observeOn，由于切换了观察者的线程，因此内部实现用队列存储事件。

先来看一段代码

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0;; i++) {   //无限循环发事件
            subscriber.onNext(i);
        }
    }
}).subscribe(new Action1<Integer>() {
    @Override
    public void call(Integer integer) {
        Log.d("NoBackpressureActivity", String.valueOf(integer));
    }
});
```

这段代码其实并不会产生背压，只会出现 ANR（application not responding），因为被观察者和订阅者处在同一个线程中，只有二者不在同一个线程时，上述代码才会出现背压。

我们对上述代码做一些修改，切换一下线程。

```
Observable.create(new Observable.OnSubscribe<Integer>() {
    @Override
    public void call(Subscriber<? super Integer> subscriber) {
        for (int i = 0; ; i++) {   //无限循环发事件
            subscriber.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Action1<Integer>() {
            @Override
            public void call(Integer integer) {
```

```
                    Log.d("Backpressure1Activity",
String.valueOf(integer));
                }
            });
```

修改完之后立即引起了 App Crash，查看日志之后发现，出现了如下的 Exception。

```
Caused by: rx.exceptions.MissingBackpressureException
```

在 RxJava 1.x 中，Observable 是支持背压的，从 Observable 的源码中可以看到，在 RxJava 1.x 中的 Buffer 的大小只有 16。

```
    /**
     * Returns an Observable that emits non-overlapping buffered items from the
source Observable each time the
     * specified boundary Observable emits an item.
     * <p>
     * <img width="640" height="395"
src="https://raw.github.com/wiki/ReactiveX/RxJava/images/rx-operators/buffe
r8.png" alt="">
     * <p>
     * Completion of either the source or the boundary Observable causes the
returned Observable to emit the
     * latest buffer and complete.
     * <dl>
     *  <dt><b>Backpressure:</b></dt>
     *  <dd>This operator does not support backpressure as it is instead
controlled by the {@code Observable}
     *      {@code boundary} and buffers data. It requests {@code Long.MAX_VALUE}
upstream and does not obey
     *      downstream requests.</dd>
     *  <dt><b>Scheduler:</b></dt>
     *  <dd>This version of {@code buffer} does not operate by default on a
particular {@link Scheduler}.</dd>
     * </dl>
     *
     * @param <B>
     *            the boundary value type (ignored)
     * @param boundary
     *            the boundary Observable
     * @return an Observable that emits buffered items from the source Observable
when the boundary Observable
```

```
     *         emits an item
     * @see #buffer(rx.Observable, int)
     * @see <a
href="http://reactivex.io/documentation/operators/buffer.html">ReactiveX
operators documentation: Buffer</a>
     */
    public final <B> Observable<List<T>> buffer(Observable<B> boundary) {
        return buffer(boundary, 16);
    }
```

也就是说，刚才的代码无须发无限次，只要发 17 次就可以引起异常。下面的代码将原先的无数次改成了 17 次。

```
        Observable.create(new Observable.OnSubscribe<Integer>() {
            @Override
            public void call(Subscriber<? super Integer> subscriber) {
                for (int i = 0; i < 17; i++) {
                    subscriber.onNext(i);
                }
            }
        }).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Action1<Integer>() {
                    @Override
                    public void call(Integer integer) {
                        Log.d("Backpressure2Activity",
String.valueOf(integer));
                    }
                });
```

果然可以抛出 MissingBackpressureException，符合预期。如果把 17 改成 16，则程序可以正常运行，打印出 0-15。

在 RxJava 1.x 中，不是所有的 Observable 都支持背压。我们知道 Observable 有 Hot 和 Cold 之分，在 2.2 节曾经讲过，如果遗忘的话则可以回过去看一下它们的概念。RxJava 1.x 中，Hot Observables 是不支持背压的，而 Cold Observables 中也有一部分不支持背压。

在 RxJava 1.x 中如何解决背压问题呢？

◎　过滤限流，通过使用限流操作符将被观察者产生的大部分事件过滤并抛弃，以达到限

流的目的，间接降低事件发射的速度，例如使用以下操作符。

sample：在一段时间内，只处理最后一个数据。

throttleFirst：在一段时间内，只处理第一个数据。

debounce：发送一个数据，开始计时，到了规定时间，若没有再发送数据，则开始处理数据，反之重新开始计时。

◎ 打包缓存，在被观察者发射事件过快导致观察者来不及处理的情况下，可以使用缓存类的操作符将其中一部分打包缓存起来，再一点一点的处理其中的事件。

buffer：将多个事件打包放入一个 List 中，再一起发射。

window：将多个事件打包放入一个 Observable 中，再一起发射。

◎ 使用背压操作符，我们可以通过一些操作符来转化成支持背压的 Observable。这些操作符包括：

onBackpressureBuffer

onBackpressureDrop

onBackpressureLatest

onBackpressureBlock（已过期）

在 RxJava 1.x 中，背压的设计并不十分完美。它的缓存池很小，只有 16，不能处理较大量的并发事件。RxJava 1.x 上游（被观察者）无法得知下游（观察者）对事件的处理能力和事件处理进度，只能把事件一次性抛给下游。接下来介绍 RxJava 2.x 的背压策略。

8.2 RxJava 2.x 的背压策略

在 RxJava 2.x 中，Observable 不再支持背压，而是改用 Flowable 来专门支持背压。默认队列大小为 128，并且要求所有的操作符强制支持背压。

从 BackpressureStrategy 的源码可以看到，Flowable 一共有 5 种背压策略。

```
/**
```

```
 * Represents the options for applying backpressure to a source sequence.
 */
public enum BackpressureStrategy {
    /**
     * OnNext events are written without any buffering or dropping.
     * Downstream has to deal with any overflow.
     * <p>Useful when one applies one of the custom-parameter onBackpressureXXX
operators.
     */
    MISSING,
    /**
     * Signals a MissingBackpressureException in case the downstream can't keep
up.
     */
    ERROR,
    /**
     * Buffers <em>all</em> onNext values until the downstream consumes it.
     */
    BUFFER,
    /**
     * Drops the most recent onNext value if the downstream can't keep up.
     */
    DROP,
    /**
     * Keeps only the latest onNext value, overwriting any previous value if the
     * downstream can't keep up.
     */
    LATEST
}
```

1. MISSING

此策略表示，通过 Create 方法创建的 Flowable 没有指定背压策略，不会对通过 OnNext 发射的数据做缓存或丢弃处理，需要下游通过背压操作符（onBackpressureBuffer()/onBackpressureDrop()/onBackpressureLatest()）指定背压策略。

2. ERROR

此策略表示，如果放入 Flowable 的异步缓存池中的数据超限了，则会抛出 MissingBackpressureException 异常。

```
Flowable.create(new FlowableOnSubscribe<Integer>() {

    @Override
    public void subscribe(FlowableEmitter<Integer> e) throws Exception
{
        for (int i = 0; i<129; i++) {
            e.onNext(i);
        }
    }
}, BackpressureStrategy.ERROR)
        .subscribeOn(Schedulers.newThread())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                System.out.println(integer);
            }
        });
```

在 Android 中运行这段代码，会立刻引起 App Crash，查看 LogCat 之后发现，发现如下的 Exception。

Caused by: io.reactivex.exceptions.MissingBackpressureException: create: could not emit value due to lack of requests

因为 Flowable 的默认队列是 128，所以将上述代码的 129 改成 128，程序就可以正常运行了。

3. BUFFER

此策略表示，Flowable 的异步缓存池同 Observable 的一样，没有固定大小，可以无限制添加数据，不会抛出 MissingBackpressureException 异常，但会导致 OOM（Out of Memory）。

```
Flowable.create(new FlowableOnSubscribe<Integer>() {

    @Override
    public void subscribe(FlowableEmitter<Integer> e) throws Exception
{
        for (int i = 0;; i++) {
            e.onNext(i);
```

```
            }
        }
    }, BackpressureStrategy.BUFFER)
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println(integer);
                }
            });
```

上述这段代码如果在 Android 中运行的话只会引起 ANR。

4. DROP

此策略表示，如果 Flowable 的异步缓存池满了，则会丢掉将要放入缓存池中的数据。

```
    Flowable.create(new FlowableOnSubscribe<Integer>() {

        @Override
        public void subscribe(FlowableEmitter<Integer> e) throws Exception
{
            for (int i = 0; i<129; i++) {
                e.onNext(i);
            }
        }
    }, BackpressureStrategy.DROP)
            .subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    System.out.println(integer);
                }
            });
```

在 Android 中运行这段代码，不会引起 Crash，但只会打印出 0~127，第 128 则被丢弃，因为 Flowable 的内部队列已经满了。

5. LATEST

此策略表示，如果缓存池满了，会丢掉将要放入缓存池中的数据。这一点与 DROP 策略一样，不同的是，不管缓存池的状态如何，LATEST 策略会将最后一条数据强行放入缓存池中。

```
        Flowable.create(new FlowableOnSubscribe<Integer>() {

            @Override
            public void subscribe(FlowableEmitter<Integer> e) throws Exception
{
                for (int i = 0; i<1000; i++) {
                    e.onNext(i);
                }
            }
        }, BackpressureStrategy.LATEST)
                .subscribeOn(Schedulers.newThread())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        System.out.println(integer);
                    }
                });
```

在 Android 中运行这段代码，也不会引起 Crash，并且会打印出 0-127 以及 999。因为 999 是最后一条数据。

Flowable 不仅可以通过 create 创建时需要指定背压策略，还可以在通过其他创建操作符，例如 just、fromArray 等创建后通过背压操作符指定背压策略。例如，onBackpressureBuffer()对应 BackpressureStrategy.BUFFER，onBackpressureDrop() 对应 BackpressureStrategy.DROP，onBackpressureLatest()对应 BackpressureStrategy.LATEST。

示例代码：

```
Flowable.interval(period, TimeUnit.MILLISECONDS)
.onBackpressureBuffer()
.subscribe(new Consumer<Long>() {
    @Override
    public void accept(Long aLong) throws Exception {
```

```
        do something ……

    }
})
```

8.3　小结

背压（Back Pressure）解决了上下游速度不一致的问题。本章讲述了 RxJava 2.x 的背压策略。RxJava 2.x 相比于 RxJava 1.x，在背压策略上有了很大的提升，但是使用背压不一定就能得到我们期待的结果，因此还是应从需求出发，结合实际情况制定属于自己的策略。

第9章
Disposable和Transformer的使用

9.1 Disposable

在 RxJava 1.x 中，Subscription 的接口可以用来取消订阅。

```
public interface Subscription {
    /**
     * Stops the receipt of notifications on the {@link Subscriber} that was
registered when this Subscription
     * was received.
     * <p>
     * This allows unregistering an {@link Subscriber} before it has finished
receiving all events (i.e. before
     * onCompleted is called).
     */
    void unsubscribe();

    /**
     * Indicates whether this {@code Subscription} is currently unsubscribed.
     *
     * @return {@code true} if this {@code Subscription} is currently unsubscribed,
{@code false} otherwise
     */
    boolean isUnsubscribed();
}
```

在 RxJava 1.x 中，Observable.subscribe()方法会返回一个 Subscription 的对象。也就是说，

我们每次订阅都会返回 Subscription。Subscription 只需调用 unsubscribe 就可以取消订阅。

```
Subscription subscription = Observable.just("Hello, World!")
    .subscribe(s -> System.out.println(s));

subscription.unsubscribe();//解除订阅关系
```

Subscription 对象是被观察者和订阅者之间的纽带。RxJava 使用 Subscription 处理取消订阅时，会停止整个调用链。如果使用了一串很复杂的操作符，则调用 unsubscribe()将会在它当前执行的地方终止，而不需要做任何额外的工作。

在 Android 开发中，我们会在 Activity / Fragment 的 onDestroy 里做一些释放资源的事情，如果使用了 RxJava 1.x，则可以用 Subscription.isUnsubscribed()检查 Subscription 是否是 unsubscribed。如果是 unsubscribed，则调用 Subscription.unsubscribe()，RxJava 会取消订阅并通知给 Subscriber，并允许垃圾回收机制释放对象，防止任何 RxJava 造成内存泄露。

在 RxJava 2.0 之后，Subscription 被改名为 Disposable。RxJava 2.x 中由于已经存在名为 org.reactivestreams.subscription 这个类（遵循 Reactive Streams 标准），为了避免名字冲突，所以将原先的 rx.Subscription 改名为 io.reactivex.disposables.Disposable。

```
/**
 * Represents a disposable resource.
 */
public interface Disposable {
    /**
     * Dispose the resource, the operation should be idempotent.
     */
    void dispose();

    /**
     * Returns true if this resource has been disposed.
     * @return true if this resource has been disposed
     */
    boolean isDisposed();
}
```

示例代码：

```
Disposable disposable = Observable.just("Hello, World!")
```

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

disposable.dispose();
```

CompositeDisposable

RxJava 1.x 中有一个复合订阅（composite subscription）的概念。在 RxJava 2.x 中，RxJava 也内置了一个类似复合订阅的容器 CompositeDisposable，每当我们得到一个 Disposable 时，就调用 CompositeDisposable.add()，将它添加到容器中，在退出的时候，调用 CompositeDisposable.clear()即可切断所有的事件。在 Android 中，我们经常会看到这样的用法，也就是会有一个针对 Activity 或者 Fragment 的 CompositeDisposable，这样就可以在 onDestroy 或者其他合适的地方取消订阅。

9.2 RxLifecycle 和 AutoDispose

在 Android 开发中，可以使用 Disposable 来管理一个订阅或者使用 CompositeDisposable 来管理多个订阅，防止由于没有及时取消，导致 Activity/Fragment 无法销毁而引起内存泄露。然而，也有一些比较成熟的库可以做这些事情。

1. RxLifecycle

GitHub 下载地址：https://github.com/trello/RxLifecycle

RxLifecycle 是配合 Activity/Fragment 生命周期来管理订阅的。由于 RxJava Observable 订阅后（调用了 subscribe 函数），一般会在后台线程执行一些操作（比如访问网络请求数据），当后台操作返回后，调用 Observer 的 onNext 等函数，然后再更新 UI 状态。但是后台线程请求是需要时间的，如果用户单击刷新按钮请求新的微博信息，在刷新还没有完成时，如果用户退出了当前界面，返回了前面的界面，而这个时候刷新的 Observable 又未取消订阅，就会导致之前的 Activity 无法被 JVM 回收，从而导致内存泄露。这就是在 Android 开发中值得注意的地方，RxLifecycle 就是专门用来做这件事的。

```
compile 'com.trello.rxlifecycle2:rxlifecycle:2.2.0'

// If you want to bind to Android-specific lifecycles
compile 'com.trello.rxlifecycle2:rxlifecycle-android:2.2.0'
```

```
// If you want pre-written Activities and Fragments you can subclass as providers
compile 'com.trello.rxlifecycle2:rxlifecycle-components:2.2.0'

// If you want pre-written support preference Fragments you can subclass as
providers
compile 'com.trello.rxlifecycle2:rxlifecycle-components-preference:2.2.0'

// If you want to use Navi for providers
compile 'com.trello.rxlifecycle2:rxlifecycle-navi:2.2.0'

// If you want to use Android Lifecycle for providers
compile 'com.trello.rxlifecycle2:rxlifecycle-android-lifecycle:2.2.0'

// If you want to use Kotlin syntax
compile 'com.trello.rxlifecycle2:rxlifecycle-kotlin:2.2.0'

// If you want to use Kotlin syntax with Android Lifecycle
compile 'com.trello.rxlifecycle2:rxlifecycle-android-lifecycle-kotlin:2.2.0'
```

在想使用该库的 Activity 或 Fragment 中，可以继承 RxActivity、RxAppCompatActivity 或 RxFragment，并添加相应的 import 语句，例如：

```
import com.trello.rxlifecycle2.components.support.RxAppCompatActivity;

...

public class MainActivity extends RxAppCompatActivity {
```

当涉及绑定 Observable 到 Activity 或 Fragment 的生命周期时，要么指定 Observable 应终止的生命周期事件，要么让 RxLifecycle 库决定何时终止 Observable 序列。

默认情况下，RxLifecycle 将在辅助生命周期事件中终止 Observable，所以如果在 Activity 的 OnCreate()方法期间订阅了 Observable，则 RxLifecycle 将在该 Activity 的 onDestroy()方法中终止 Observable 序列。如果在 Fragment 的 onAttach()方法中订阅，那么 RxLifecycle 将在 onDetach()方法中终止该序列。

使用 RxLifecycleAndroid.bindActivity 进行设置：

```
Observable<Integer> myObservable = Observable.range(0, 25);
```

```
...

  @Override
  public void onResume() {
      super.onResume();
      myObservable
              .compose(RxLifecycleAndroid.bindActivity(lifecycle))
              .subscribe();
  }
```

或者可以指定 lifecycle 事件，在该事件中，RxLifecycle 应该使用 RxLifecycle.bindUntilEvent 终止 Observable 序列。

这里，我们指定在 onDestroy()方法中终止 Observable 序列：

```
@Override
public void onResume() {
  super.onResume();
  myObservable
    .compose(RxLifecycle.bindUntilEvent(lifecycle, ActivityEvent.DESTROY))
    .subscribe();
}
```

2. AutoDispose

GitHub 下载地址：https://github.com/uber/AutoDispose

AutoDispose 是 Uber 开源的库，它与 RxLifecycle 的区别是，不仅可以在 Android 平台上使用，还可以在 Java（的企业级）平台上使用，适用的范围更加宽广。

现在除 CompositeDisposable 和 RxLifecycle 外，还多了一个选择——AutoDispose。AutoDispose 支持 Kotlin、Android Architecture Components，并且 AutoDispose 可以与 RxLifecycle 进行相互操作。

9.3 Transformer 在 RxJava 中的使用

1. Transformer 的用途

Transformer 是转换器的意思。早在 RxJava 1.x 版本就已有了 Observable.Transformer、

Single.Transformer 和 Completable.Transformer，在 RxJava 2.x 版本中有 ObservableTransformer、SingleTransformer、CompletableTransformer、FlowableTransformer 和 MaybeTransformer。其中，FlowableTransformer 和 MaybeTransformer 是新增的。由于 RxJava 2 将 Observable 拆分成了 Observable 和 Flowable，所以有了 FlowableTransformer。同样，Maybe 也是 RxJava 2 新增的一个类型，所以有 MaybeTransformer。

Transformer 能够将一个 Observable/Flowable/Single/Completable/Maybe 对象转换成另一个 Observable/Flowable/Single/Completable/Maybe 对象，与调用一系列的内联操作符一模一样。

举个简单的例子，写一个 transformer()方法将一个发射整数的 Observable 转换为发射字符串的 Observable。

```
public static <String> ObservableTransformer<Integer, java.lang.String>
transformer() {
        return new ObservableTransformer<Integer, java.lang.String>() {
            @Override
            public ObservableSource<java.lang.String> apply(@NonNull
Observable<Integer> upstream) {
                return upstream.map(new Function<Integer, java.lang.String>() {
                    @Override
                    public java.lang.String apply(@NonNull Integer integer)
throws Exception {
                        return java.lang.String.valueOf(integer);
                    }
                });
            }
        };
    }
```

接下来是使用 transformer()方法，通过标准的 RxJava 的操作。

```
Observable.just(123,456)
        .compose(transformer())
        .subscribe(new Consumer<String>() {
            @Override
             public void accept(@io.reactivex.annotations.NonNull String s)
throws Exception {
                System.out.println("s="+s);
                }
            });
```

最后打印了二次，分别是：

```
s=123
s=456
```

通过这个例子，可以简单直观地了解到 Transformer 的作用。

其实，在大名鼎鼎的图片加载框架 Glide 以及 Picasso 中也有类似的 Transform 概念，能够变换图形。

2. 与 compose 操作符结合使用

compose 操作符能够从数据流中得到原始的被观察者。当创建被观察者时，compose 操作符会立即执行，而不像其他的操作符需要在 onNext()调用后才能执行。

关于 compose 操作符，推荐一篇文章，Don't break the chain: use RxJava's compose() operator，国内也有相应的翻译 http://www.jianshu.com/p/e9e03194199e。

常用的场景

（1）切换到主线程

对于网络请求，我们经常会做如下操作来切换线程：

```
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
```

做一个简单的封装：

```
import io.reactivex.FlowableTransformer
import io.reactivex.ObservableTransformer
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers

/**
 * Created by Tony Shen on 2017/7/13.
 */
object RxJavaUtils {

    @JvmStatic
    fun <T> observableToMain():ObservableTransformer<T, T> {
```

```
        return ObservableTransformer{
            upstream ->
            upstream.subscribeOn(Schedulers.io())
                    .observeOn(AndroidSchedulers.mainThread())
        }
    }

    @JvmStatic
    fun <T> flowableToMain(): FlowableTransformer<T, T> {

        return FlowableTransformer{
            upstream ->
            upstream.subscribeOn(Schedulers.io())
                    .observeOn(AndroidSchedulers.mainThread())
        }
    }
}
```

上面这段代码是用 Kotlin 写的，笔者的个人习惯是把一些工具类用 Kotlin 来编写，而且 Kotlin 的 Lambda 表达式也更为直观。

对于 Flowable 切换到主线程的操作，可以这样使用

```
.compose(RxJavaUtils.flowableToMain())
```

（2）RxLifecycle 中的 LifecycleTransformer

9.2 节蹭提到过，trello 出品的 RxLifecycle 能够配合 Android 的生命周期，防止 App 内存泄漏，其中就使用了 LifecycleTransformer。 知乎也做了一个类似的 RxLifecycle，能够做同样的事情。

在笔者的项目中也使用了知乎的 RxLifecycle，并且对 LifecycleTransformer 稍微做了一些修改，将五个 Transformer 合并成了一个。

```
import org.reactivestreams.Publisher;

import io.reactivex.Completable;
import io.reactivex.CompletableSource;
import io.reactivex.CompletableTransformer;
import io.reactivex.Flowable;
```

```
import io.reactivex.FlowableTransformer;
import io.reactivex.Maybe;
import io.reactivex.MaybeSource;
import io.reactivex.MaybeTransformer;
import io.reactivex.Observable;
import io.reactivex.ObservableSource;
import io.reactivex.ObservableTransformer;
import io.reactivex.Single;
import io.reactivex.SingleSource;
import io.reactivex.SingleTransformer;
import io.reactivex.annotations.NonNull;
import io.reactivex.functions.Function;
import io.reactivex.functions.Predicate;
import io.reactivex.processors.BehaviorProcessor;

/**
 * Created by Tony Shen on 2017/5/25.
 */

public class LifecycleTransformer<T> implements ObservableTransformer<T, T>,
        FlowableTransformer<T, T>,
        SingleTransformer<T, T>,
        MaybeTransformer<T, T>,
        CompletableTransformer {

    private final BehaviorProcessor<Integer> lifecycleBehavior;

    private LifecycleTransformer() throws IllegalAccessException {
        throw new IllegalAccessException();
    }

    public LifecycleTransformer(@NonNull BehaviorProcessor<Integer>
lifecycleBehavior) {
        this.lifecycleBehavior = lifecycleBehavior;
    }

    @Override
    public CompletableSource apply(Completable upstream) {
        return upstream.ambWith(
                lifecycleBehavior.filter(new Predicate<Integer>() {
                    @Override
```

```
                    public boolean test(@LifecyclePublisher.Event Integer event)
throws Exception {
                        return event == LifecyclePublisher.ON_DESTROY_VIEW ||
                                event == LifecyclePublisher.ON_DESTROY ||
                                event == LifecyclePublisher.ON_DETACH;
                    }
                }).take(1).flatMapCompletable(new Function<Integer,
Completable>() {
                    @Override
                    public Completable apply(Integer flowable) throws Exception
{
                        return Completable.complete();
                    }
                })
        );
    }

    @Override
    public Publisher<T> apply(final Flowable<T> upstream) {
        return upstream.takeUntil(
                lifecycleBehavior.skipWhile(new Predicate<Integer>() {
                    @Override
                    public boolean test(@LifecyclePublisher.Event Integer event)
throws Exception {
                        return event != LifecyclePublisher.ON_DESTROY_VIEW &&
                                event != LifecyclePublisher.ON_DESTROY &&
                                event != LifecyclePublisher.ON_DETACH;
                    }
                })
        );
    }

    @Override
    public MaybeSource<T> apply(Maybe<T> upstream) {
        return upstream.takeUntil(
                lifecycleBehavior.skipWhile(new Predicate<Integer>() {
                    @Override
                    public boolean test(@LifecyclePublisher.Event Integer event)
throws Exception {
                        return event != LifecyclePublisher.ON_DESTROY_VIEW &&
                                event != LifecyclePublisher.ON_DESTROY &&
                                event != LifecyclePublisher.ON_DETACH;
```

```
                }
            })
        );
    }

    @Override
    public ObservableSource<T> apply(Observable<T> upstream) {
        return upstream.takeUntil(
                lifecycleBehavior.skipWhile(new Predicate<Integer>() {
                    @Override
                    public boolean test(@LifecyclePublisher.Event Integer event)
throws Exception {
                        return event != LifecyclePublisher.ON_DESTROY_VIEW &&
                                event != LifecyclePublisher.ON_DESTROY &&
                                event != LifecyclePublisher.ON_DETACH;
                    }
                }).toObservable()
        );
    }

    @Override
    public SingleSource<T> apply(Single<T> upstream) {
        return upstream.takeUntil(
                lifecycleBehavior.skipWhile(new Predicate<Integer>() {
                    @Override
                    public boolean test(@LifecyclePublisher.Event Integer event)
throws Exception {
                        return event != LifecyclePublisher.ON_DESTROY_VIEW &&
                                event != LifecyclePublisher.ON_DESTROY &&
                                event != LifecyclePublisher.ON_DETACH;
                    }
                })
        );
    }
}
```

（3）缓存的使用

对于缓存，我们一般是这样写：

```
cache.put(key,value);
```

更优雅一点的做法是使用 AOP，大致是这样写：

```
@Cacheable(key = "...")
getValue() {
    ....
}
```

如果想在 RxJava 的链式调用中也使用缓存，还可以考虑使用 transformer 的方式，下面就写一个简单的方法：

```
/**
 * Created by Tony Shen on 2017/7/13.
 */

public class RxCache {

    public static <T> FlowableTransformer<T, T> transformer(final String key,
final Cache cache) {
        return new FlowableTransformer<T, T>() {
            @Override
            public Publisher<T> apply(@NonNull Flowable<T> upstream) {

                return upstream.map(new Function<T, T>() {
                    @Override
                    public T apply(@NonNull T t) throws Exception {
                        cache.put(key,(Serializable) t);
                        return t;
                    }
                });

            }
        };
    }
}
```

结合上述三种使用场景，封装了一个方法用于获取内容，在这里网络框架使用 Retrofit。虽然 Retrofit 本身支持通过 Interceptor 的方式来添加 Cache，但是某些业务场景下可能还是想用自己的 Cache，那么可以采用类似下面的封装。

```
    /**
     * 获取内容
     * @param fragment
     * @param param
     * @param cacheKey
     * @return
     */
    public Flowable<ContentModel> getContent(Fragment fragment,ContentParam
param,String cacheKey) {

        return apiService.loadVideoContent(param)
                .compose(RxLifecycle.bind(fragment).<ContentModel>toLifecycle
Transformer())
                .compose(RxJavaUtils.<ContentModel>flowableToMain())
                .compose(RxCache.<ContentModel>transformer(cacheKey,App.getIn
stance().cache));
    }
```

（4）追踪 RxJava 的使用

初学者可能会对 RxJava 内部的数据流向感到困惑，所以笔者写了一个类，用于追踪 RxJava 的数据流向，同时对于调试代码也很有帮助。

先来看一个简单的例子：

```
Observable.just("tony","cafei","aaron")
                .compose(RxTrace.<String>logObservable("first",RxTrace.LOG_SU
BSCRIBE|RxTrace.LOG_NEXT_DATA))
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@io.reactivex.annotations.NonNull String
s) throws Exception {
                        System.out.println("s="+s);
                    }
                });
```

图 9-1 显示了上面代码中的数据流向。

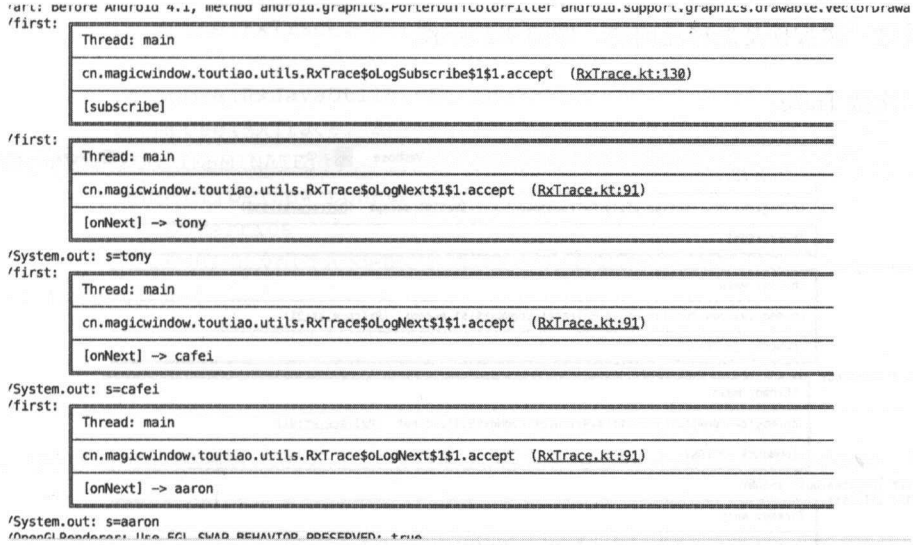


图 9-1

然后，在刚才代码的基础上加一个 map 操作符，把小写的字符串都转换成大写。

```
Observable.just("tony","cafei","aaron")
                .compose(RxTrace.<String>logObservable("first",RxTrace.LOG_SU
BSCRIBE|RxTrace.LOG_NEXT_DATA))
                .map(new Function<String, String>() {


                    @Override
                    public String apply(@io.reactivex.annotations.NonNull
String s) throws
                            Exception {
                        return s.toUpperCase();
                    }
                })
                .compose(RxTrace.<String>logObservable("second",RxTrace.LOG_N
EXT_DATA))
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@io.reactivex.annotations.NonNull String
s) throws Exception {
                        System.out.println("s="+s);
                    }
                });
```

看看这一次数据是怎样流向的，如图 9-2 所示。

第二次做 Trace。

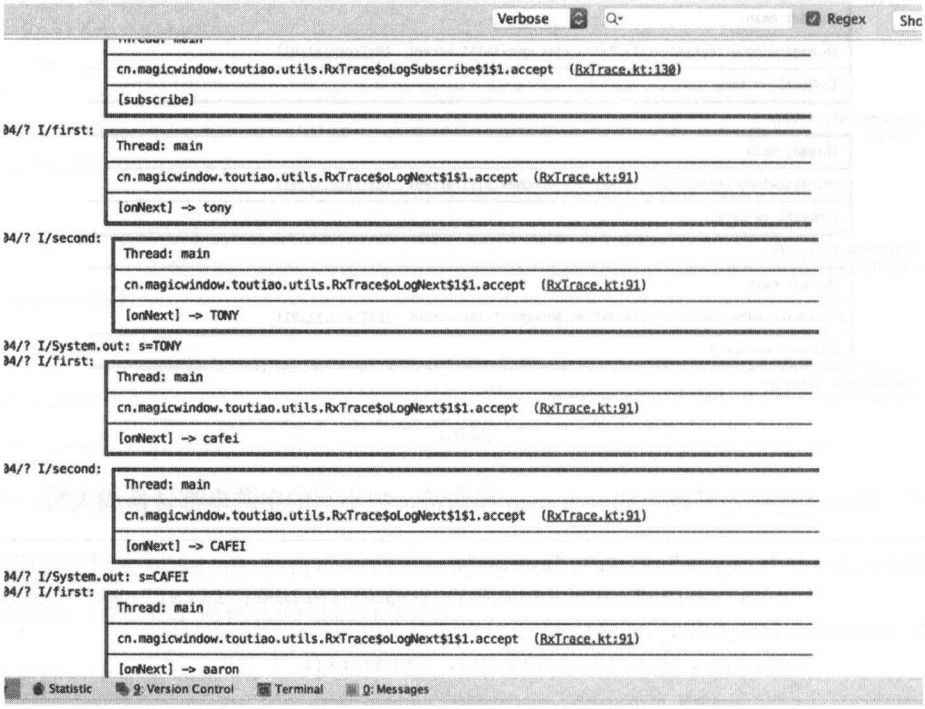


图 9-2

最后，再加上对 onComlete 和 OnTerminate 的监测。

```
Observable.just("tony","cafei","aaron")
                .compose(RxTrace.<String>logObservable("first",RxTrace.LOG_SU
BSCRIBE|RxTrace.LOG_NEXT_DATA))
                .map(new Function<String, String>() {


                    @Override
                    public String apply(@io.reactivex.annotations.NonNull
String s) throws
                            Exception {
                        return s.toUpperCase();
                    }
                })
```

```
                .compose(RxTrace.<String>logObservable("second",RxTrace.LOG_N
EXT_DATA))
                .compose(RxJavaUtils.<String>observableToMain())
                .compose(RxTrace.<String>logObservable("third",RxTrace.LOG_CO
MPLETE|RxTrace.LOG_TERMINATE))
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@io.reactivex.annotations.NonNull String
s) throws Exception {
                        System.out.println("s="+s);
                    }
                });
```

上面已经展示过的截图就不显示了，这里展示最后的 onComlete 和 OnTerminate，如图 9-3 所示。

第三次做 Trace。

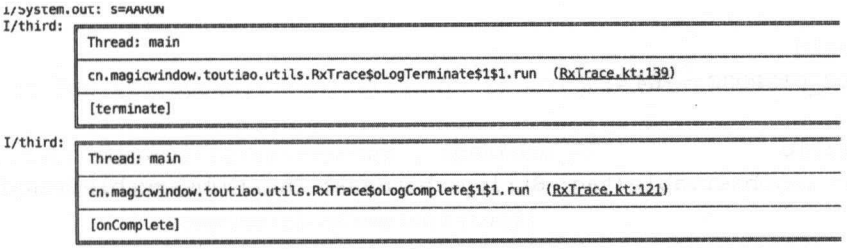


图 9-3

最后，把 RxTrace 的代码展示出来，它是由 Kotlin 编写的，大量使用了 transformer 和 compose：

```
import com.saframework.log.L
import io.reactivex.ObservableTransformer
import io.reactivex.functions.Function



/**
 * Created by Tony Shen on 2017/7/11.
 */

object RxTrace {
```

```
    @JvmField
    val LOG_NEXT_DATA = 1

    @JvmField
    val LOG_NEXT_EVENT = 2

    @JvmField
    val LOG_ERROR = 4

    @JvmField
    val LOG_COMPLETE = 8

    @JvmField
    val LOG_SUBSCRIBE = 16

    @JvmField
    val LOG_TERMINATE = 32

    @JvmField
    val LOG_DISPOSE = 64

    @JvmStatic
    fun <T> logObservable(tag: String, bitMask: Int): ObservableTransformer<T,
T> {

        return ObservableTransformer {upstream->
            var upstream = upstream

            if (bitMask and LOG_SUBSCRIBE > 0) {
                upstream = upstream.compose(oLogSubscribe(tag))
            }
            if (bitMask and LOG_TERMINATE > 0) {
                upstream = upstream.compose(oLogTerminate(tag))
            }
            if (bitMask and LOG_ERROR > 0) {
                upstream = upstream.compose(oLogError(tag))
            }
            if (bitMask and LOG_COMPLETE > 0) {
                upstream = upstream.compose(oLogComplete(tag))
            }
            if (bitMask and LOG_NEXT_DATA > 0) {
```

```
            upstream = upstream.compose(oLogNext(tag))
        } else if (bitMask and LOG_NEXT_EVENT > 0) {
            upstream = upstream.compose(oLogNextEvent(tag))
        }
        if (bitMask and LOG_DISPOSE > 0) {
            upstream = upstream.compose(oLogDispose(tag))
        }
        upstream
    }
}

@JvmStatic
fun <T> log(tag: String): ObservableTransformer<T, T> {

    return ObservableTransformer { upstream ->
        upstream.compose(oLogAll(tag))
                .compose(oLogNext(tag))
    }
}

private fun <T> oLogAll(tag: String): ObservableTransformer<T, T> {

    return ObservableTransformer { upstream ->
        upstream.compose(oLogError(tag))
                .compose(oLogComplete(tag))
                .compose(oLogSubscribe(tag))
                .compose(oLogTerminate(tag))
                .compose(oLogDispose(tag))
    }
}

private fun <T> oLogNext(tag: String): ObservableTransformer<T, T> {

    return ObservableTransformer {
        upstream -> upstream.doOnNext{
            data ->
                L.i(tag, String.format("[onNext] -> %s", data))
        }
    }
}

private fun <T> oLogNextEvent(tag: String): ObservableTransformer<T, T> {
```

```
        return ObservableTransformer {
            upstream -> upstream.doOnNext {
                L.i(tag, String.format("[onNext]"))
            }
        }
    }

    private fun <T> oLogError(tag: String): ObservableTransformer<T, T> {

        val message = Function<Throwable, String> { throwable -> if
(throwable.message != null) throwable.message!! else
throwable.javaClass.simpleName }

        return ObservableTransformer {
            upstream -> upstream.doOnError {
                throwable ->
                    L.i(tag, String.format("[onError] -> %s",
message.apply(throwable)))
            }
        }
    }

    private fun <T> oLogComplete(tag: String): ObservableTransformer<T, T> {

        return ObservableTransformer {
            upstream -> upstream.doOnComplete {
                L.i(tag, String.format("[onComplete]"))
            }
        }
    }

    private fun <T> oLogSubscribe(tag: String): ObservableTransformer<T, T> {

        return ObservableTransformer {
            upstream -> upstream.doOnSubscribe {
                L.i(tag, String.format("[subscribe]"))
            }
        }
    }

    private fun <T> oLogTerminate(tag: String): ObservableTransformer<T, T> {
```

```
        return ObservableTransformer {
            upstream -> upstream.doOnTerminate {
                L.i(tag, String.format("[terminate]"))
            }
        }
    }

    private fun <T> oLogDispose(tag: String): ObservableTransformer<T, T> {

        return ObservableTransformer {
            upstream -> upstream.doOnDispose {
                L.i(tag, String.format("[dispose]"))
            }
        }
    }
}
```

compose 操作符和 Transformer 结合使用，一方面可以让代码看起来更加简洁，另一方面能够提高代码的复用性。RxJava 提倡链式调用，compose 能够防止链式被打破。

9.4　小结

本章介绍了 Disposable 以及 Transformer 的用途。Disposable 能够管理一个订阅，Transformer 顾名思义是转换器的意思，能够将一个 Observable、Flowable、Single、Completable、Maybe 对象转换成另一个 Observable、Flowable、Single、Completable、Maybe 对象。

虽然它们看起来风马牛不相及，但是可以借助 compose 操作符将它们联系在一起。

第 10 章

RxJava的并行编程

10.1 RxJava 并行操作

第 4.2 节中，我们详细介绍了 RxJava 的线程模型，被观察者（Observable/Flowable/Single/Completable/Maybe）发射的数据流可以经历各种线程切换，但是数据流的各个元素之间不会产生并行执行的效果。并行不是并发，也不是同步，更不是异步。

并发（concurrency）是指一个处理器同时处理多个任务。并行（parallelism）是多个处理器或者是多核的处理器同时处理多个不同的任务。并行是同时发生的多个并发事件，具有并发的含义，而并发则不一定是并行。

Java 8 新增了并行流来实现并行的效果，只需在集合上调用 parallelStream()方法即可。

```
List<Integer> result = new ArrayList();
for(Integer i=1;i<=100;i++) {

    result.add(i);
}

result.parallelStream()
        .map(new java.util.function.Function<Integer, String>() {

            @Override
            public String apply(Integer integer) {
                return integer.toString();
            }
```

```
            }).forEach(new java.util.function.Consumer<String>() {
        @Override
        public void accept(String s) {

            System.out.println("s="+s+";Current Thread
Name="+Thread.currentThread().getName());
        }
    });
```

执行结果如图 10-1 所示。

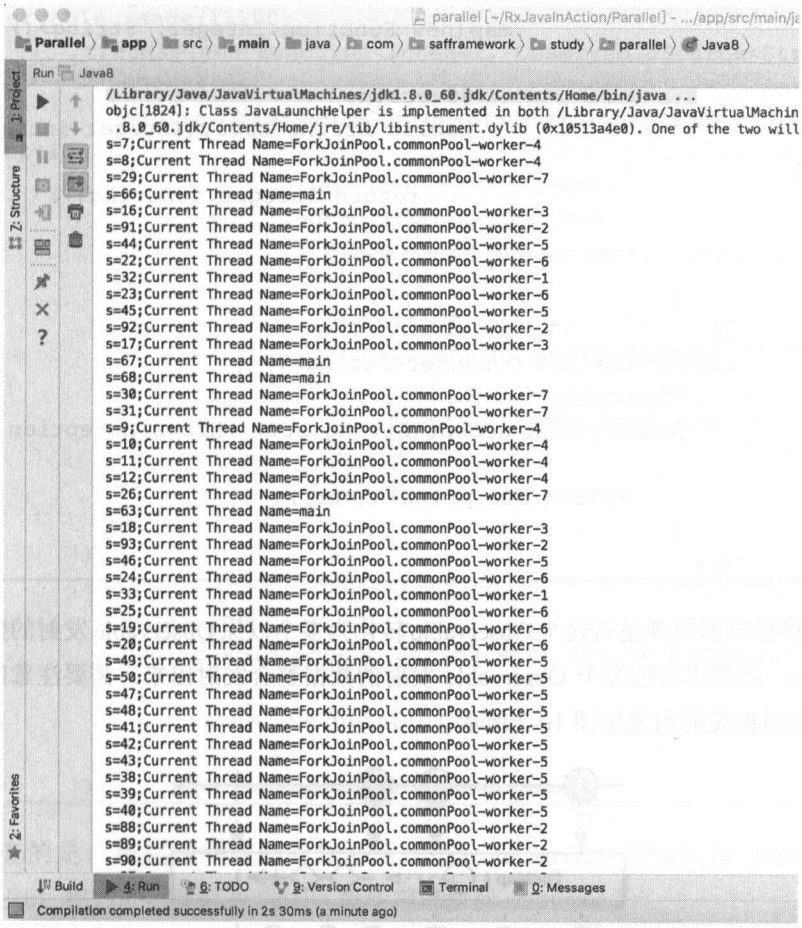


图 10-1

从执行结果中可以看到，Java 8 借助了 JDK 的 fork/join 框架来实现并行编程。

1. 借助 flatMap 实现并行

在 RxJava 中可以借助 flatMap 操作符来实现类似于 Java 8 的并行执行效果。

```
        Observable.range(1,100)
                .flatMap(new Function<Integer, ObservableSource<String>>() {
                    @Override
                    public ObservableSource<String> apply(Integer integer)
throws Exception {
                        return Observable.just(integer)
                                .subscribeOn(Schedulers.computation())
                                .map(new Function<Integer, String>() {

                                    @Override
                                    public String apply(Integer integer) throws
Exception {
                                        return integer.toString();
                                    }
                                });
                    }
                })
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String str) throws Exception {

                        System.out.println(str);
                    }
                });
```

flatMap 操作符的原理是将这个 Observable 转化为多个以原 Observable 发射的数据作为源数据 Observable，然后再将这多个 Observable 发射的数据整合发射出来。需要注意的是，最后的顺序可能会交错地发射出来如图 10-2 所示。

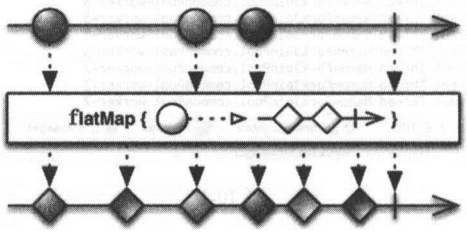


图 10-2

flatMap 会对原始 Observable 发射的每一项数据执行变换操作。在这里，生成的每个 Observable 使用线程池（指定了 computation 作为 Scheduler）并发地执行。

当然，我们还可以使用 ExecutorService 来创建一个 Scheduler，对刚才的代码稍微做一些改动。

```
        int threadNum = Runtime.getRuntime().availableProcessors()+1;

        ExecutorService executor = Executors.newFixedThreadPool(threadNum);
        final Scheduler scheduler = Schedulers.from(executor);
        Observable.range(1,100)
                .flatMap(new Function<Integer, ObservableSource<String>>() {
                    @Override
                    public ObservableSource<String> apply(Integer integer)
throws Exception {
                        return Observable.just(integer)
                                .subscribeOn(scheduler)
                                .map(new Function<Integer, String>() {

                                    @Override
                                    public String apply(Integer integer) throws
Exception {
                                        return integer.toString();
                                    }
                                });
                    }
                })
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String str) throws Exception {

                        System.out.println(str);
                    }
                });
```

需要补充的是：当完成所有的操作之后，ExecutorService 需要执行 shutdown()来关闭 ExecutorService。我们可以使用 doFinally 操作符来执行 shutdown()。

doFinally 操作符可以在 onError 或者 onComplete 之后调用指定的操作，或由下游处理。

增加了 doFinally 操作符之后，代码是这样的。

```
        int threadNum = Runtime.getRuntime().availableProcessors()+1;

        final ExecutorService executor =
Executors.newFixedThreadPool(threadNum);
        final Scheduler scheduler = Schedulers.from(executor);
        Observable.range(1,100)
                .flatMap(new Function<Integer, ObservableSource<String>>() {
                    @Override
                    public ObservableSource<String> apply(Integer integer)
throws Exception {
                        return Observable.just(integer)
                                .subscribeOn(scheduler)
                                .map(new Function<Integer, String>() {

                                    @Override
                                    public String apply(Integer integer) throws
Exception {
                                        return integer.toString();
                                    }
                                });
                    }
                })
                .doFinally(new Action() {
                    @Override
                    public void run() throws Exception {
                        executor.shutdown();
                    }
                })
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(String str) throws Exception {

                        System.out.println(str);
                    }
                });
```

2. 通过 Round-Robin 算法实现并行

Round-Robin 算法是最简单的一种负载均衡算法。它的原理是把来自用户的请求轮流分配给内部的服务器：从服务器 1 开始，直到服务器 N，然后重新开始循环，也被称为哈希取模法，是非常常用的数据分片方法。Round-Robin 算法的优点是简洁，它无须记录当前所有连接的状

态，所以是一种无状态调度。

通过 Round-Robin 算法把数据按线程数分组，例如分成 5 组，每组个数相同，一起发送处理。这样做的目的是可以减少 Observable 的创建，从而节省系统资源，但是会增加处理时间。Round-Robin 算法可以看成是对时间和空间的综合考虑。

```
        final AtomicInteger batch = new AtomicInteger(0);

        Observable.range(1,100)
                .groupBy(new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(@NonNull Integer integer) throws
Exception {
                        return batch.getAndIncrement() % 5;
                    }
                })
                .flatMap(new Function<GroupedObservable<Integer, Integer>,
ObservableSource<?>>() {
                    @Override
                    public ObservableSource<?> apply(@NonNull
GroupedObservable<Integer, Integer> integerIntegerGroupedObservable) throws
Exception {
                        return
integerIntegerGroupedObservable.observeOn(Schedulers.io())
                                .map(new Function<Integer, String>() {

                                    @Override
                                    public String apply(@NonNull Integer integer)
throws Exception {
                                        return integer.toString();
                                    }
                                });
                    }
                })
                .subscribe(new Consumer<Object>() {
                    @Override
                    public void accept(@NonNull Object o) throws Exception {
                        System.out.println(o);
                    }
                });
```

在这里，我们也可以使用 ExecutorService 创建 Scheduler，来替代 Schedulers.io()。

```
        final AtomicInteger batch = new AtomicInteger(0);

        int threadNum = 5;

        final ExecutorService executor =
Executors.newFixedThreadPool(threadNum);
        final Scheduler scheduler = Schedulers.from(executor);

        Observable.range(1,100)
                .groupBy(new Function<Integer, Integer>() {
                    @Override
                    public Integer apply(@NonNull Integer integer) throws
Exception {
                        return batch.getAndIncrement() % threadNum;
                    }
                })
                .flatMap(new Function<GroupedObservable<Integer, Integer>,
ObservableSource<?>>() {
                    @Override
                    public ObservableSource<?> apply(@NonNull
GroupedObservable<Integer, Integer> integerIntegerGroupedObservable) throws
Exception {
                        return
integerIntegerGroupedObservable.observeOn(scheduler)
                                .map(new Function<Integer, String>() {

                                    @Override
                                    public String apply(@NonNull Integer integer)
throws Exception {
                                        return integer.toString();
                                    }
                                });
                    }
                })
                .subscribe(new Consumer<Object>() {
                    @Override
                    public void accept(@NonNull Object o) throws Exception {
                        System.out.println(o);
                    }
                });
```

10.2　ParallelFlowable

10.2.1　ParallelFlowable 介绍

如图 10-3 所示，RxJava 2.0.5 版本新增了 ParallelFlowable API，它允许并行地执行一些操作符，例如 map、filter、concatMap、flatMap、collect、reduce 等。

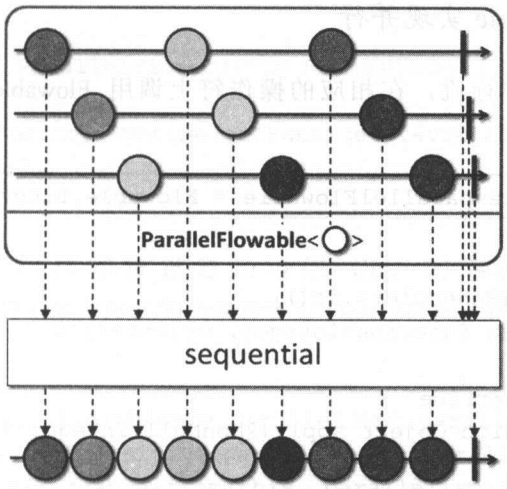


图 10-3

```
/**
 * Abstract base class for Parallel publishers that take an array of Subscribers.
 * <p>
 * Use {@code from()} to start processing a regular Publisher in 'rails'.
 * Use {@code runOn()} to introduce where each 'rail' should run on thread-vise.
 * Use {@code sequential()} to merge the sources back into a single Flowable.
 *
 * <p>History: 2.0.5 - experimental
 * @param <T> the value type
 * @since 2.1 - beta
 */
@Beta
public abstract class ParallelFlowable<T> {
   ...
}
```

ParallelFlowable 是并行的 Flowable 版本，并不是新增的被观察者类型。在 ParallelFlowable

中，很多典型的操作符（如 take、skip 等）是不可用的。

在 RxJava 中并没有 ParallelObservable，因为在 RxJava 2 之后，Observable 不再支持背压。然而在并行处理中背压是必不可少的，否则会淹没在并行操作符的内部队列中。

同理，也不存在 ParallelSingle、ParallelCompletable 以及 ParallelMaybe。

1. ParallelFlowable 实现并行

类似于 Java 8 的并行流，在相应的操作符上调用 Flowable 的 parallel()就会返回 ParallelFlowable。

```
ParallelFlowable parallelFlowable = Flowable.range(1,100).parallel();

parallelFlowable
        .runOn(Schedulers.io())
        .map(new Function<Integer, Object>() {

            @Override
            public Object apply(@NonNull Integer integer) throws
Exception {
                return integer.toString();
            }
        })
        .sequential()
        .subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String str) throws Exception {
                System.out.println(str);
            }
        });
```

其中，parallel()调用了 ParallelFlowable.from(@NonNull Publisher<? extends T> source)

```
public final ParallelFlowable<T> parallel() {
    return ParallelFlowable.from(this);
}
```

ParallelFlowable 的 from()方法是通过 Publisher 并以循环的方式在多个“轨道”（CPU 数）上消费它的。

```
    /**
     * Take a Publisher and prepare to consume it on multiple 'rails' (number
of CPUs)
     * in a round-robin fashion.
     * @param <T> the value type
     * @param source the source Publisher
     * @return the ParallelFlowable instance
     */
    @CheckReturnValue
    public static <T> ParallelFlowable<T> from(@NonNull Publisher<? extends T>
source) {
        return from(source, Runtime.getRuntime().availableProcessors(),
Flowable.bufferSize());
    }
```

默认情况下，并行级别被设置为可用 CPU 的数量（Runtime.getRuntime().availableProcessors()），并且顺序源的预取量设置为 Flowable.bufferSize()。两者都可以通过重载 parallel()方法来指定。

```
    public final ParallelFlowable<T> parallel(int parallelism) {
        ObjectHelper.verifyPositive(parallelism, "parallelism");
        return ParallelFlowable.from(this, parallelism);
    }

    public final ParallelFlowable<T> parallel(int parallelism, int prefetch) {
        ObjectHelper.verifyPositive(parallelism, "parallelism");
        ObjectHelper.verifyPositive(prefetch, "prefetch");
        return ParallelFlowable.from(this, parallelism, prefetch);
    }
```

最后，如果已经使用了必要的并行操作，则可以通过 ParallelFlowable.sequential()操作符返回到顺序流。

```
parallelFlowable
                .....
                .sequential()
                .subscribe(new Consumer<Object>() {
                    @Override
                    public void accept(@NonNull Object o) throws Exception {

                    }
```

```
        });
```

2. ParallelFlowable 与 Scheduler

ParallelFlowable 遵循与 Flowable 相同的异步原理，因此 parallel()本身并不引入顺序源的异步消耗，只准备并行流，但是可以通过 runOn(Scheduler)操作符定义异步。这点与 Flowable 有很大不同，Flowable 使用 subscribeOn、observeOn 操作符。

```
ParallelFlowable<Integer> psource = source.runOn(Schedulers.io());
```

runOn()可以指定 prefetch 的数量。

```
    public final ParallelFlowable<T> runOn(@NonNull Scheduler scheduler) {
        return runOn(scheduler, Flowable.bufferSize());
    }

    public final ParallelFlowable<T> runOn(@NonNull Scheduler scheduler, int
prefetch) {
        ObjectHelper.requireNonNull(scheduler, "scheduler");
        ObjectHelper.verifyPositive(prefetch, "prefetch");
        return RxJavaPlugins.onAssembly(new ParallelRunOn<T>(this, scheduler,
prefetch));
    }
```

10.2.2 ParallelFlowable 的操作符

并非所有的顺序操作在并行世界中都是有意义的。目前 ParallelFlowable 只支持如下操作：

map、filter, flatMap, concatMap, reduce, collect, sorted, toSortedList, compose, fromArray, doOnCancel, doOnError, doOnComplete, doOnNext, doAfterNext, doOnSubscribe, doAfterTerminated, doOnRequest

这些 ParallelFlowable 可用的操作符，使用方法与 Flowable 中的用法一样。

10.2.3 ParallelFlowable 和 Flowable.flatMap 比较

前面介绍了使用 Observable.flatMap 来实现并行。Flowable.flatMap 实现并行的原理和 Observable.flatMap 实现并行的原理相同。

那么什么时候使用 flatMap 进行并行处理比较好，什么时候使用 ParallelFlowable 比较好呢？

RxJava 本质上是连续的，借助 flatMap 操作符进行分离和加入一个序列可能会变得很复杂，并引起一定的开销。但是如果使用 ParallelFlowable，则开销会更小。

然而，ParallelFlowable 的操作符很有限，如果有一些特殊的操作需要并行执行，而这些操作不能用 ParallelFlowable 所支持的操作符来表达，那么就应该使用基于 Flowable.flatMap 来实现并行。

因此，优先推荐使用 ParallelFlowable，对于无法使用 ParallelFlowable 的操作符，则可以使用 flatMap 来实现并行。

10.3　小结

随着多核处理器上 CPU 核数的增加，如何高效地利用这些计算资源以满足日趋复杂的业务场景变得越来越重要。

在本章，我们介绍了如何通过使用 RxJava 来实现并行编程，本章的所有代码均已上传到了 GitHub。

第 11 章
RxBinding的使用

11.1 RxBinding 简介

1. RxBinding 介绍

RxBinding 是 Jake Wharton 大神写的框架，它的 API 能够把 Android 平台和兼容包内的 UI 控件变为 Observable 对象，这样就可以把 UI 控件的事件当作 RxJava 中的数据流来使用了。比如 View 的 onClick 事件，使用 RxView.clicks(view) 即可获取一个 Observable 对象，每当用户单击这个 View 的时候，该 Observable 对象就会发射一个事件， Observable 的观察者就可以通过 onNext 回调知道用户单击了 View。

RxBinding 的 GitHub 地址：https://github.com/JakeWharton/RxBinding

RxBinding 的优点：

- ◎ 它是对 Android View 事件的扩展，它使得开发者可以对 View 事件使用 RxJava 的各种操作。
- ◎ 提供了与 RxJava 一致的回调，使得代码简洁明了，尤其是页面上充斥着大量的监听事件，以及各种各样的匿名内部类。
- ◎ 几乎支持所有的常用控件及事件（v4、v7、design、recyclerview 等），另外每个库还有对应的 Kotlin 支持库。

RxBinding 的下载（仅针对 Gradle 项目）。

Android 平台 bindings：

```
compile 'com.jakewharton.rxbinding2:rxbinding:2.0.0'
```

‘support-v4’ 包 bindings:

```
compile 'com.jakewharton.rxbinding2:rxbinding-support-v4:2.0.0'
```

‘appcompat-v7’ 包 bindings：

```
compile 'com.jakewharton.rxbinding2:rxbinding-appcompat-v7:2.0.0'
```

‘design’ 包 bindings：

```
compile 'com.jakewharton.rxbinding2:rxbinding-design:2.0.0'
```

‘recyclerview-v7’ 包 bindings：

```
compile 'com.jakewharton.rxbinding2:rxbinding-recyclerview-v7:2.0.0'
```

‘leanback-v17’ 包 bindings:

```
compile 'com.jakewharton.rxbinding2:rxbinding-leanback-v17:2.0.0'
```

RxBinding 的每一个子项目都有针对 Kotlin 的扩展，比如 rxbinding 变成 rxbinding-kotlin，rxbinding-support-v4 变成 rxbinding-support-v4-kotlin。

2. 响应式的 Android UI

对 UI 事件（例如点击、滑动和文本输入）的响应几乎是 Android App 开发的基本部分，但是 Android SDK 对 UI 事件的处理有些复杂，我们通常需要使用各种 listeners、handlers、TextWatchers 和其他组件等组合来响应 UI 事件。这些组件中的每一个都需要编写大量的样板代码，更为糟糕的是，实现这些不同组件的方式并不一致。例如，你可以通过实现 OnClickListener 来处理 OnClick 事件：

```
Button button = (Button)findViewById(R.id.button);
button.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View v) {
        //do something
    }
});
```

但这与实现 TextWatcher 的方式完全不同：

```
final EditText name = (EditText) v.findViewById(R.id.name);
name.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after)
{
    }

    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count)
{
        //do something
    }

    @Override
    public void afterTextChanged(Editable s) {
    }
});
```

这种一致性的缺乏可能会为代码增加很多复杂性。如果有些 UI 组件需要依赖于其他 UI 组件的输出，那么事情会变得更加复杂。

即使是一个简单的需求，例如要求用户将其名称输入到 EditText，以便个性化地展示 TextViews 的文本内容，而 TextViews 需要嵌套回调，这是非常难以实现和维护的（有人将嵌套回调称为“回调地狱”）。

显然，处理 UI 事件的标准化方法有大大简化代码的空间，而 RxBinding 就是这样的库，它提供的绑定能够将任何 Android View 事件转换为一个 Observable。

一旦将 View 事件转换为 Observable，它将发射数据流形式的 UI 事件，我们就可以订阅这个数据流，这与订阅其他 Observable 的方式相同。接下来，看看如何实现 OnClick 事件：

```
Button button = (Button) findViewById(R.id.button);
RxView.clicks(button)
     .subscribe(new Consumer<Object>() {
                @Override
                public void accept(@NonNull Object o) throws Exception {
                    //do something
                }
```

```
        });
```

这种方法不仅更简洁，而且是一种标准的实现方式，我们可以将其应用于整个 App 的所有 UI 事件。例如，捕获文本输入与捕获点击事件的模式是一样的：

```
RxTextView.textChanges(editText)
        .subscribe(new Consumer<CharSequence>() {
            @Override
            public void accept(CharSequence charSequence) throws Exception {
                //do something
            }
        });
```

11.2　RxBinding 使用场景

RxBinding 可以应用于整个 App 的所有 UI 事件，下面列举一些 RxBinding 比较常见的使用场景。

1. 点击事件

按钮的点击事件是每一个 App 都会用到的场景，可以使用 RxView 的 clicks(@NonNull View view)方法来绑定 UI 控件，如图 11-1 所示。

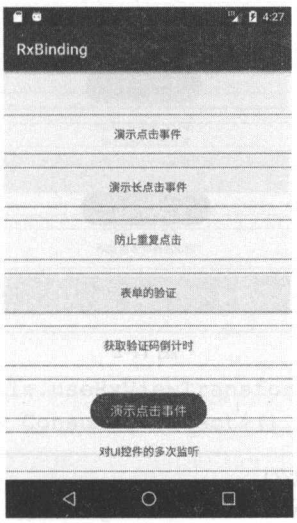


图 11-1

示例代码：

```
RxView.clicks(text1)
        .subscribe(new Consumer<Object>() {
            @Override
            public void accept(@NonNull Object o) throws Exception {

                Toast.makeText(MainActivity.this,"演示点击事件
",Toast.LENGTH_SHORT).show();
            }
        });
```

点击“演示点击事件”按钮后，会弹出一个 toast。

2. 长点击事件

长点击事件也是一个比较常见的事件，可以使用 RxView 的 longClicks(@NonNull View view) 方法来绑定 UI 控件，如图 11-2 所示。

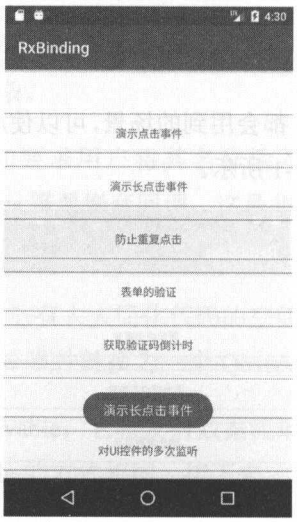


图 11-2

示例代码：

```
RxView.longClicks(text2)
        .subscribe(new Consumer<Object>() {
            @Override
```

```
                public void accept(@NonNull Object o) throws Exception {

                    Toast.makeText(MainActivity.this,"演示长点击事件
",Toast.LENGTH_SHORT).show();
                }
            });
```

长按“演示长点击事件”按钮之后，会弹出一个 toast。

3. 防止重复点击

在弱网络环境下，经常会遇到点击某个按钮没有响应的情况，此时心急的用户可能会多次点击按钮，从而造成事件的多次触发，显然这是我们不愿意看到的情况。可以利用 throttleFirst 操作符获取某段时间内的第一次点击事件，如图 11-3 所示。

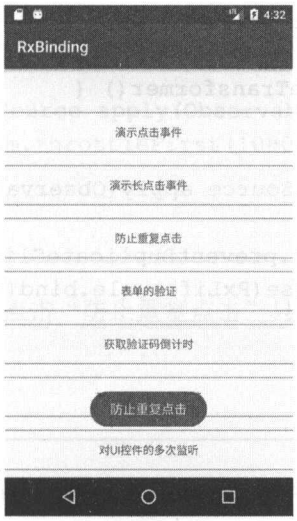


图 11-3

示例代码：

```
RxView.clicks(text3)
        .compose(RxUtils.useRxViewTransformer(MainActivity.this))
        .subscribe(new Consumer<Object>() {
            @Override
            public void accept(@NonNull Object o) throws Exception {
```

```
                        Toast.makeText(MainActivity.this,"防止重复点击
",Toast.LENGTH_SHORT).show();
                    }
                });
```

这里使用了 compose 操作符，它封装了两个操作：一个是防止 UI 事件的重复点击，另一个是绑定 Activity 的生命周期，防止内存泄露。来看一下 useRxViewTransformer 方法的具体代码，它里面包含了两个 compose()。

```
    /**
     * 对 RxView 绑定的事件
     * 封装了防止重复点击和 RxLifecycle 的生命周期
     */
    public static ObservableTransformer useRxViewTransformer(final
AppCompatActivity targetActivity) {

        return new ObservableTransformer() {

            @Override
            public ObservableSource apply(Observable upstream) {
                return
upstream.compose(RxJavaUtils.preventDuplicateClicksTransformer())
                        .compose(RxLifecycle.bind(targetActivity).toLifecycle
Transformer());
            }
        };
    }
```

先来看看第一个 compose 的作用，第二个 compose 在后面的小节会讲解。

```
    /**
     * 防止重复点击的 Transformer
     */
    @JvmStatic
    fun <T> preventDuplicateClicksTransformer(): ObservableTransformer<T, T> {
        return ObservableTransformer { upstream ->
            upstream.throttleFirst(1000, TimeUnit.MILLISECONDS)
        }
    }
```

这是一段 Kotlin 代码，出自笔者自己封装的 Android 库。它的用途是在 1s 内只取第一次点击，这样可以防止 1s 内产生多次点击事件。

这个库完全使用 Kotlin 编写，主要作用是收集常见的 Android 工具类，这目前还在完善中。

GitHub 地址：https://github.com/fengzhizi715/SAF-Kotlin-Utils

将 Kotlin 的代码翻译成 Java 的代码也很简单。

```
    /**
     * 防止重复点击的 Transformer
     */
    public static ObservableTransformer
preventDuplicateClicksTransformer(final AppCompatActivity targetActivity) {

        return new ObservableTransformer() {

            @Override
            public ObservableSource apply(Observable upstream) {
                return upstream.throttleFirst(1000, TimeUnit.MILLISECONDS);
            }
        };
    }
```

所以在示例代码中，1s 内多次点击“防止重复点击”只会出现一个 toast。若 1s 后再次点击，才能再看到新的 toast。

4. 表单的验证

App 内常见的表单验证是用户登录页面，我们需要对用户名、密码做一些校验。对于校验，有些是服务端做的，例如，用户名是否存在、用户名的密码是否正确等。而有些校验则需要客户端来做，例如，用户名是否输入、输入的用户名是否规范、密码是否输入等。

例如，手机号码不足 11 位时，会出现一个提示，如图 11-4 所示。

如果密码没有输入，就点击“登录”按钮，则会弹出一个提示，如图 11-5 所示。

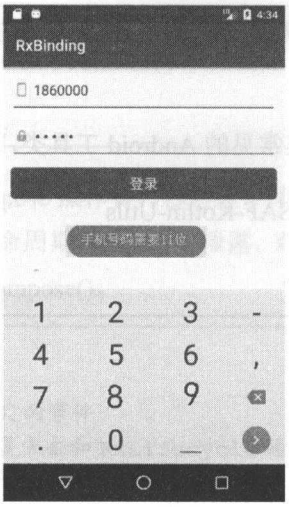


图 11-4

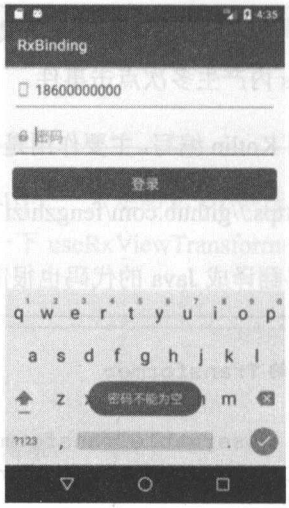


图 11-5

只有当用户名和密码都输入正确时，点击“登录”按钮才会告知用户登录成功，如图 11-6 所示。

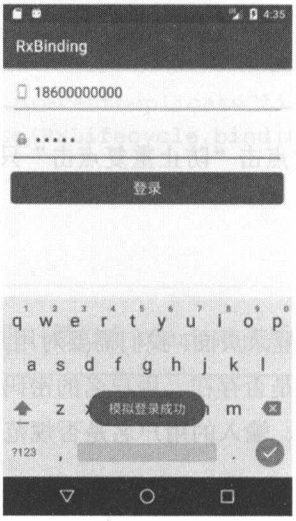


图 11-6

示例代码：

```
    Observable<CharSequence> ObservablePhone =
RxTextView.textChanges(phone);
```

```
        Observable<CharSequence> ObservablePassword =
RxTextView.textChanges(password);

        Observable.combineLatest(ObservablePhone, ObservablePassword, new
BiFunction<CharSequence,CharSequence,ValidationResult>() {

            @Override
            public ValidationResult apply(@NonNull CharSequence o1, @NonNull
CharSequence o2) throws Exception {

                if (o1.length()>0 || o2.length()>0) {

                    login.setBackgroundDrawable(getResources().getDrawable(R.
drawable.shape_login_pressed));
                } else {

                    login.setBackgroundDrawable(getResources().getDrawable(R.
drawable.shape_login_normal));
                }

                ValidationResult result = new ValidationResult();

                if (o1.length()==0) {

                    result.flag = false;
                    result.message = "手机号码不能为空";
                } else if (o1.length()!=11) {

                    result.flag = false;
                    result.message = "手机号码需要 11 位";
                } else if (o1 !=null && !AppUtils.isPhoneNumber(o1.toString()))
{

                    result.flag = false;
                    result.message = "手机号码需要数字";
                } else if(o2.length()==0) {

                    result.flag = false;
                    result.message = "密码不能为空";
                }

                return result;
```

```
            }
        }).subscribe(new Consumer<ValidationResult>() {

            @Override
            public void accept(@NonNull ValidationResult r) throws Exception {

                result = r;
            }
        });

        RxView.clicks(login)
                .compose(RxUtils.useRxViewTransformer(TestEditTextActivity.th
is))
                .subscribeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(@NonNull Object o) throws Exception {

                        if (result==null) return;

                        if (result.flag) {

                            Toast.makeText(TestEditTextActivity.this,"模拟登录成
功",Toast.LENGTH_SHORT).show();
                        } else if (Preconditions.isNotBlank(result.message)) {

                            Toast.makeText(TestEditTextActivity.this,result.me
ssage,Toast.LENGTH_SHORT).show();
                        }
                    }
                });
```

在示例代码中，我们使用了 RxTextView.textChanges()对 EditText 控件实现监听。相当于对 EditText 实现如下代码：

```
phone.addTextChangedListener(new TextWatcher() {
    @Override
    public void beforeTextChanged(CharSequence s, int start, int count, int after)
{
```

```
    }

    @Override
    public void onTextChanged(CharSequence s, int start, int before, int count)
{
        // do something ...
    }

    @Override
    public void afterTextChanged(Editable s) {

    }
});
```

combineLatest 的作用是将多个 Observable 发射的数据组装起来然后再发射出来。这里两个输入框只要内容发生变化，就会发送 Observable，此时我们即可在 BiFunction 中利用验证方法去判断输入框中最新的内容，最终返回一个 ValidationResult 对象。

```
/**
 * Created by tony on 2017/9/18.
 */

public class ValidationResult {

    public boolean flag;
    public String message;

    public ValidationResult() {

        flag = true;
        message = "";
    }
}
```

ValidationResult 的 flag 反映了表单验证的结果，flag 为 true 时表示验证成功，flag 为 false 时表示验证失败。如果验证失败，则 message 会显示失败的原因。点击“登录”按钮时，会先验证 ValidationResult 的值，如果表单验证成功，则显示“模拟登录成功”；如果表单验证失败，则显示 ValidationResult 的 message。

5. 获取验证码倒计时

用户注册账号时，一般需要获取验证码来验证手机号码，如图 11-7 所示。

在等待验证码的过程中，App 的界面上通常会有一个倒计时，提示我们剩余 xx 秒可以重新获取验证码，如图 11-8 所示。

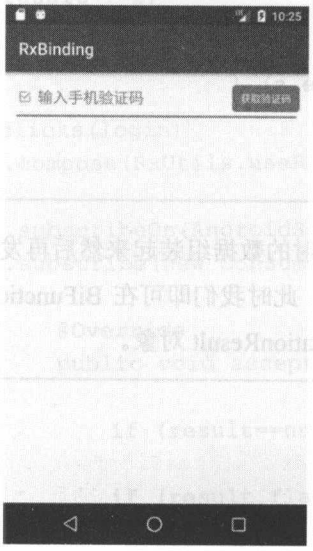


图 11-7

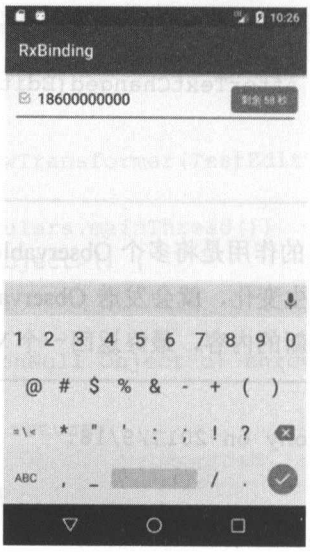


图 11-8

示例代码：

```
    long MAX_COUNT_TIME = 60;

     RxView.clicks(verificationCode)
            .throttleFirst(MAX_COUNT_TIME, TimeUnit.SECONDS)
            .flatMap(new Function<Object, ObservableSource<Long>>() {
                @Override
                public ObservableSource<Long> apply(Object o) throws
Exception {
                    //更新发送按钮的状态，并初始化显现倒计时文字
                    RxView.enabled(verificationCode).accept(false);
                    RxTextView.text(verificationCode).accept("剩余 " +
MAX_COUNT_TIME + " 秒");

                    // do something
```

```
                    //返回 n 秒内的倒计时观察者对象
                    return Observable.interval(1, TimeUnit.SECONDS,
Schedulers.io()).take(MAX_COUNT_TIME);
                }
            })
            .map(new Function<Long, Long>() { //将递增数字替换成递减的倒计时数
//字
                @Override
                public Long apply(Long aLong) throws Exception {
                    return MAX_COUNT_TIME - (aLong + 1);
                }
            })
            .observeOn(AndroidSchedulers.mainThread())//切换到 Android 的主
//线程。
            .subscribe(new Consumer<Long>() {
                @Override
                public void accept(Long aLong) throws Exception {

                    if (aLong == 0) {
                        RxView.enabled(verificationCode).accept(true);
                        RxTextView.text(verificationCode).accept("获取验证码
");
                    } else {
                        RxTextView.text(verificationCode).accept("剩余 " +
aLong + " 秒");
                    }

                }
            });
```

Observable.interval(1, TimeUnit.SECONDS, Schedulers.io())表示每 1s 发射一次数据。take(MAX_COUNT_TIME)和后面的操作表示按钮在 60s 内不可再次被点击，并且在这段时间内每隔一秒发射一次数据用于更新 UI。在实际使用中，需要在 flatMap 里做获取短信验证码的网络请求。

6. 对 RecyclerView 的支持

RxBinding 提供了一个 rxbinding-recyclerview-v7 的库，专门用于对 RecyclerView 的支持。

其中，RxRecyclerView 提供了几个状态的观察：

◎ scrollStateChanges 观察 RecyclerView 的滚动状态。
◎ scrollEvents 观察 RecyclerView 的滚动事件。
◎ childAttachStateChangeEvents 观察 child view 的 detached 状态，当 LayoutManager 或者 RecyclerView 认为不再需要一个 child view 时，就会调用这个方法。如果 child view 占用资源，则应当释放资源。

下面以观察 RecyclerView 的滚动状态为例，如图 11-9 所示。

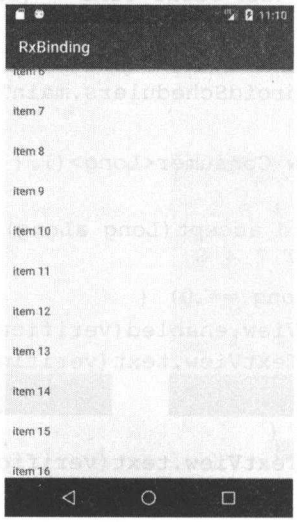


图 11-9

示例代码：

```
RxRecyclerView
        .scrollStateChanges(recyclerview)
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer scrollState) throws Exception {

                L.i("scrollState = " + scrollState);
            }
        });
```

可以看到 logcat 输出了如图 11-10 所示的日志。

```
rk.study.rxbinding D/EGL_emulation: eglMakeCurrent: 0xae654600: ver 2 0 (tinfo 0xae652680)
rk.study.rxbinding E/Surface: getSlotFromBufferLocked: unknown buffer: 0xaa09b040
rk.study.rxbinding I/SAF_L:
                    Thread: main
                    com.saframework.study.rxbinding.activity.TestRecyclerViewActivity$1.accept  (TestRecyclerViewActivity.java:54)
                    scrollState = 1
rk.study.rxbinding I/SAF_L:
                    Thread: main
                    com.saframework.study.rxbinding.activity.TestRecyclerViewActivity$1.accept  (TestRecyclerViewActivity.java:54)
                    scrollState = 2
rk.study.rxbinding I/SAF_L:
                    Thread: main
                    com.saframework.study.rxbinding.activity.TestRecyclerViewActivity$1.accept  (TestRecyclerViewActivity.java:54)
                    scrollState = 0
```

图 11-10

scrollState 表示 RecyclerView 中定义的滚动状态。

```
public static final int SCROLL_STATE_IDLE = 0; // RecyclerView 当前没有滚动

public static final int SCROLL_STATE_DRAGGING = 1; // RecyclerView 正在被拖动

public static final int SCROLL_STATE_SETTLING = 2; // 手已经离开屏幕，RecyclerView
// 正在做动画移动到最终位置
```

除可以对 RecyclerView 状态的进行监听外，还能对点击事件进行监听，如图 11-11 所示。

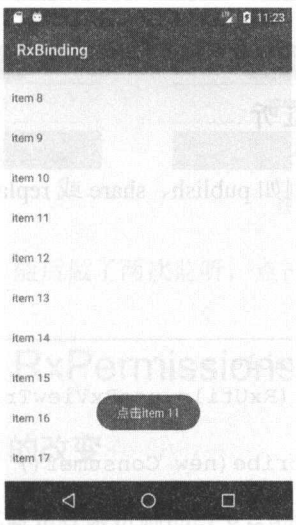


图 11-11

在 Adapter 的 onBindViewHolder()中，可以使用 clicks()来绑定 itemView 的点击事件。

```
@Override
public void onBindViewHolder(ViewHolder holder, final int position) {

    final String item = mDatas.get(position);

    if (Preconditions.isNotBlank(item)) {

        holder.text.setText(item);

        RxView.clicks(holder.itemView)
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(Object o) throws Exception {

                        Toast.makeText(mContext,item,Toast.LENGTH_SHORT).s
how();
                    }
                });
    }

}
```

如果点击了某一个 item，则会弹出相应的 toast。

7. 对 UI 控件进行多次监听

可以利用 RxJava 的操作符，例如 publish、share 或 replay，实现对 UI 控件的多次监听，如图 11-12 和图 11-13 所示。

示例代码：

```
    Observable clickObservable =
RxView.clicks(text7).compose(RxUtils.useRxViewTransformer(MainActivity.this
)).share();
    clickObservable.subscribe(new Consumer() {
        @Override
        public void accept(Object o) throws Exception {
```

```
                Toast.makeText(MainActivity.this,"对 text7 的第一次监听
",Toast.LENGTH_SHORT).show();
            }
        });

        clickObservable.subscribe(new Consumer() {
            @Override
            public void accept(Object o) throws Exception {

                Toast.makeText(MainActivity.this,"对 text7 的第二次监听
",Toast.LENGTH_SHORT).show();
            }
        });
```

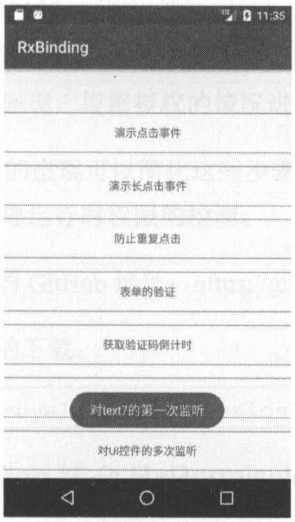


图 11-12

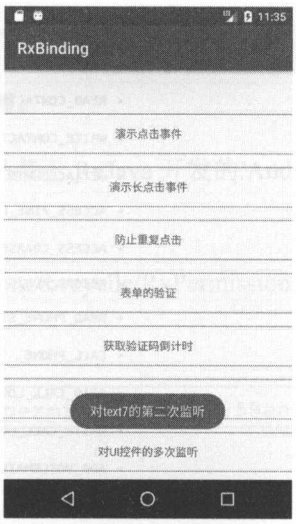


图 11-13

text7 控件使用了 share 操作符，随后做了两次监听，点击该控件后分别显示两个 toast。

11.3　RxBinding 结合 RxPermissions 的使用

11.3.1　Android 6.0 之后权限的改变

Android 6.0 带来一个很大 变化就是权限机制的改变，特别是运行时权限。

Android 6.0+添加的运行时权限可分为两类：

◎ Normal Permissions：这类权限不涉及个人隐私，不需要用户授权，比如手机震动、访问网络等。

◎ Dangerous Permissions：这类权限涉及个人隐私，需要用户授权，比如读取 SD 卡、访问通讯录等。

如图 11-14 所示，显示了 Android 的危险权限和权限组。Dangerous Permissions 是有分组的。App 运行在 Android 6.0+的手机之上，如果用户申请了某个 Dangerous Permissions，而该用户已经授权了一个与他现在申请的是同一组的 Dangerous Permissions，那么系统会自动授权，无须用户再次授权。

权限组	权限
CALENDAR	• READ_CALENDAR • WRITE_CALENDAR
CAMERA	• CAMERA
CONTACTS	• READ_CONTACTS • WRITE_CONTACTS • GET_ACCOUNTS
LOCATION	• ACCESS_FINE_LOCATION • ACCESS_COARSE_LOCATION
MICROPHONE	• RECORD_AUDIO
PHONE	• READ_PHONE_STATE • CALL_PHONE • READ_CALL_LOG • WRITE_CALL_LOG • ADD_VOICEMAIL • USE_SIP • PROCESS_OUTGOING_CALLS
SENSORS	• BODY_SENSORS
SMS	• SEND_SMS • RECEIVE_SMS • READ_SMS • RECEIVE_WAP_PUSH • RECEIVE_MMS
STORAGE	• READ_EXTERNAL_STORAGE • WRITE_EXTERNAL_STORAGE

图 11-14

对于 Android 6.0 以下的手机，用户在安装 App 的时候可以看到权限声明产生一个权限列表，用户只有在同意之后才能完成 App 的安装。如果用户想要使用某个 App，就需要忍受其一些不必要的权限（例如访问通讯录、短信的权限等）。从 Android 6.0 以后我们可以直接安装 App，当 App 需要我们授予不恰当的权限的时候，我们可以予以拒绝。当然作为用户也可以在手机的设置界面里对每个 App 的权限进行查看，并对单个权限进行授权或者解除授权。

值得注意的是，App 的 targetSdkVersion 是 23 及以上，并且 App 运行在 Android 6.0 及以上的设备时，需同时满足这两个条件才需要动态地请求危险权限。

11.3.2　RxPermission 的介绍

在处理运行时权限时，通常需要两步：

◎　申请权限；
◎　处理权限回调，根据授权的情况进行回调。

RxPermissions 的出现可以简化这些步骤，它是基于 RxJava 开发的 Android 框架，旨在帮助 Android 6.0 之后处理运行时权限的检测。

RxPermission 的 GitHub 地址：https://github.com/tbruyelle/RxPermissions

RxPermission 的下载：

```
compile 'com.tbruyelle.rxpermissions2:rxpermissions:0.9.5@aar'
```

11.3.3　RxBinding 结合 RxPermissions

在 RxPermission 使用之前，需要先创建 RxPermissions 的实例。可以在 Activity 的 onCreate() 中进行创建，创建之后才能使用它：

```
RxPermissions rxPermissions = new RxPermissions(this);
```

1. 在 RxBinding 中使用 RxPermissions

举一个拨打电话的例子，CALL_PHONE 在 Android 6.0 之后是一个 Dangerous Permissions，第一次使用时需要动态申请该权限，只有得到允许才能完成后面打电话的动作。

下面看一个结合 RxBinding 来使用 RxPermissions 的例子。

如图 11-15 所示，单击“演示打电话 1”按钮之后，会出现一个弹框，让用户选择是否授予权限。如果选择允许，就可以完成后面打电话的过程，如图 11-16 所示。如果不允许，则下次再点击“演示打电话 1”按钮，还会出现需要授权的对话框。

```
        RxView.clicks(text1)
                .subscribe(new Consumer<Object>() {
                    @Override
                    public void accept(@NonNull Object o) throws Exception {

                        rxPermissions.request(Manifest.permission.CALL_PHONE)
                                .subscribe(new Consumer<Boolean>() {
                                    @Override
                                    public void accept(Boolean granted) throws
Exception {

                                        if (granted) {

                                            Intent intent = new
Intent(Intent.ACTION_CALL);
                                            intent.setData(Uri.parse("tel:" +
"10000"));
                                            startActivity(intent);
                                        } else {

                                            L.i("授权失败");
                                        }
                                    }
                                });
                    }
                });
```

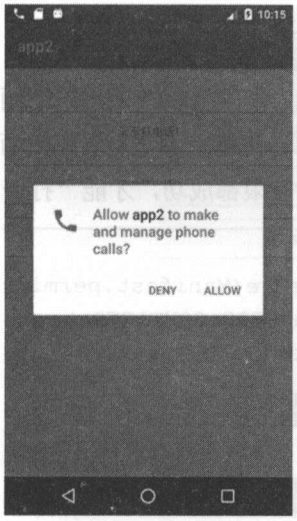


图 11-15

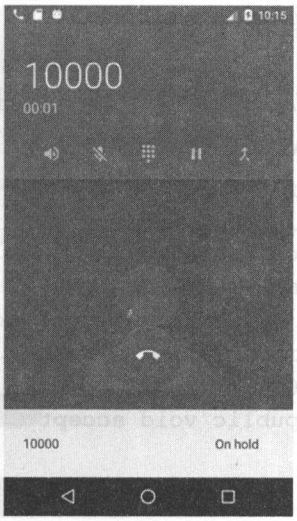


图 11-16

2. RxBinding 结合 compose，使用 RxPermissions

对上述的代码稍作修改，RxBinding 可以结合 compose 操作符来使用 RxPermissions。本书 9.3 节曾介绍过 compose 操作符。

```
    RxView.clicks(text2)
            .compose(rxPermissions.ensure(Manifest.permission.CALL_PHONE))
            .subscribe(new Consumer<Boolean>() {

                @Override
                public void accept(@NonNull Boolean granted) throws Exception
{

                    if (granted) {

                        Intent intent = new Intent(Intent.ACTION_CALL);
                        intent.setData(Uri.parse("tel:" + "10000"));
                        startActivity(intent);
                    } else {

                        L.i("授权失败");
                    }
                }
            });
```

3. 使用多个权限的用法

RxPermissions 也支持申请多个权限，如图 11-17 至 11-19 所示，下面的例子展示了同时申请 CAMERA 和 READ_CONTACTS 的权限。单击按钮之后，需要授权两次，任何一次授权的失败都会导致“打开相机失败”。只有两次申请权限都成功，才能“打开相机成功”。

```
        RxView.clicks(text3)
                .compose(rxPermissions.ensure(Manifest.permission.CAMERA,
                        Manifest.permission.READ_CONTACTS))
                .subscribe(new Consumer<Boolean>() {
                    @Override
                    public void accept(Boolean granted) throws Exception {

                        if (granted) {

                            Toast.makeText(MainActivity.this,"打开相机成功
",Toast.LENGTH_SHORT).show();
                        } else {

                            L.i("授权失败");
                            Toast.makeText(MainActivity.this,"打开相机失败
",Toast.LENGTH_SHORT).show();
                        }
                    }
                });
```

图 11-17

图 11-18

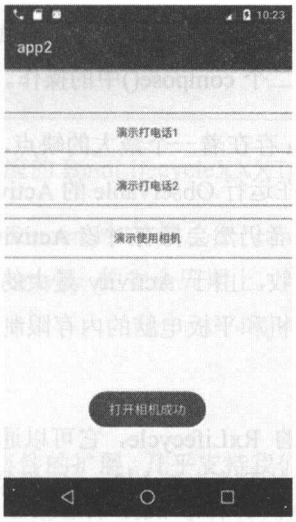


图 11-19

11.4　RxBinding 使用的注意点

在 11.2 节的第三个使用场景中，我们使用了 useRxViewTransformer 方法。

```
    /**
     * 对 RxView 绑定的事件
     * 封装了防止重复点击和 RxLifecycle 的生命周期
     */
    public static ObservableTransformer useRxViewTransformer(final
AppCompatActivity targetActivity) {

        return new ObservableTransformer() {

            @Override
            public ObservableSource apply(Observable upstream) {
                return
upstream.compose(RxJavaUtils.preventDuplicateClicksTransformer())
                        .compose(RxLifecycle.bind(targetActivity).toLifecycle
Transformer());
            }
        };
    }
```

这个方法封装了两个操作：一个是防止 UI 事件的重复点击，另一个是绑定 Activity 的生命周期，防止内存泄露。现在讲解第二个 compose()中的操作。

在 Android App 中使用 RxJava 存在着一个最大的缺点，即不完整的订阅会导致内存泄漏。当 Android 系统尝试销毁包含着正在运行 Observable 的 Activity/Fragment 时，会发生内存泄漏。由于 Observable 正在运行，其观察者仍然会持有对该 Activity/Fragment 的引用，因此系统无法对此 Activity/Fragment 进行垃圾回收。由于 Activity 是大的对象，因此可能会导致严重的内存管理问题，尤其是 Android 智能手机和平板电脑的内存限制，大量内存泄漏和有限内存的组合可能会导致 Out Of Memory。

因此，Trello 开发了大名鼎鼎的 RxLifecycle，它可以通过绑定生命周期的方式，来解决内存泄漏的问题。在本书的第 9 章曾介绍过此 RxLifecycle 框架。

GitHub 地址：https://github.com/trello/RxLifecycle

后来，知乎也开发了一款类似的 RxLifecycle，它允许你仅用一句话绑定你的 Observable 到 Activity/Fragment 的生命周期上。知乎的 RxLifecycle 与最初 Trello 开发的 RxLifecycle 相比，在使用上更加简单。

GitHub 地址：https://github.com/zhihu/RxLifecycle

笔者参考知乎的库，做了一个类似功能的库，目前该库已经在生产环境上使用。

GitHub 地址：https://github.com/fengzhizi715/SAF/tree/master/saf-rxlifecycle

下载：

```
compile 'com.safframework:saf-rxlifecycle:1.0.0'
```

与知乎的 RxLifecycle 的区别是，LifecycleTransformer 实现了多个 Transformer 接口。

```
/**
 * Created by Tony Shen on 2017/5/25.
 */

public class LifecycleTransformer<T> implements ObservableTransformer<T, T>,
        FlowableTransformer<T, T>,
        SingleTransformer<T, T>,
        MaybeTransformer<T, T>,
```

```
        CompletableTransformer {
        ......
}
```

而知乎的 RxLifecycle 有多个对应的 BindLifecycleXXXTransformer。

所以，前面的 useRxViewTransformer()中的第二个 compose，实际上使用了笔者自己的 RxLifecycle 来绑定 Observable 到 Activity 的生命周期上，防止出现内存泄露。

11.5　小结

RxBinding 是对 Android View 事件的扩展，几乎支持我们常用的所有控件及事件（v4、v7、design、recyclerview 等）。RxBinding 把各种 View 事件转为 Observable 事件流，使得开发者能够对 View 事件使用 RxJava 的各种操作，减少了开发者的代码量。

但是在使用 RxBinding 时，最好能够结合 RxLifecycle 之类的框架一起使用，防止出现内存泄露，从而让我们的代码更加健壮。

第12章

RxAndroid 2.x和Retrofit的使用

12.1 RxAndroid 2.x 简介

1. 介绍

近几年 RxJava 逐渐成为 Android 开发的新宠，越来越多的 Android 开发者正在使用或者即将使用 RxJava。要想在 Android 上使用 RxJava，RxAndroid 必不可少。

RxAndroid 的 GitHub 地址：https://github.com/ReactiveX/RxAndroid。

RxAndroid 也隶属于 ReactiveX 组织，它是 RxJava 在 Android 上的一个扩展。从其 GitHub 的官方主页上，可以了解到 RxAndroid 提供了一个调度程序，能够切换到 Android 主线程或者任意指定的 Looper。

RxAndroid 不能单独使用，它只有依赖 RxJava 才能使用。注意，选择所依赖的 RxJava 版本时最好用最新的版本，因为新版本会修复之前版本的 Bug 并且提供一些新的特性。

RxAndroid 的下载：

```
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
compile 'io.reactivex.rxjava2:rxjava:2.1.9'
```

2. 使用

（1）AndroidSchedulers.mainThread()

在本书的第 4 章曾经讲过 RxJava 的调度器，在 Android 中使用时需要新增一个调度器，用于将指定的操作切换到 Android 的主线程中运行，方便做一些更新 UI 的操作。RxAndroid 就提供了这样的调度器。

下面的例子展示了从网上获取图片然后将图片转换成 Bitmap，最后显示到 ImageView 上的过程。这里没有使用 Android 的 AsyncTask 或 Handler，而是通过 subscribeOn、observeOn 不断地切换线程来达到目的的。

```
        Observable.create(new ObservableOnSubscribe<Bitmap>() {

            @Override
            public void subscribe(ObservableEmitter<Bitmap> e) throws Exception
{

                L.i("e.onNext()");
                e.onNext(getBitmap());
            }
        })
                //设置数据加载在子线程中进行
                .subscribeOn(Schedulers.io())
                //设置图片加载在主线程中进行
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Bitmap>() {
                    @Override
                    public void accept(Bitmap bitmap) throws Exception {

                        if (bitmap != null) {
                            L.i("bitmap is not null");
                            imageView.setImageBitmap(bitmap);
                        }
                    }
                });
```

getBitmap()方法显示了通过 URL 来请求网络图片，并将 HttpURLConnection 获取的 InputStream 转换成 Bitmap。

```
    private Bitmap getBitmap() {
        HttpURLConnection con;

        try {
            URL url = new
URL("http://www.designerspics.com/wp-content/uploads/2014/09/grass_shrubs_2
_free_photo.jpg");
            con = (HttpURLConnection) url.openConnection();
            con.setConnectTimeout(20000);
            con.connect();
            if (con.getResponseCode() == 200) {
                return BitmapFactory.decodeStream(con.getInputStream());
            }
        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        return null;
    }
```

图 12-1 打印了不同操作的日志，在日志中会反映出当前方法所在的线程的名字。

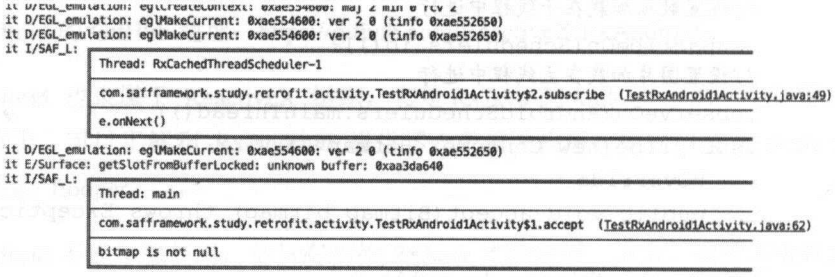


图 12-1

通过日志可以看到“e.onNext()”是在 iO()线程中打印出来的，也就是说，getBitmap()也是在 iO()线程中运行的。而“bitmap is not null”是在主线程中打印出来的，说明 Bitmap 更新到 ImageView 时已经切换到 UI 线程了。此时切换的过程，是通过下面这句代码来实现的。

```
observeOn(AndroidSchedulers.mainThread())
```

AndroidSchedulers 类似于 RxJava 的 Schedulers 类，提供了创建 Scheduler 的工厂方法。

最后，图片会展示到 ImageView 控件上，如图 12-2 所示。

图 12-2

（2）AndroidSchedulers.from(Looper looper)

RxAndroid 除了可以切换到主线程，还可以使用任意指定的 Looper。

下面的例子是对刚才的代码做了一些改动，Observable 创建之后先发射图片的 URL，然后再在 map 操作中获取图片的 Bitmap，最后将 Bitmap 显示到 ImageView 上。

```
   private final static String IMAGE_URL =
"http://n.sinaimg.cn/news/1_img/upload/60c98dca/20170929/KSE9-fymkwwk6862900.jpg";

    .....

    Observable.create(new ObservableOnSubscribe<String>() {

            @Override
            public void subscribe(ObservableEmitter<String> e) throws Exception
{

                L.i("IMAGE_URL="+IMAGE_URL);
                e.onNext(IMAGE_URL);
```

```
        }
    })
            .subscribeOn(AndroidSchedulers.from(new
Handler().getLooper()))
            .observeOn(Schedulers.io())
            .map(new Function<String, Bitmap>() {

                @Override
                public Bitmap apply(String s) throws Exception {

                    L.i("s="+s);
                    return getBitmap(s);
                }
            })
            //设置图片加载在主线程中进行
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<Bitmap>() {
                @Override
                public void accept(Bitmap bitmap) throws Exception {

                    if (bitmap != null) {
                        L.i("bitmap is not null");
                        imageView.setImageBitmap(bitmap);
                    }
                }
            });
```

图 12-3 仍然显示了不同操作的日志。

```
D/EGL_emulation: eglMakeCurrent: 0xae594600: ver 2 0 (tinfo 0xae592680)
E/Surface: getSlotFromBufferLocked: unknown buffer: 0xaa21a8c0
I/SAF_L:
    Thread: main
    com.saframework.study.retrofit.activity.TestRxAndroid2Activity$3.subscribe  (TestRxAndroid2Activity.java:58)
    IMAGE_URL=http://www.designerspics.com/wp-content/uploads/2014/09/grass_shrubs_2_free_photo.jpg
I/SAF_L:
    Thread: RxCachedThreadScheduler-2
    com.saframework.study.retrofit.activity.TestRxAndroid2Activity$2.apply  (TestRxAndroid2Activity.java:69)
    s=http://www.designerspics.com/wp-content/uploads/2014/09/grass_shrubs_2_free_photo.jpg
D/EGL_emulation: eglMakeCurrent: 0xae594600: ver 2 0 (tinfo 0xae592680)
E/Surface: getSlotFromBufferLocked: unknown buffer: 0xaa21aaa0
I/SAF_L:
    Thread: main
    com.saframework.study.retrofit.activity.TestRxAndroid2Activity$1.accept  (TestRxAndroid2Activity.java:80)
    bitmap is not null
```

图 12-3

在代码片段中打印了三次日志：第一次日志是在主线程中，第二次日志显示了 getBitmap() 已经切换到 iO()线程中使用了，最后一次打印表明此时又切回主线程了。

12.2　Retrofit 简介

Retrofit 是一个在 Android 开发中非常流行的网络框架，底层依赖 OkHttp。Retrofit 和 OkHttp 都出自 Square 的技术团队。

Retrofit 的 GitHub 地址：https://github.com/square/retrofit。

应用程序通过 Retrofit 请求网络，实际上是使用 Retrofit 接口层封装请求参数、Header、URL 等信息，之后由 OkHttp 完成后续的请求操作，在服务端返回数据之后，OkHttp 将原始的结果交给 Retrofit，Retrofit 再根据用户的需求对结果进行解析的过程。

Retrofit 支持大多数的 Http 方法。

Retrofit 的特点如下。

（1）Retrofit 是可插拔的，允许不同的执行机制及其库用于执行 http 调用。允许 API 请求，与应用程序其余部分中任何现有线程模型和/或任务框架无缝组合。

Retrofit 为常见的框架提供了适配器（Adapter）：

- RxJava1.x Observable & Single - com.squareup.retrofit2:adapter-rxjava
- RxJava2.x Observable, Flowable, Single, Completable & Maybe -com.squareup.retrofit2:adapter-rxjava2
- Guava ListenableFuture - com.squareup.retrofit2:adapter-guava
- Java 8 CompletableFuture - com.squareup.retrofit2:adapter-java8

（2）允许不同的序列化格式及其库，用于将 Java 类型转换为其 http 表示形式，并将 http 实体解析为 Java 类型。

Retrofit 为常见的序列化格式提供了转换器（Converter）：

- Gson - com.squareup.retrofit2:converter-gson。
- Jackson - com.squareup.retrofit2:converter-jackson。
- Moshi - com.squareup.retrofit2:converter-moshi。

- Protobuf - com.squareup.retrofit2:converter-protobuf。
- Wire - com.squareup.retrofit2:converter-wire。
- Simple Framework - com.squareup.retrofit2:converter-simplexml。
- Scalars - com.squareup.retrofit2:converter-scalars。

开源社区也已经为其他库和序列化格式创建了各种第三方转换器（Converter）：

- LoganSquare - com.github.aurae.retrofit2:converter-logansquare。
- FastJson - org.ligboy.retrofit2:converter-fastjson 和 org.ligboy.retrofit2:converter-fastjson-经 android。

OkHttp 的特点如下。

- 支持 HTTP2/SPDY 黑科技。
- socket 自动选择最优路线，并支持自动重连。
- 拥有自动维护的 socket 连接池，减少握手次数。
- 拥有队列线程池，轻松写并发。
- 拥有 Interceptors 轻松处理请求与响应（比如透明 GZIP 压缩、LOGGING）。
- 基于 Headers 的缓存策略。

在实际开发中使用 Retrofit 时，一般都会对 OkHttp 做一些定制化的改动，以满足实际业务的需求。

12.3 Retrofit 与 RxJava 的完美配合

Retrofit 是一个网络框架，如果想尝试响应式的编程方式，则可以结合 RxJava 一起使用。Retrofit 对 RxJava 1 和 RxJava 2 都提供了 Adapter。

下面会结合一个例子来讲解 Retrofit 和 RxJava 2 在 Android 上的使用。这个例子是将苏州市南门地区的 PM2.5、PM10、SO2 的数据展示到 App 上。在 http://pm25.in/上可以找到获取这些数据的接口，Pm25.in 是一个公益性的网站，免费提供空气质量数据。在调用这些接口之前，需要去该网站注册，并申请一个 AppKey。

Retrofit 使用步骤如下。

第一步　添加 Retrofit 依赖。

在 App 的 build.gradle 中添加所需要的 Retrofit 库，以及 RxJava2 的 adapter 库。

```
    compile 'com.squareup.retrofit2:retrofit:2.3.0'
    compile 'com.squareup.retrofit2:adapter-rxjava2:2.3.0'
```

第二步　创建 RetrofitManager。

一般需要创建一个 Retrofit 的管理类，在这里创建一个名为 RetrofitManager 的类，方便在整个 App 中使用。

RetrofitManager 的代码如下：

```
package com.saframework.study.retrofit.http;

import com.saframework.http.interceptor.LoggingInterceptor;
import com.saframework.study.retrofit.api.APIService;

import java.util.concurrent.TimeUnit;

import okhttp3.OkHttpClient;
import retrofit2.Retrofit;
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory;
import retrofit2.converter.fastjson.FastJsonConverterFactory;

/**
 * Created by tony on 2017/9/29.
 */

public class RetrofitManager {

    private static Retrofit mRetrofit;

    public static Retrofit retrofit() {
        if (mRetrofit == null) {
            OkHttpClient.Builder builder = new OkHttpClient.Builder();
            builder.writeTimeout(30 * 1000, TimeUnit.MILLISECONDS);
            builder.readTimeout(20 * 1000, TimeUnit.MILLISECONDS);
            builder.connectTimeout(15 * 1000, TimeUnit.MILLISECONDS);
```

```
            LoggingInterceptor loggingInterceptor = new
LoggingInterceptor.Builder()
                    .loggable(true)
                    .request()
                    .requestTag("Request")
                    .response()
                    .responseTag("Response")
                    .build();

            //设置拦截器
            builder.addInterceptor(loggingInterceptor);

            OkHttpClient okHttpClient = builder.build();
            mRetrofit = new Retrofit.Builder()
                    .baseUrl(APIService.API_BASE_SERVER_URL)
                    .addConverterFactory(FastJsonConverterFactory.create())
                    .addCallAdapterFactory(RxJava2CallAdapterFactory.create()
)
                    .client(okHttpClient)
                    .build();
        }
        return mRetrofit;
    }
}
```

可以看到在 RetrofitManager 中对 OkHttp 添加了日志拦截器。它是用于记录 OKHttp 网络请求的日志的拦截器，完全用 Kotlin 语言编写。

GitHub 地址：https://github.com/fengzhizi715/saf-logginginterceptor。

在 RetrofitManager 中还添加了 fastjson 的转换器，在网络框架中可以使用 fastjson 来解析返回的 response 对象。

在 build.gradle 中添加 converter-fastjson：

```
compile 'org.ligboy.retrofit2:converter-fastjson-android:2.1.0'
```

第三步　创建 APIService。

接下来，我们需要定义网络请求的接口。Pm25.in 提供了多个获取空气质量数据的接口，这里选取其中 3 个接口，分别是获取一个城市所有监测点的 PM2.5 数据接口、获取一个城市所

有监测点的 PM10 数据接口、获取一个城市所有监测点的 SO2 数据接口。

```
package com.saframework.study.retrofit.api;

import com.saframework.study.retrofit.model.PM10Model;
import com.saframework.study.retrofit.model.PM25Model;
import com.saframework.study.retrofit.model.SO2Model;

import java.util.List;

import io.reactivex.Maybe;
import retrofit2.http.GET;
import retrofit2.http.Query;

/**
 * Created by tony on 2017/9/29.
 */

public interface APIService {

    String API_BASE_SERVER_URL = "http://www.pm25.in/";

    @GET("api/querys/pm2_5.json")
    Maybe<List<PM25Model>> pm25(@Query("city") String cityId, @Query("token")
String token);

    @GET("api/querys/pm10.json")
    Maybe<List<PM10Model>> pm10(@Query("city") String cityId, @Query("token")
String token);

    @GET("api/querys/so2.json")
    Maybe<List<SO2Model>> so2(@Query("city") String cityId, @Query("token")
String token);
}
```

在 APIService 中，每个方法返回的类型都是 Maybe 类型，其实也可以返回 Observable、Flowable 等类型，但是在 2.4 节里曾介绍过 Maybe 的特性，所以这里特意使用 Maybe。

注意，这里如果不使用 RxJava，而是使用 Java 8，并且使用 Java 8 的适配器，那么在 APIService 中，每个方法返回的类型都应该是 CompletableFuture。CompletableFuture 会在本书的第 16 章中讲解到。

第四步　Retrofit 的使用。

做完上述步骤后，就可以在 App 中使用 Retrofit 了，下面的代码分别调用了 3 个接口，并过滤出了南门地区的相关数据。

```
        APIService apiService =
RetrofitManager.retrofit().create(APIService.class);

        apiService.pm25(Constant.CITY_ID,Constant.TOKEN)
                .compose(RxJavaUtils.<List<PM25Model>>maybeToMain())
                .filter(new Predicate<List<PM25Model>>() {
                    @Override
                    public boolean test(List<PM25Model> pm25Models) throws
Exception {

                        return Preconditions.isNotBlank(pm25Models);
                    }
                })
                .flatMap(new Function<List<PM25Model>,
MaybeSource<PM25Model>>() {
                    @Override
                    public MaybeSource<PM25Model> apply(List<PM25Model>
pm25Models) throws Exception {

                        for (PM25Model model:pm25Models){

                            if ("南门".equals(model.position_name)) {
                                return Maybe.just(model);
                            }
                        }

                        return null;
                    }
                })
                .subscribe(new Consumer<PM25Model>() {
                    @Override
                    public void accept(PM25Model model) throws Exception {

                        if (model != null) {
                            quality.setText("空气质量指数: " + model.quality);
                            pm2_5.setText("PM2.5 1 小时内平均: " + model.pm2_5);
```

```
                        pm2_5_24h.setText("PM2.5 24 小时滑动平均：" +
model.pm2_5_24h);
                    }
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println(throwable.getMessage());
                }
            });

        apiService.pm10(Constant.CITY_ID,Constant.TOKEN)
                .compose(RxJavaUtils.<List<PM10Model>>maybeToMain())
                .filter(new Predicate<List<PM10Model>>() {
                    @Override
                    public boolean test(List<PM10Model> pm10Models) throws
Exception {
                        return Preconditions.isNotBlank(pm10Models);
                    }
                })
                .flatMap(new Function<List<PM10Model>,
MaybeSource<PM10Model>>() {
                    @Override
                    public MaybeSource<PM10Model> apply(List<PM10Model>
pm10Models) throws Exception {

                        for (PM10Model model:pm10Models){

                            if ("南门".equals(model.position_name)) {

                                return Maybe.just(model);
                            }
                        }

                        return null;
                    }
                })
                .subscribe(new Consumer<PM10Model>() {
                    @Override
                    public void accept(PM10Model model) throws Exception {

                        if (model!=null) {
```

```
                        pm10.setText("PM10 1小时内平均: "+model.pm10);
                        pm10_24h.setText("PM10 24小时滑动平均:
"+model.pm10_24h);
                    }
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println(throwable.getMessage());
                }
            });

        apiService.so2(Constant.CITY_ID,Constant.TOKEN)
                .compose(RxJavaUtils.<List<SO2Model>>maybeToMain())
                .filter(new Predicate<List<SO2Model>>() {
                    @Override
                    public boolean test(List<SO2Model> so2Models) throws
Exception {
                        return Preconditions.isNotBlank(so2Models);
                    }
                })
                .flatMap(new Function<List<SO2Model>, MaybeSource<SO2Model>>()
{
                    @Override
                    public MaybeSource<SO2Model> apply(List<SO2Model> so2Models)
throws Exception {

                        for (SO2Model model:so2Models){

                            if ("南门".equals(model.position_name)) {

                                return Maybe.just(model);
                            }
                        }

                        return null;
                    }
                })
                .subscribe(new Consumer<SO2Model>() {
                    @Override
```

```
                public void accept(SO2Model model) throws Exception {

                    if (model!=null) {

                        so2.setText("二氧化硫 1 小时平均："+model.so2);
                        so2_24h.setText("二氧化硫 24 小时滑动平均：
"+model.so2_24h);
                    }
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println(throwable.getMessage());
                }
            });
```

这里还使用了 maybeToMain()方法，它的代码如下：

```
@JvmStatic
fun <T> maybeToMain(): MaybeTransformer<T, T> {

    return MaybeTransformer{
        upstream ->
        upstream.subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
    }
}
```

它用于切换线程，返回 MaybeTransformer 对象。因为 apiService 中每个返回的方法都是 Maybe 类型，所以这里会用到 MaybeTransformer。使用了 maybeToMain()后，除网络请求是在 iO()线程中运行外，其余的操作都是在主线程中运行的。

其实，我们也可以对上面的代码做一些改动，让 filter、flatMap 操作也在 iO()线程中运行，展示数据时才切换回主线程。

```
    apiService.pm25(cityId,token)
//              .compose(RxJavaUtils.<List<PM25Model>>maybeToMain())
            .subscribeOn(Schedulers.io())
            .filter(new Predicate<List<PM25Model>>() {
                @Override
```

```
                public boolean test(List<PM25Model> pm25Models) throws
Exception {

                    return Preconditions.isNotBlank(pm25Models);
                }
            })
            .flatMap(new Function<List<PM25Model>, MaybeSource<PM25Model>>() {
                @Override
                public MaybeSource<PM25Model> apply(List<PM25Model>
pm25Models) throws Exception {

                    for (PM25Model model:pm25Models){

                        if ("南门".equals(model.position_name)) {

                            return Maybe.just(model);
                        }
                    }

                    return null;
                }
            })
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<PM25Model>() {
                @Override
                public void accept(PM25Model model) throws Exception {

                    if (model != null) {

                        quality.setText("空气质量指数: " + model.quality);
                        pm2_5.setText("PM2.5 1 小时内平均: " + model.pm2_5);
                        pm2_5_24h.setText("PM2.5 24 小时滑动平均: " +
model.pm2_5_24h);
                    }
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    System.out.println(throwable.getMessage());
                }
            });
```

最后，我们看到这些数据都显示在了 App 上，所有的网络请求都是异步进行的，如图 12-4 所示。

在第二步中曾提到过日志拦截器，下面就来看一下日志拦截器的效果，如图 12-5 所示。

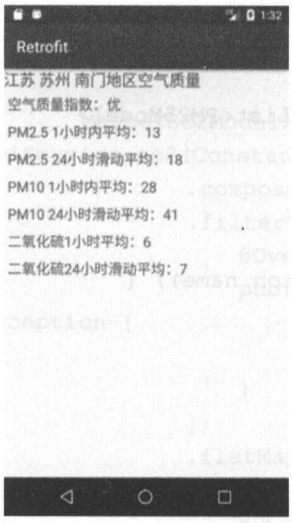


图 12-4

图 12-5

从 Request 中能显示出 http 的方法名、接口的 URL，如果是 POST 方法还会打印出 POST Body。在 Response 中，会显示调用接口花费的时间，枚举出 Header 的值，返回接口的具体内容。

下面列举一些 Retrofit 其余常见的使用场景。

（1）合并多个网络请求。

例如，需要在某一个信息流列表中插入多条广告，每一条广告都需要做一次网络请求。这时就可以考虑使用 zip 操作符，将请求信息流，以及请求的多个广告的请求合并起来，等所有请求完成之后，再用合并函数将广告插到信息流固定的位置上，最后以列表的形式呈现给用户。

或者针对本节的例子，也可以使用 zip 操作符来合并多个请求。

```
        Maybe<PM25Model> pm25Maybe =
apiService.pm25(Constant.CITY_ID,Constant.TOKEN)
                .compose(RxJavaUtils.<List<PM25Model>>maybeToMain())
                .filter(new Predicate<List<PM25Model>>() {
                    @Override
                    public boolean test(List<PM25Model> pm25Models) throws
Exception {

                        return Preconditions.isNotBlank(pm25Models);
                    }
                })
                .flatMap(new Function<List<PM25Model>,
MaybeSource<PM25Model>>() {
                    @Override
                    public MaybeSource<PM25Model> apply(List<PM25Model>
pm25Models) throws Exception {

                        for (PM25Model model:pm25Models){

                            if ("南门".equals(model.position_name)) {
                                return Maybe.just(model);
                            }
                        }

                        return null;
                    }
                });

        Maybe<PM10Model> pm10Maybe =
apiService.pm10(Constant.CITY_ID,Constant.TOKEN)
                .compose(RxJavaUtils.<List<PM10Model>>maybeToMain())
                .filter(new Predicate<List<PM10Model>>() {
                    @Override
```

```
                public boolean test(List<PM10Model> pm10Models) throws
Exception {
                    return Preconditions.isNotBlank(pm10Models);
                }
            })
            .flatMap(new Function<List<PM10Model>,
MaybeSource<PM10Model>>() {
                @Override
                public MaybeSource<PM10Model> apply(List<PM10Model>
pm10Models) throws Exception {

                    for (PM10Model model:pm10Models){

                        if ("南门".equals(model.position_name)) {

                            return Maybe.just(model);
                        }
                    }

                    return null;
                }
            });

        Maybe<SO2Model> so2Maybe =
apiService.so2(Constant.CITY_ID,Constant.TOKEN)
            .compose(RxJavaUtils.<List<SO2Model>>maybeToMain())
            .filter(new Predicate<List<SO2Model>>() {
                @Override
                public boolean test(List<SO2Model> so2Models) throws
Exception {
                    return Preconditions.isNotBlank(so2Models);
                }
            })
            .flatMap(new Function<List<SO2Model>, MaybeSource<SO2Model>>()
{
                @Override
                public MaybeSource<SO2Model> apply(List<SO2Model> so2Models)
throws Exception {

                    for (SO2Model model:so2Models){

                        if ("南门".equals(model.position_name)) {
```

```
                        return Maybe.just(model);
                    }
                }

                return null;
            }
        });
    // 合并多个网络请求
    Maybe.zip(pm25Maybe, pm10Maybe, so2Maybe, new Function3<PM25Model,
PM10Model, SO2Model, ZipObject>() {

        @Override
        public ZipObject apply(PM25Model pm25Model, PM10Model pm10Model,
SO2Model so2Model) throws Exception {

            ZipObject zipObject = new ZipObject();
            zipObject.pm2_5_quality = pm25Model.quality;
            zipObject.pm2_5 = pm25Model.pm2_5;
            zipObject.pm2_5_24h = pm25Model.pm2_5_24h;

            zipObject.pm10 = pm10Model.pm10;
            zipObject.pm10_24h = pm10Model.pm10_24h;

            zipObject.so2 = so2Model.so2;
            zipObject.so2_24h = so2Model.so2_24h;

            return zipObject;
        }
    }).subscribe(new Consumer<ZipObject>() {
        @Override
        public void accept(ZipObject zipObject) throws Exception {

            quality.setText("空气质量指数: " + zipObject.pm2_5_quality);
            pm2_5.setText("PM2.5 1小时内平均: " + zipObject.pm2_5);
            pm2_5_24h.setText("PM2.5 24小时滑动平均:" + zipObject.pm2_5_24h);

            pm10.setText("PM10 1小时内平均: "+zipObject.pm10);
            pm10_24h.setText("PM10 24小时滑动平均: "+zipObject.pm10_24h);

            so2.setText("二氧化硫1小时平均: "+zipObject.so2);
```

```
            so2_24h.setText("二氧化硫 24 小时滑动平均：" +zipObject.so2_24h);

        }
    }, new Consumer<Throwable>() {
        @Override
        public void accept(Throwable throwable) throws Exception {
            System.out.println(throwable.getMessage());
        }
    });
```

（2）返回默认值

有时，网络请求失败可以使用 onErrorReturn 操作符，返回一个空的对象作为默认值。

（3）多个网络请求嵌套使用

若是 A 请求完成之后，才能去调用 B 请求，则可以考虑使用 flatMap 操作符。

（4）重试机制

对于一些重要的接口，需要采用重试机制。因为有些时候用户的网络环境比较差，第一次请求接口超时了，那么再一次请求可能就会成功。虽然有一定的延迟，但至少返回了数据，保证了用户体验。

```
apiService.loadContent(param)
            .retryWhen(new RetryWithDelay(3,1000))
            .compose(RxLifecycle.bind(fragment).<ContentModel>toLifecycle
Transformer())
```

在这里 retryWhen 操作符与 RetryWithDelay 一起搭配使用，表示有 3 次重试机会，每次的延迟时间是 1000 ms。RetryWithDelay 是笔者的工具类，用 Kotlin 语言编写的。

```
import android.util.Log
import io.reactivex.Flowable
import io.reactivex.annotations.NonNull
import io.reactivex.functions.Function
import org.reactivestreams.Publisher
import java.util.concurrent.TimeUnit

/**
 * 重试机制，与 retryWhen 操作符搭配使用
 * Created by tony on 2017/11/6.
```

```
 */

class RetryWithDelay(private val maxRetries: Int, private val retryDelayMillis:
Int) : Function<Flowable<out Throwable>, Publisher<*>> {

    private var retryCount: Int = 0

    init {
        this.retryCount = 0
    }

    @Throws(Exception::class)
    override fun apply(@NonNull attempts: Flowable<out Throwable>): Publisher<*>
{

        return attempts.flatMap { throwable ->
            if (++retryCount <= maxRetries) {

                Log.i("RetryWithDelay", "get error, it will try after " +
retryDelayMillis
                        + " millisecond, retry count " + retryCount)
                // When this Observable calls onNext, the original
                // Observable will be retried (i.e. re-subscribed).
                Flowable.timer(retryDelayMillis.toLong(),
TimeUnit.MILLISECONDS)

            } else {

                // Max retries hit. Just pass the error along.
                Flowable.error<Any>(throwable)
            }
        }
    }
}
```

12.4 小结

12.1 节讲述了 RxAndroid，在 Android 开发中如果想要使用 RxJava，那么 RxAndroid 是必不可少的。它是一个调度程序，能够切换到 Android 主线程或者任意指定的 Looper。

本章后面的小节讲述了 Retrofit，它是 Android 开发中流行的网络框架。在 12.3 节通过一个例子能够看到 Retrofit 将后端的 API 封装为 Java 接口，Retrofit 在内部使用动态代理将该接口的注解转换成 Http 请求，最后再执行 Http 请求。Retrofit 能够帮助开发者屏蔽网络框架的细节，因而开发者只需关注具体的业务。

第 13 章
开发EventBus

13.1 传统的 EventBus

EventBus 是事件总线框架，它是一种消息发布—订阅的模式，它的工作机制类似于观察者模式，通过通知者去注册观察者，最后由通知者向观察者发布消息。

在 Android 开发中，使用 EventBus 能够解耦 AsyncTask、Handler、Thread、Broadcast 等各种组件。除此之外，我们在开发 Android App 时还能使用 EventBus 来轻松实现跨越多个 Fragment 之间的通信。

EventBus 早期出自于 Google Guava，Guava 工程包含了若干被 Google 的 Java 项目广泛依赖的核心库。EventBus 是 Google Guava 中众多功能之一。

后来，Greenrobot 和 Square 也开发了类似的 EventBus 框架，这些框架只适合在 Android 系统中使用，而且在 Android 开发中变得非常流行。如果能够合理使用这些 EventBus 框架，就可以优雅地实现模块之间的解耦。如果使用不合理，则在开发中很可能会定义非常多的 Event，这也是被人所诟病的地方。

Greenrobot 的 Event Bus 官网：http://greenrobot.org/eventbus/。

GitHub：https://github.com/greenrobot/EventBus。

Event Bus 如图 13-1 所示。

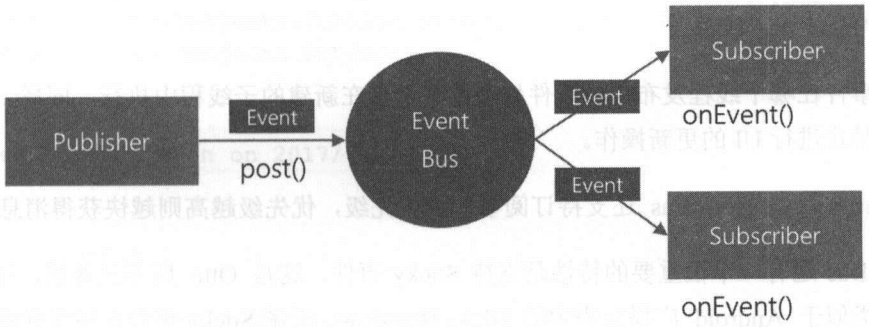


图 13-1

EventBus 的主要角色如下。

◎ Event：传递的事件对象。
◎ Subscriber：事件的订阅者。
◎ Publisher：事件的发布者。
◎ ThreadMode：定义函数在何种线程中执行。

其中，ThreadMode 总共有四种线程模型。

（1）Main UI 主线程。

无论事件是在哪个线程中发布出来的，该事件处理函数都会在 UI 线程中执行。该方法可以用来更新 UI，但是不能处理耗时操作。

（2）Background 后台线程。

如果事件是在 UI 线程中发布出来的，那么该事件处理函数就会在新的线程中运行。如果事件本来就是子线程中发布出来的，那么该事件处理函数将直接在当前线程中执行。在此事件处理函数中禁止进行 UI 更新操作。

（3）Posting 和发布者处在同一个线程。

无论该事件是在哪个线程发布出来的，事件处理函数会当前线程中运行。在线程模型为 PostThread 的事件处理函数中应尽量避免执行耗时操作，因为它会阻塞事件的传递，甚至有可能会引起 ANR（Application Not Responding）。

（4）Async 异步线程。

无论事件在哪个线程发布，该事件处理函数都会在新建的子线程中执行。同样，此事件处理函数中禁止进行 UI 的更新操作。

Greenrobot 的 EventBus 还支持订阅事件的优先级，优先级越高则越快获得消息。

EventBus 还有一个很重要的特性是支持 Sticky 事件，这是 Otto 所不具备的，这个 Sticky 事件有点类似于 Android 广播分类中的 Sticky Broadcast。所谓 Sticky 事件是指事件消费者在事件发布之后才注册的也能接收到该事件的特殊类型。在本章的后面，我们也会开发一个支持 Sticky 事件的 RxBus。

Square 的 Otto 官网：http://square.github.io/otto/。

GitHub：https://github.com/square/otto。

Otto 相比于 Greenrobot 的 EventBus 体积小了很多，它是通过反射牺牲了微小的性能，同时极大地降低了程序的耦合度。而且 Otto 的 API 相对于 EventBus 也更加友好。

Otto 的订阅者运行的线程与发送的线程是一致的，但是 EventBus 会根据用户声明订阅的函数时指定的线程来去运行，这也得益于 EventBus 拥有优秀的线程模型。所以相对而言，Greenrobot 的 EventBus 在功能上更加丰富。

目前，Otto 项目的 GitHub 上，官方已经宣布弃用 Square，推荐使用 RxJava 和 RxAndroid 来替代 Otto。这也是本章接下来的内容，在后面几个小节中，我们会使用 RxJava 来编写四种类型的 RxBus，每一种类型相对于前一种类型都增加了一些功能。

13.2 开发一个新的 EventBus（一）

EventBus 是一个很好的解耦工具，也是一个优秀的事件总线框架，而 RxJava 给我们带来了响应式的编程方式。如果在项目中已经使用了 RxJava，那么可以无须使用 EventBus，因为使用 RxJava 基本能够编写出实现 EventBus 的全部功能，而且代码量极少。

先来看看 RxJava 是如何编写一个最基本可用的 EventBus 版本的吧。

```
import io.reactivex.Observable;
```

```
import io.reactivex.subjects.PublishSubject;
import io.reactivex.subjects.Subject;

/**
 * Created by Tony Shen on 2017/6/21.
 */
public class RxBus {

    private final Subject<Object> mBus;

    private RxBus() {
        mBus = PublishSubject.create().toSerialized();
    }

    public static RxBus get() {
        return Holder.BUS;
    }

    public void post(Object obj) {
        mBus.onNext(obj);
    }

    public <T> Observable<T> toObservable(Class<T> tClass) {
        return mBus.ofType(tClass);
    }

    public Observable<Object> toObservable() {
        return mBus;
    }

    public boolean hasObservers() {
        return mBus.hasObservers();
    }

    private static class Holder {
        private static final RxBus BUS = new RxBus();
    }
}
```

在这个最基本的 EventBus 中，我们使用 Subject 来发送事件。Subject 既是 Observable，又

是 Observer(Subscriber)。在本书的 2.5 节中曾讲述过 Subject，如果有些遗忘可以翻回去看看。

在这里，EventBus 的私有构造函数中使用

```
mBus = PublishSubject.create().toSerialized();
```

这样的好处在于多线程环境下，可以保证线程安全，如果单独使用 Subject，线程是不安全的。

这个 RxBus 看上去是如此简单，真的能够实现 EventBus 的基本功能吗？我们先来看看它是如何使用的。

首先需要注册 Event 事件。下面的函数注册了两个 Event 事件，分别是 Fragment1Event 和 Fragment2Event。这两个事件又分别被两个不同的 fragment 所接收。此外，在注册事件的时候还用到了 CompositeDisposable，本书的第 9 章曾经讲解过 CompositeDisposable，这个类存在于 RxJava 2.x 中，在退出 Activity 或者 Fragment 时需要调用 compositeDisposable.clear()方法以便取消所有的门阐，防止内存泄露。

```
private void registerEvents() {

        compositeDisposable.add(rxBus.toObservable(Fragment1Event.class)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(@NonNull Object o) throws Exception {

                        fragment2.getText2().setText("fragment2 已经接收到事件
");
                    }
                }));

        compositeDisposable.add(rxBus.toObservable(Fragment2Event.class)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(@NonNull Object o) throws Exception {
```

```
                    fragment1.getText1().setText("fragment1 已经接收到事件
");
                }
            }));
    }
```

这个 demo 的具体 UI 界面如图 13-2 所示，它的布局是由上下两个 Fragment 组成的。

图 13-2

它完整的 Activity 是这样的，注意在 onDestroy() 方法中记得要调用 compositeDisposable.clear()。另外，RxBus 已经定义在 BaseActivity 中了。

```
import android.os.Bundle;
import android.support.v4.app.FragmentTransaction;

import com.saframework.study.rxbus1.R;
import com.saframework.study.rxbus1.app.BaseActivity;
import com.saframework.study.rxbus1.domain.Fragment1Event;
import com.saframework.study.rxbus1.domain.Fragment2Event;
import com.saframework.study.rxbus1.fragment.Fragment1;
import com.saframework.study.rxbus1.fragment.Fragment2;
```

```
import io.reactivex.android.schedulers.AndroidSchedulers;
import io.reactivex.annotations.NonNull;
import io.reactivex.disposables.CompositeDisposable;
import io.reactivex.functions.Consumer;

/**
 * Created by Tony Shen on 2017/6/22.
 */

public class TestEventBusActivity extends BaseActivity {

    private Fragment1 fragment1;
    private Fragment2 fragment2;
    private CompositeDisposable compositeDisposable = new
CompositeDisposable();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_test_event_bus);

        initViews();
        registerEvents();
    }

    private void initViews() {

        fragment1 = new Fragment1();
        fragment2 = new Fragment2();

        FragmentTransaction mTransactiont =
getSupportFragmentManager().beginTransaction();
        mTransactiont.replace(R.id.layout1, fragment1,
fragment1.getClass().getName());
        mTransactiont.replace(R.id.layout2, fragment2,
fragment2.getClass().getName());
        mTransactiont.commit();
    }

    private void registerEvents() {

        compositeDisposable.add(rxBus.toObservable(Fragment1Event.class)
```

```
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(@NonNull Object o) throws Exception {

                        fragment2.getText2().setText("fragment2 已经接收到事件
");
                    }
                }));

        compositeDisposable.add(rxBus.toObservable(Fragment2Event.class)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(@NonNull Object o) throws Exception {

                        fragment1.getText1().setText("fragment1 已经接收到事件
");
                    }
                }));
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        compositeDisposable.clear();
    }
}
```

当我们点击“发送事件到 FRAGMENT2”按钮时，会在 fragment2 的界面上显示“fragment2 已经接收到事件”，如图 13-3 所示。

如果点击“发送事件到 FRAGMENT1”按钮，则相应地在 fragment1 的界面上就会显示“fragment1 已经接收到事件”，如图 13-4 所示。

图 13-3

图 13-4

介绍完 TestEventBusActivity 如何注册 Event 事件之后，我们再来看看在具体的 fragment1 中是如何发送 Event 事件的。

```
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Button;
import android.widget.TextView;

import com.jakewharton.rxbinding2.view.RxView;
import com.saframework.injectview.Injector;
import com.saframework.injectview.annotations.InjectView;
import com.saframework.study.rxbus1.R;
import com.saframework.study.rxbus1.app.BaseFragment;
import com.saframework.study.rxbus1.domain.Fragment1Event;

import java.util.concurrent.TimeUnit;

import io.reactivex.annotations.NonNull;
import io.reactivex.functions.Consumer;
```

```
/**
 * Created by Tony Shen on 2017/6/22.
 */

public class Fragment1 extends BaseFragment {


    @InjectView(R.id.text1)
    TextView text1;

    @InjectView(R.id.button1)
    Button button1;

    @Override
    public View onCreateView(LayoutInflater inflater, ViewGroup container,
Bundle savedInstanceState) {
        View v = inflater.inflate(R.layout.fragment_1, container, false);

        Injector.injectInto(this, v);

        initViews();

        return v;
    }

    private void initViews() {

        RxView.clicks(button1)
                .throttleFirst(500, TimeUnit.MILLISECONDS)
                .subscribe(new Consumer<Object>() {

                    @Override
                    public void accept(@NonNull Object o) throws Exception {
                        rxBus.post(new Fragment1Event());
                    }
                });
    }

    public TextView getText1() {
        return text1;
    }
```

```
}
```

在 initViews()方法中，如果点击了 button1，则 RxBus 就会发送 Fragment1Event 事件。使用起来与 Otto 差不多。

```
rxBus.post(new Fragment1Event());
```

在 UI 界面上，button1 对应的文字内容是“发送事件到 FRAGMENT2”。点击这个按钮后，Fragment2 会接收消息并将内容展示在 text2 控件上。

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
              android:orientation="vertical"
              android:layout_width="match_parent"
              android:layout_height="match_parent">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="this is fragment1"
        android:layout_margin="5dp"/>

    <TextView
        android:id="@+id/text1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginTop="10dp"
        android:layout_marginLeft="5dp"/>

    <Button
        android:id="@+id/button1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="发送事件到 Fragment2"/>
</LinearLayout>
```

在这个 Fragment 中，使用了 RxBinding 这个库来封装点击事件，由于它本身也是使用 RxJava 来编写的，RxView.clicks()返回的是 Observable，所以可以使用 throttleFirst 操作符来防止控件被多次点击。

最后，本节所有的完整代码都在 GitHub 上，读者可以下载下来运行本章的工程，以加深印象。

13.3　开发一个新的 EventBus（二）

第一个 EventBus 只是完成了最基本的功能，虽然也考虑了多线程的处理，但是没有考虑到背压的情况。因为在 RxJava 2 中，Subject 已经不再支持背压了。

所以，第二个 EventBus 在第一个 EventBus 的基础上增加了对背压的处理，将 Subject 替换成支持背压的 PublishProcessor。

```
import io.reactivex.Flowable;
import io.reactivex.processors.FlowableProcessor;
import io.reactivex.processors.PublishProcessor;

/**
 * Created by Tony Shen on 2017/6/21.
 */

public class RxBus {

    private final FlowableProcessor<Object> mBus;

    private RxBus() {
        mBus = PublishProcessor.create().toSerialized();
    }

    public static RxBus get() {
        return Holder.BUS;
    }

    public void post(Object obj) {
        mBus.onNext(obj);
    }

    public <T> Flowable<T> toFlowable(Class<T> tClass) {
        return mBus.ofType(tClass);
    }
```

```
    public Flowable<Object> toFlowable() {
        return mBus;
    }

    public boolean hasSubscribers() {
        return mBus.hasSubscribers();
    }

    private static class Holder {
        private static final RxBus BUS = new RxBus();
    }
}
```

13.2 节使用 EventBus 进行跨 Fragment 的通信，这一次来看看跨 Activity 是如何进行通信的。

首先还是在 MainActivity 中注册一个事件：CrossActivityEvent。这个事件是让其他 Activity 来发送的，只要 MainActivity 没有执行 finish()方法，它就能接收到来自其他 Activity 发送的 CrossActivityEvent。下面是注册 CrossActivityEvent 的代码。

```
   private void registerEvents() {

        disposable = rxBus.toFlowable(CrossActivityEvent.class)
                .subscribe(new Consumer<CrossActivityEvent>() {
            @Override
            public void accept(@NonNull CrossActivityEvent event) throws
Exception {
                Toast.makeText(MainActivity.this,"来自 MainActivity 的
Toast",Toast.LENGTH_SHORT).show();
            }
        });
    }
```

然后单独创建一个 Activity，只用于发送 CrossActivityEvent。

```
import android.os.Bundle;

import com.saframework.study.rxbus2.app.BaseActivity;
import com.saframework.study.rxbus2.domain.CrossActivityEvent;

/**
 * Created by Tony Shen on 2017/6/22.
```

```
 */

public class TestCrossActivity extends BaseActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        rxBus.post(new CrossActivityEvent());
    }
}
```

当进入 TestCrossActivity 时，能够看到“来自 MainActivity 的 Toast”，这说明 MainActivity 已经接收到 CrossActivityEvent 并做了处理，发出一个 toast，如图 13-5 所示。

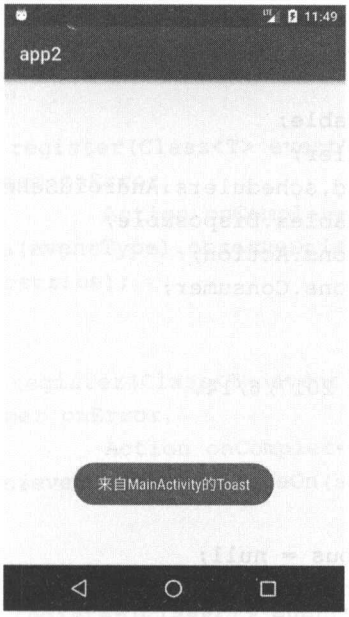


图 13-5

13.4　开发一个新的 EventBus（三）

RxJava 的操作符在链式调用中一旦有一个抛出了异常，Observer 就会直接执行 onError()方法，从而导致整个链式调用的结束。这是 RxJava 本身的设计原则。

对于前面两种类型的 EventBus，在订阅者处理事件时如果遇到了异常的情况，那么订阅者就会无法再收到事件。这对于 EventBus 而言是一个大麻烦。所以，我们又引入 Jake Wharton 大神写的 RxRelay，引入它之后即使出现异常也不会终止订阅关系，从而保证 App 还能正常使用。

RxRelay 的 Github 地址：https://github.com/JakeWharton/RxRelay。

RxRelay 中的各个 Relay 既是 Observable 的类型，也是 Consumer 的类型，它们是一个没有 onComplete 和 onError 的 Subject。所以不必担心下游触发的终止状态（onComplete 或 onError）。

再来看看第三个 EventBus 是如何实现的。

```
package com.saframework.study.rxbus3;

import com.jakewharton.rxrelay2.PublishRelay;
import com.jakewharton.rxrelay2.Relay;

import io.reactivex.Observable;
import io.reactivex.Scheduler;
import io.reactivex.android.schedulers.AndroidSchedulers;
import io.reactivex.disposables.Disposable;
import io.reactivex.functions.Action;
import io.reactivex.functions.Consumer;

/**
 * Created by Tony Shen on 2017/6/14.
 */

public class RxBus {

    private Relay<Object> bus = null;
    private static RxBus instance;

    //禁用构造方法
    private RxBus() {
        bus = PublishRelay.create().toSerialized();
    }

    public static RxBus get() {
        return Holder.BUS;
    }
```

```
    public void post(Object event) {
        bus.accept(event);
    }

    public  <T> Observable<T> toObservable(Class<T> eventType) {
        return bus.ofType(eventType);
    }

    public boolean hasObservers() {
        return bus.hasObservers();
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext) {
        return
toObservable(eventType).observeOn(scheduler).subscribe(onNext);
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext, Consumer onError,
                                   Action onComplete, Consumer onSubscribe) {
        return toObservable(eventType).observeOn(scheduler).subscribe(onNext,
onError, onComplete, onSubscribe);
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext, Consumer onError,
                                   Action onComplete) {
        return toObservable(eventType).observeOn(scheduler).subscribe(onNext,
onError, onComplete);
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext, Consumer onError) {
        return toObservable(eventType).observeOn(scheduler).subscribe(onNext,
onError);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext);
```

```
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext,
Consumer onError,
                                   Action onComplete, Consumer onSubscribe) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext, onError, onComplete, onSubscribe);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext,
Consumer onError,
                                   Action onComplete) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext, onError, onComplete);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext,
Consumer onError) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext, onError);
    }

    public void unregister(Disposable disposable) {
        if (disposable != null && !disposable.isDisposed()) {
            disposable.dispose();
        }
    }

    private static class Holder {
        private static final RxBus BUS = new RxBus();
    }

}
```

第三个 EventBus 相对于前面两种 EventBus 封装得更加友好，API 也更加类似于 Greenrobot 的 EventBus，笔者在个人的项目中也会经常使用它。

首先，来看看在 MainActivity 中是如何注册事件的。下面的代码在 MainActivity 中注册了两个事件，分别是 CrossActivityEvent 和 ExceptionEvent，前面已经介绍过 CrossActivityEvent 的作用，下面介绍一下 ExceptionEvent 的作用。

```
  private void registerEvents() {

      compositeDisposable.add(rxBus.register(CrossActivityEvent.class,
AndroidSchedulers.mainThread(), new Consumer<CrossActivityEvent>() {
          @Override
          public void accept(@NonNull CrossActivityEvent event) throws
Exception {
              Toast.makeText(MainActivity.this,"来自 MainActivity 的
Toast",Toast.LENGTH_SHORT).show();
          }
      }));

      compositeDisposable.add(rxBus.register(ExceptionEvent.class,
AndroidSchedulers.mainThread(), new Consumer<ExceptionEvent>() {
          @Override
          public void accept(@NonNull ExceptionEvent event) throws Exception
{
              String str = null;
              System.out.println(str.substring(0));
          }
      }, new Consumer<Throwable>() {
          @Override
          public void accept(@NonNull Throwable throwable) throws Exception
{

              L.i(throwable.getMessage());
          }
      }));
  }
```

在 MainActivity 中，有一个按钮可以跳转到 TestExceptionActivity，如图 13-6 所示。

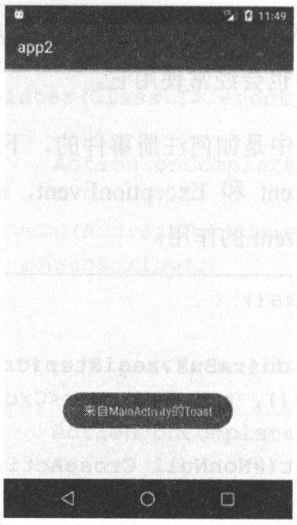


图 13-6

```
package com.saframework.study.rxbus3.activity;

import android.os.Bundle;

import com.saframework.study.rxbus3.app.BaseActivity;
import com.saframework.study.rxbus3.domain.ExceptionEvent;

/**
 * Created by Tony Shen on 2017/6/22.
 */

public class TestExceptionActivity extends BaseActivity {

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        rxBus.post(new ExceptionEvent());
    }
}
```

可以看到启动 TestExceptionActivity 时会发送一个 ExceptionEvent 事件，该事件之前已经在 MainActivity 中注册过了。

```
compositeDisposable.add(rxBus.register(ExceptionEvent.class,
AndroidSchedulers.mainThread(), new Consumer<ExceptionEvent>() {
            @Override
            public void accept(@NonNull ExceptionEvent event) throws Exception
{
                String str = null;
                System.out.println(str.substring(0));
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(@NonNull Throwable throwable) throws Exception
{

                L.i(throwable.getMessage());
            }
        }));
```

值得注意的是，通常情况下这样写 onNext()中的代码一定会报空指针异常，从而引起 App 崩溃。

```
String str = null;
System.out.println(str.substring(0));
```

但事实上，当 TestExceptionActivity 发出 ExceptionEvent 之后，App 并没有崩溃，而是打印了如图 13-7 所示的 log。这正是在 onError()中已经定义的，如果抛出 Throwable，则打印 throwable.getMessage()。

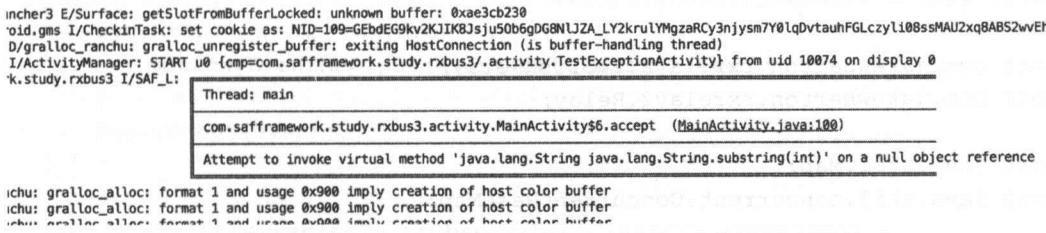


图 13-7

第三种类型的 RxBus，在使用时会更加让人放心。移动互联网发展到今天，业界的 Crash 标准一般是万分之三，如果某款 App 频繁 Crash，则用户一定会卸载它。能够让 App 稳定运行是第三种类型与前面二者的最大差别。

13.5 开发一个新的 EventBus（四）

通常，在使用 EventBus 时需要先注册好事件才能使用。是否存在这种情况，某一事件还没有来得及注册，但是已经发送了。等到事件注册成功之后，订阅者再进行消费。

```
rxBus.post(new SomeEvent());
```

这种情况也是存在的，Greenrobot 的 EventBus 称之为 Sticky 事件。Sticky 事件是指事件订阅者（消费者）在事件发布之后再进行注册，但是也能接收到该事件的特殊类型。

在使用上，Sticky 事件与普通的事件会有一些不同，需要使用 postSticky(Object event)。

```
rxBus.postSticky(new StickyEvent());
```

当然，如果明确知道该 StickyEvent 已经注册过了，那么也可以把 StickyEvent 当作普通的 Event 来使用。

```
rxBus.post(new StickyEvent());
```

支持 Sticky 事件的 EventBus 相对第 13.4 节讲述的第三个 EventBus 多了好几个方法。特别是 registerSticky(Class eventType, Scheduler scheduler, Consumer onNext) ，用于注册 Sticky Event。如果使用 register()来注册 Sticky Event，将无法使用 Sticky Event 的特性。

下面来看看第四个 EventBus 的具体代码。

```
package com.saframework.study.rxbus4;

import com.jakewharton.rxrelay2.PublishRelay;
import com.jakewharton.rxrelay2.Relay;

import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import io.reactivex.Observable;
import io.reactivex.ObservableEmitter;
import io.reactivex.ObservableOnSubscribe;
import io.reactivex.Scheduler;
import io.reactivex.android.schedulers.AndroidSchedulers;
import io.reactivex.annotations.NonNull;
import io.reactivex.disposables.Disposable;
```

```
import io.reactivex.functions.Action;
import io.reactivex.functions.Consumer;

/**
 * Created by Tony Shen on 2017/6/14.
 */

public class RxBus {

    private Relay<Object> bus = null;
    private static RxBus instance;
    private final Map<Class<?>, Object> mStickyEventMap;

    //禁用构造方法
    private RxBus() {
        bus = PublishRelay.create().toSerialized();
        mStickyEventMap = new ConcurrentHashMap<>();
    }

    public static RxBus get() {
        return Holder.BUS;
    }

    public void post(Object event) {
        bus.accept(event);
    }

    public void postSticky(Object event) {
        synchronized (mStickyEventMap) {
            mStickyEventMap.put(event.getClass(), event);
        }
        bus.accept(event);
    }

    public <T> Observable<T> toObservable(Class<T> eventType) {
        return bus.ofType(eventType);
    }

    /**
     * 根据传递的 eventType 类型返回特定类型(eventType)的被观察者
     */
```

```
    public <T> Observable<T> toObservableSticky(final Class<T> eventType) {
        synchronized (mStickyEventMap) {
            Observable<T> observable = bus.ofType(eventType);
            final Object event = mStickyEventMap.get(eventType);

            if (event != null) {
                return observable.mergeWith(Observable.create(new
ObservableOnSubscribe<T>() {
                    @Override
                    public void subscribe(@NonNull ObservableEmitter<T> e)
throws Exception {
                        e.onNext(eventType.cast(event));
                    }
                }));
            } else {
                return observable;
            }
        }
    }

    public boolean hasObservers() {
        return bus.hasObservers();
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext) {
        return
toObservable(eventType).observeOn(scheduler).subscribe(onNext);
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext, Consumer onError,
                                   Action onComplete, Consumer onSubscribe) {
        return toObservable(eventType).observeOn(scheduler).subscribe(onNext,
onError, onComplete, onSubscribe);
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext, Consumer onError,
                                   Action onComplete) {
        return toObservable(eventType).observeOn(scheduler).subscribe(onNext,
onError, onComplete);
```

```
    }

    public <T> Disposable register(Class<T> eventType, Scheduler scheduler,
Consumer<T> onNext, Consumer onError) {
        return toObservable(eventType).observeOn(scheduler).subscribe(onNext,
onError);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext,
Consumer onError,
                                   Action onComplete, Consumer onSubscribe) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext, onError, onComplete, onSubscribe);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext,
Consumer onError,
                                   Action onComplete) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext, onError, onComplete);
    }

    public <T> Disposable register(Class<T> eventType, Consumer<T> onNext,
Consumer onError) {
        return
toObservable(eventType).observeOn(AndroidSchedulers.mainThread()).subscribe
(onNext, onError);
    }

    public <T> Disposable registerSticky(Class<T> eventType, Scheduler
scheduler, Consumer<T> onNext) {
        return
toObservableSticky(eventType).observeOn(scheduler).subscribe(onNext);
    }
```

```
    public <T> Disposable registerSticky(Class<T> eventType, Consumer<T> onNext)
{
        return
toObservableSticky(eventType).observeOn(AndroidSchedulers.mainThread()).sub
scribe(onNext);
    }

    public <T> Disposable registerSticky(Class<T> eventType, Consumer<T> onNext,
Consumer onError) {
        return
toObservableSticky(eventType).observeOn(AndroidSchedulers.mainThread()).sub
scribe(onNext,onError);
    }

    /**
     * 移除指定 eventType 的 Sticky 事件
     */
    public <T> T removeStickyEvent(Class<T> eventType) {
        synchronized (mStickyEventMap) {
            return eventType.cast(mStickyEventMap.remove(eventType));
        }
    }

    /**
     * 移除所有的 Sticky 事件
     */
    public void removeAllStickyEvents() {
        synchronized (mStickyEventMap) {
            mStickyEventMap.clear();
        }
    }

    public void unregister(Disposable disposable) {
        if (disposable != null && !disposable.isDisposed()) {
            disposable.dispose();
        }
    }

    private static class Holder {
        private static final RxBus BUS = new RxBus();
    }
```

```
}
```

在这里，postSticky(Object event)和 post(Object event)的实现有很大的不同。registerSticky()方法也会调用 toObservableSticky(Class eventType) 方法来返回被观察者。这是第四个 EventBus 与第三个 EventBus 最大的区别。

先来看看 postSticky()的源码。

```
public void postSticky(Object event) {
    synchronized (mStickyEventMap) {
        mStickyEventMap.put(event.getClass(), event);
    }
    bus.accept(event);
}
```

mStickyEventMap 会把 Sticky 事件先缓存起来，mStickyEventMap 是一个 ConcurrentHashMap，所以它是线程安全的 HashMap，只用于缓存 Sticky 事件。

再来看看 toObservableSticky()的源码。

```
public <T> Observable<T> toObservableSticky(final Class<T> eventType) {
    synchronized (mStickyEventMap) {
        Observable<T> observable = bus.ofType(eventType);
        final Object event = mStickyEventMap.get(eventType);

        if (event != null) {
            return observable.mergeWith(Observable.create(new
ObservableOnSubscribe<T>() {
                @Override
                public void subscribe(@NonNull ObservableEmitter<T> e)
throws Exception {
                    e.onNext(eventType.cast(event));
                }
            }));
        } else {
            return observable;
        }
    }
}
```

先从 mStickyEventMap 中寻找是否包含该类型的事件，返回为空则说明没有 Sticky 事件要发送，返回 bus 订阅的 Subject。如果不为空，则说明有 Sticky 事件需要发送，Subject 和 Sticky 事件发送进行合并，从而保证 Sticky 事件在之后注册也能被订阅者消费。

最后再来看看实际的使用场景，下面的 TestStickyActivity 在 RxBus 注册事件之前就已经发送了两个事件，分别是 StickyEvent 和 NormalEvent。

```
package com.saframework.study.rxbus4.activity;

import android.os.Bundle;
import android.widget.Toast;

import com.saframework.study.rxbus4.app.BaseActivity;
import com.saframework.study.rxbus4.domain.NormalEvent;
import com.saframework.study.rxbus4.domain.StickyEvent;

import io.reactivex.android.schedulers.AndroidSchedulers;
import io.reactivex.annotations.NonNull;
import io.reactivex.disposables.CompositeDisposable;
import io.reactivex.functions.Consumer;

/**
 * Created by Tony Shen on 2017/6/22.
 */

public class TestStickyActivity extends BaseActivity {

    private CompositeDisposable compositeDisposable = new
CompositeDisposable();

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

        initData();
        registerEvents();
    }

    private void initData() {

        rxBus.postSticky(new StickyEvent());
```

```
        rxBus.post(new NormalEvent());
    }

    private void registerEvents() {

        compositeDisposable.add(rxBus.registerSticky(StickyEvent.class,
AndroidSchedulers.mainThread(), new Consumer<StickyEvent>() {

            @Override
            public void accept(@NonNull StickyEvent event) throws Exception {

                Toast.makeText(TestStickyActivity.this,"this is
StickyEvent",Toast.LENGTH_SHORT).show();
            }
        }));

        compositeDisposable.add(rxBus.register(NormalEvent.class,
AndroidSchedulers.mainThread(), new Consumer<NormalEvent>() {

            @Override
            public void accept(@NonNull NormalEvent event) throws Exception {

                Toast.makeText(TestStickyActivity.this,"this is
NormalEvent",Toast.LENGTH_SHORT).show();
            }
        }));
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        rxBus.removeStickyEvent(StickyEvent.class);
        compositeDisposable.clear();
    }
}
```

最终进入该页面时只显示了“this is StickyEvent”，因为 EventBuss 使用 registerSticky 注册了 StickyEvent，而 NormalEvent 由于最后没有注册，所以没有被执行，如图 13-8 所示。

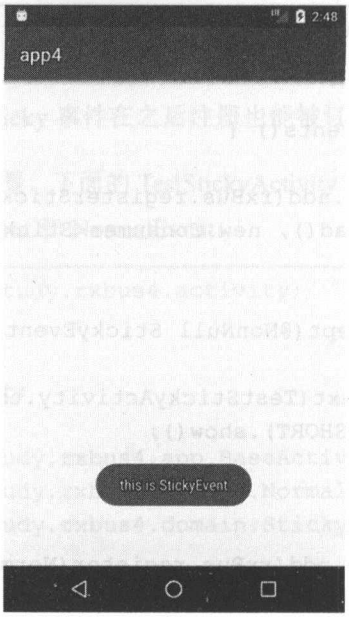


图 13-8

其实，Sticky 事件的用途还是很广泛的。在 App 开发中，App 初始化时可能要做很多事情，比如获取 App 的配置接口、做一些耗时的初始化工作等。为了让 App 能够快速进入主界面，可以在 App 的 Applicaton 类中使用事件总线来发送一个 Sticky 事件，在主界面中注册好 Sticky 事件。这样 App 就无须在启动页中做很多初始化的工作，从而能够快速进入主界面，而且也不会耽误初始化的过程。

13.6 小结

本章介绍了传统的 EventBus 的特性，以及如何使用 RxJava 2 来编写四种 EventBus。

之所以要写四种类型的 EventBus，是希望通过一步步地给 EventBus 添加功能，帮助读者理解 RxJava 的特性。在这里，我们重温了 RxJava 的 Back Pressure、Subject、PublishProcessor、异常处理等，还有 EventBus 的 Sticky 事件，以及在什么场景下才会用到 Sticky 事件，在 RxBus 中如何实现 Sticky 事件等。

有一点需要注意的是，只有第二种 EventBus 是支持背压的，其他三种 EventBus 均不支持。

第 14 章
使用RxJava封装HttpClient 4.5

14.1 HttpClient 的介绍

对于网络框架，除 JDK 自带的 URLConnection 外，在服务端开发中，Java 程序员用得比较多的是 Apache 的 HttpClient。笔者从 HttpClient 3.x 版本开始使用，到现在已经是 HttpClient 4.5 版本了。

在 Java 项目中，如果要使用 HttpClient 4.x，对于 Gradle 构建的工程，只需在对应 module 的 build.gradle 文件中添加如下配置，即可完成 HttpClient 依赖包的添加。

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'

    compile "org.apache.httpcomponents:httpclient:4.5.2"
    ...
}
```

Maven 构建的项目也是同理。

举一个简单的例子，通过 Get 请求来访问网易的首页。

```
        HttpEntity entity = null;

        try {
            String url = "http://www.163.com";

            // 使用默认配置创建 HttpClient 的实例
```

```
        CloseableHttpClient client = HttpClients.createDefault();

        HttpGet get = new HttpGet(url);

        CloseableHttpResponse response = client.execute(get);

        // 服务器返回码
        int statusCode = response.getStatusLine().getStatusCode();
        System.out.println("statusCode = " + statusCode);

        // 服务器响应成功
        if (statusCode==200) {
            // 服务器返回内容
            String respStr = null;
            entity = response.getEntity();
            if(entity != null) {
                respStr = EntityUtils.toString(entity, "UTF-8");
            }
            System.out.println(respStr);
        }

    } catch (Exception e) {
        e.printStackTrace();
    } finally {

        if (entity!=null) {
            // 释放资源
            try {
                EntityUtils.consume(entity);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
```

在这里，我们需要一个 HttpClient 的实例对象，一般情况下使用的是 org.apache.http.impl.client.CloseableHttpClient，创建它的直接方法是：

```
CloseableHttpClient client = HttpClients.createDefault();
```

HttpClients 提供了创建 CloseableHttpClient 实例的工厂方法。对于不考虑并发或者网络超时

的情况，只是作为演示，那么上面的程序足够让我们了解 HttpClient 的使用。然而在现实中，我们必须要考虑这些因素，所以下面的 HttpClient 中引入了连接池。

```
package com.saframework.study.httpclient;

import org.apache.http.*;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.config.RequestConfig;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;

import org.apache.http.util.EntityUtils;

import java.io.IOException;


/**
 * Created by tony on 2017/9/11.
 */
public class TestHttpClientWithPool {

    /** 全局连接池对象 */
    private static final PoolingHttpClientConnectionManager connManager = new
PoolingHttpClientConnectionManager();

    /**
     * 静态代码块配置连接池信息
     */
    static {
        // 设置最大连接数
        connManager.setMaxTotal(200);
        // 设置每个连接的路由数
        connManager.setDefaultMaxPerRoute(20);
    }

    /**
     * 获取 Http 客户端连接对象
     *
```

```
     * @param timeOut 超时时间
     * @return Http客户端连接对象
     */
    public static CloseableHttpClient getHttpClient(int timeOut) {
        // 创建Http请求配置参数
        RequestConfig requestConfig = RequestConfig.custom()
                // 获取连接超时时间
                .setConnectionRequestTimeout(timeOut)
                // 请求超时时间
                .setConnectTimeout(timeOut)
                // 响应超时时间
                .setSocketTimeout(timeOut)
                .build();

        // 创建HttpClient
        return HttpClients.custom()
                // 把请求相关的超时信息设置到连接客户端
                .setDefaultRequestConfig(requestConfig)
                // 把请求重试设置到连接客户端
                .setRetryHandler(new RetryHandler())
                // 配置连接池管理对象
                .setConnectionManager(connManager)
                .build();
    }

    /**
     * Get请求
     *
     * @param url 请求地址
     * @param timeOut 超时时间
     * @return
     */
    public static String httpGet(String url, int timeOut) {
        String msg = null;

        // 获取客户端连接对象
        CloseableHttpClient httpClient = getHttpClient(timeOut);
        // 创建Get请求对象
        HttpGet httpGet = new HttpGet(url);

        CloseableHttpResponse response = null;
```

```
        try {
            // 执行请求
            response = httpClient.execute(httpGet);
            // 获取响应实体
            HttpEntity entity = response.getEntity();
            // 获取响应信息
            msg = EntityUtils.toString(entity, "UTF-8");
        } catch (ClientProtocolException e) {
            System.err.println("协议错误");
            e.printStackTrace();
        } catch (ParseException e) {
            System.err.println("解析错误");
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("I/O 错误");
            e.printStackTrace();
        } finally {
            if (response!= null) {
                try {
                    EntityUtils.consume(response.getEntity());
                    response.close();
                } catch (IOException e) {
                    System.err.println("释放链接错误");
                    e.printStackTrace();
                }
            }
        }

        return msg;
    }

    public static void main(String[] args) {

        System.out.println(httpGet("http://www.163.com", 6000));
    }
}
```

其中，RetryHandler 实现了 HttpRequestRetryHandler 接口，它用于实现网络请求的重试机制。在这里， HttpClient 重试的次数 3，属于 Hard Code，更合理的做法是将重试次数放在配置文件中，程序使用时读取配置文件的信息。

```
package com.saframework.study.httpclient;

import org.apache.http.HttpEntityEnclosingRequest;
import org.apache.http.HttpRequest;
import org.apache.http.NoHttpResponseException;
import org.apache.http.client.HttpRequestRetryHandler;
import org.apache.http.client.protocol.HttpClientContext;
import org.apache.http.conn.ConnectTimeoutException;
import org.apache.http.protocol.HttpContext;

import javax.net.ssl.SSLException;
import javax.net.ssl.SSLHandshakeException;
import java.io.IOException;
import java.io.InterruptedIOException;
import java.net.UnknownHostException;

/**
 * Created by tony on 2017/9/11.
 */
public class RetryHandler implements HttpRequestRetryHandler {

    @Override
    public boolean retryRequest(IOException exception, int executionCount,
HttpContext context) {

        if (executionCount >= 3) {// 如果已经重试了 3 次，就放弃
            return false;
        }

        if (exception instanceof NoHttpResponseException) {// 如果服务器丢掉了连
// 接，那么就重试
            return true;
        }

        if (exception instanceof SSLHandshakeException) {// 不要重试 SSL 握手异常
            return false;
        }

        if (exception instanceof InterruptedIOException) {// 超时
            return true;
        }
```

```
        if (exception instanceof UnknownHostException) {// 目标服务器不可达
            return false;
        }

        if (exception instanceof ConnectTimeoutException) {// 连接被拒绝
            return false;
        }

        if (exception instanceof SSLException) {// ssl 握手异常
            return false;
        }

        HttpClientContext clientContext = HttpClientContext.adapt(context);
        HttpRequest request = clientContext.getRequest();

        // 如果请求是幂等的，就再次尝试
        if (!(request instanceof HttpEntityEnclosingRequest)) {
            return true;
        }
        return false;
    }
}
```

在实际后端开发中，HttpClient 需要结合自身业务的特点，来创建定制化的 HttpClient 实例，而且 RequestConfig 的参数也需要放在配置文件中。

TestHttpClientWithPool 类中只支持 Get 这一种 Http 方法，它是最基本的请求方式，其实还需要支持 Post 等更多的请求方式。对于 Restful 的架构，Put、Delete、Patch 等方法也是不可缺少的。随着越来越多的接口支持 https，在实际开发中我们使用的 HttpClient 还需支持 SSL。

14.2　使用 RxJava 进行重构

14.1 节我们介绍了 HttpClient 的作用以及简单的使用方式，本节主要讲解 RxJava 和 HttpClient 的结合。

首先，使用 RxJava 把第一个 HttpClient 的程序改造一下。用 create 操作符创建一个 Observable，然后在 map 操作符中完成 HttpGet 请求，把 CloseableHttpResponse 给观察者进行消费，如果网络请求成功，则打印服务器返回的 html 内容。

```
        Observable.create(new ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<String> e) throws
Exception {

                String url = "http://www.163.com";
                e.onNext(url);
            }
        }).map(new io.reactivex.functions.Function<String,
CloseableHttpResponse>() {

            @Override
            public CloseableHttpResponse apply(@NonNull String url) throws
Exception {

                CloseableHttpClient client = HttpClients.createDefault();

                HttpGet get = new HttpGet(url);

                return client.execute(get);
            }
        }).subscribe(new Consumer<CloseableHttpResponse>() {
            @Override
            public void accept(CloseableHttpResponse response) throws Exception
{

                // 服务器返回码
                int statusCode = response.getStatusLine().getStatusCode();
                System.out.println("statusCode = " + statusCode);

                HttpEntity entity = response.getEntity();

                // 服务器返回内容
                String respStr = null;

                if (entity != null) {
                    respStr = EntityUtils.toString(entity, "UTF-8");
                }

                System.out.println(respStr);

                // 释放资源
```

```
                EntityUtils.consume(entity);
            }
        });
```

虽说在这里使用 Observable 已经足够了，但是在 2.4 节里我们介绍过 Maybe 的特性，它可以看成是 Single 和 Completable 的结合。网络请求并不是一个连续事件流，一般只会发起一次网络请求返回数据并且只收到一个事件，因此，可以考虑将 onComplete() 与 onNext() 合并。所以，这里将 Observable 改成 Maybe。

```
        Maybe.create(new MaybeOnSubscribe<String>() {

            @Override
            public void subscribe(@NonNull MaybeEmitter<String> e) throws
Exception {
                String url = "http://www.163.com";
                e.onSuccess(url);
            }
        }).map(new Function<String, CloseableHttpResponse>() {

            @Override
            public CloseableHttpResponse apply(@NonNull String url) throws
Exception {

                CloseableHttpClient client = HttpClients.createDefault();

                HttpGet get = new HttpGet(url);

                return client.execute(get);
            }
        }).subscribe(new Consumer<CloseableHttpResponse>() {
            @Override
            public void accept(CloseableHttpResponse response) throws Exception
{

                // 服务器返回码
                int statusCode = response.getStatusLine().getStatusCode();
                System.out.println("statusCode = " + statusCode);

                HttpEntity entity = response.getEntity();

                // 服务器返回内容
```

```
                String respStr = null;

                if (entity != null) {
                    respStr = EntityUtils.toString(entity, "UTF-8");
                }

                System.out.println(respStr);

                // 释放资源
                EntityUtils.consume(entity);
            }
        });
```

接下来，再将使用连接池的例子修改一下，这里主要改动的是 httpGet()方法，它返回了 Maybe 对象。

```
package com.saframework.study.httpclient;

import io.reactivex.Maybe;
import io.reactivex.MaybeEmitter;
import io.reactivex.MaybeOnSubscribe;
import io.reactivex.annotations.NonNull;
import io.reactivex.disposables.Disposable;
import io.reactivex.functions.Consumer;
import io.reactivex.functions.Function;
import org.apache.http.HttpEntity;
import org.apache.http.ParseException;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.config.RequestConfig;
import org.apache.http.client.methods.CloseableHttpResponse;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.CloseableHttpClient;
import org.apache.http.impl.client.HttpClients;
import org.apache.http.impl.conn.PoolingHttpClientConnectionManager;
import org.apache.http.util.EntityUtils;

import java.io.IOException;

/**
 * Created by tony on 2017/9/11.
 */
```

```
public class TestHttpClientWithPoolAndMaybe {

    /**
     * 全局连接池对象
     */
    private static final PoolingHttpClientConnectionManager connManager = new
PoolingHttpClientConnectionManager();

    /**
     * 静态代码块配置连接池信息
     */
    static {
        // 设置最大连接数
        connManager.setMaxTotal(200);
        // 设置每个连接的路由数
        connManager.setDefaultMaxPerRoute(20);
    }

    /**
     * 获取 Http 客户端连接对象
     *
     * @param timeOut 超时时间
     * @return Http 客户端连接对象
     */
    public static CloseableHttpClient getHttpClient(int timeOut) {
        // 创建 Http 请求配置参数
        RequestConfig requestConfig = RequestConfig.custom()
                // 获取连接超时时间
                .setConnectionRequestTimeout(timeOut)
                // 请求超时时间
                .setConnectTimeout(timeOut)
                // 响应超时时间
                .setSocketTimeout(timeOut)
                .build();

        // 创建 httpClient
        return HttpClients.custom()
                // 把请求相关的超时信息设置到连接客户端
                .setDefaultRequestConfig(requestConfig)
                // 把请求重试设置到连接客户端
                .setRetryHandler(new RetryHandler())
```

```
                // 配置连接池管理对象
                .setConnectionManager(connManager)
                .build();
    }

    /**
     * GET 请求
     *
     * @param url     请求地址
     * @param timeOut 超时时间
     * @return
     */
    public static Maybe<String> httpGet(String url, int timeOut) {

        return Maybe.create(new MaybeOnSubscribe<String>() {

            @Override
            public void subscribe(@NonNull MaybeEmitter<String> e) throws
Exception {
                e.onSuccess(url);
            }
        }).map(new Function<String, String>() {
            @Override
            public String apply(@NonNull String s) throws Exception {

                // 获取客户端连接对象
                CloseableHttpClient httpClient = getHttpClient(timeOut);
                // 创建 GET 请求对象
                HttpGet httpGet = new HttpGet(url);

                CloseableHttpResponse response = httpClient.execute(httpGet);
                String msg = null;
                try {
                    // 执行请求
                    response = httpClient.execute(httpGet);
                    // 获取响应实体
                    HttpEntity entity = response.getEntity();
                    // 获取响应信息
                    msg = EntityUtils.toString(entity, "UTF-8");
                } catch (ClientProtocolException e) {
                    System.err.println("协议错误");
```

```
                e.printStackTrace();
            } catch (ParseException e) {
                System.err.println("解析错误");
                e.printStackTrace();
            } catch (IOException e) {
                System.err.println("IO 错误");
                e.printStackTrace();
            } finally {
                if (response != null) {
                    try {
                        EntityUtils.consume(response.getEntity());
                        response.close();
                    } catch (IOException e) {
                        System.err.println("释放链接错误");
                        e.printStackTrace();
                    }
                }
            }

            return msg;
        }
    });
}

public static void main(String[] args) {

    httpGet("http://www.163.com", 6000)
            .subscribe(new Consumer<String>() {
                @Override
                public void accept(String s) throws Exception {
                    System.out.println(s);
                }
            });
}
}
```

因为 httpGet()做了改动，所以完成一次网络请求需要按照如下的方式来写。如果没有调用 subscribe()方法，则网络请求不会执行。

```
    httpGet("http://www.163.com", 6000)
            .subscribe(new Consumer<String>() {
```

```
            @Override
            public void accept(String s) throws Exception {
                System.out.println(s);
            }
        });
```

至此，就大致完成了对之前 HttpClient 例子的改造。在实际开发中，网络框架不仅是服务端开发的基础框架，也是客户端开发的基础框架。一般情况下，开发者需要结合自身的业务需求来封装 HttpClient。比如说，如果要支持 Restful 的架构，则必须要有 Get、Post、Put、Delete 等方法。如果有图片上传的需求，则必须对图片上传做封装。若是有高并发需求，则需要对连接数、超时时间做一些调优。有时候，我们还会在 Header 中传递一些参数。所以，HttpClient 的封装与业务需求是紧密关联的。

在实际开发中，除 Apache HttpClient 外，还存在着很多网络框架，表 14-1 中列举了一些常见的网络框架。

表 14-1

常见网络框架	官网地址	特　性
JDK's URLConnection		使用传统的线程阻塞 I/O
Apache HTTP Client	https://hc.apache.org/httpcomponents-client-ga/index.html	使用基于线程池的线程阻塞 I/O
Apache Async HTTP Client	https://hc.apache.org/httpcomponents-asyncclient-4.1.x/index.html	使用 NIO
Jersey	https://jersey.github.io/	它是 Rest 客户端/服务器框架。客户端 API 可以使用多个 HTTP 框架，包括 URLConnection 和 Apache HTTP Client
OkHttp	https://square.github.io/okhttp	OkHttp 提供了对最新的 HTTP 协议版本 HTTP/2 和 SPDY 的支持，这使得对同一个主机发出的所有请求都可以共享相同的套接字连接。如果 HTTP/2 和 SPDY 不可用，则 OkHttp 会使用连接池来复用连接以提高效率
Retrofit	https://square.github.io/retrofit	将 HTTP API 转换为 Java 接口后，即使用多个 HTTP 框架，包括 Apache HTTP Client
Grizzly	https://grizzly.java.net	能够帮助开发人员利用 Java NIO API 构建可扩展、高性能、健壮的服务器

续表

常见网络框架	官网地址	特　性
Netty	http://netty.io	Netty 是一个高性能、异步事件驱动的 NIO 框架，它提供了对 TCP、UDP 和文件传输的支持。作为一个异步 NIO 框架，Netty 的所有 I/O 操作都是异步非阻塞的，通过 Future-Listener 机制，用户可以方便地主动获取或者通过通知机制获得 I/O 操作结果
Jetty Async HTTP Client	https://www.eclipse.org/jetty/documentation/current/http-client.html	使用 NIO
Async HTTP Client	https://github.com/AsyncHttpClient/async-http-client	包装了 Netty、Grizzly 或 JDK 的 HTTP 支持
clj-http	https://github.com/dakrone/clj-http	包装了 Apache HTTP 客户端
http-kit	http://www.http-kit.org/client.html	高性能、高并发 Clojure HTTP Server & Client
http async client	https://github.com/cch1/http.async.client	包装了 Async HTTP Client

在魔窗的服务端，我们曾使用 Async HTTP Client 做各种同步和异步的调用。另外，Async HTTP Client 也是支持 RxJava 的。

14.3　实现一个简单的图片爬虫

在实际开发中，我们经常需要对网络上的图片或者数据进行采集。此时，就需要使用网络爬虫来完成这些工作。

市面上有很多非常成熟的爬虫框架，笔者自己也写过一个简单的图片爬虫程序，可以用于抓取单张图片、多张图片、某个网页下的所有图片以及多个网页下的所有图片。

GitHub 地址 https://github.com/fengzhizi715/PicCrawler

这个爬虫使用了 HttpClient、RxJava 2 以及 Java 8 的一些特性。它支持一些简单的定制，比如定制 User-Agent、Referer、Cookies 等，它的结构如图 14-1 所示。

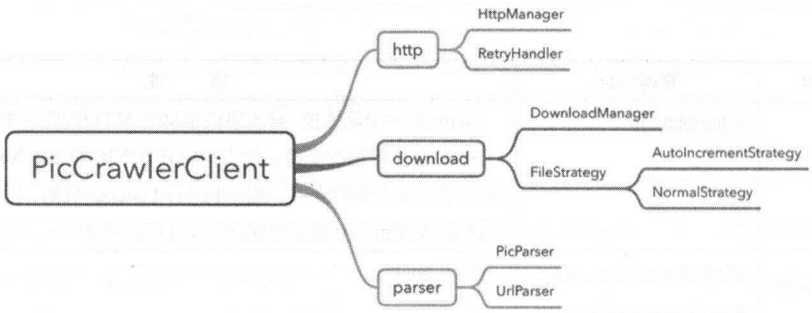


图 14-1

下面来看一个使用它的例子，爬取某个特定 URL 的全部图片。

```
String url = "http://www.designerspics.com/"; // 针对某一个网址
 picCrawlerClient.get()
        .timeOut(6000)
        .fileStrategy(new FileStrategy() {

            @Override
            public String filePath() {
                return "temp";
            }

            @Override
            public String picFormat() {
                return "png";
            }

            @Override
            public FileGenType genType() {

                return FileGenType.AUTO_INCREMENT;
            }
        })
        .build()
        .downloadWebPageImages(url);
```

在这里，timeOut()表示网络请求的超时时间。fileStrategy()表示存放的目录、文件使用的格式、生成文件时使用何种策略。PicCrawler 支持多种文件的生成策略，比如随机生成文件名、从 1 开始自增长地生成文件名、生成指定的文件名等。

最后的 downloadWebPageImages()方法表示下载某个 URL 下的全部图片，它的源码如下：

```
    /**
     * 下载整个网页的全部图片
     * @param url
     */
    public void downloadWebPageImages(String url) {

        if (Preconditions.isNotBlank(url)) {

            isWebPage = true;

            pageParser = new PicParser();

            Flowable.just(url)
                    .map(s->httpManager.createHttpWithGet(s))
                    .map(response->parseHtmlToImages(response,(PicParser)page
Parser))
                    .subscribe(urls -> downloadPics(urls),
                            throwable->
System.out.println(throwable.getMessage()));
        }
    }
```

从源码中可以看到 downloadWebPageImages()总共分成三步：创建网络请求、解析出当前页面中包含的图片路径、下载这些图片。

第一步，创建网络请求使用了 HttpClient，它支持 header 传递一些参数，同时还有一个特性：如果图片有防盗链接，则可以使用 referer()方法，传入网站的网址，这样就可以愉快地下载图片了。对于懒人还有一个方法：autoReferer()，可以不必传入网站的网址。这里可以看到，如果设置了 autoReferer()，则会从 URL 中提取网站的网址。

```
    /**
     * 创建网络请求—Get 请求
     * @param url
     * @return
     */
    public CloseableHttpResponse createHttpWithGet(String url) {

        // 获取客户端连接对象
```

```
        CloseableHttpClient httpClient = getHttpClient();
        // 创建 Get 请求对象
        HttpGet httpGet = new HttpGet(url);

        if (Preconditions.isNotBlank(httpParam)) {

            boolean autoReferer = httpParam.isAutoReferer();

            Map<String,String> header = httpParam.getHeader();

            if (Preconditions.isNotBlank(header)) {

                if (autoReferer && !header.containsKey("Referer")) {

                    header.put("Referer", Utils.getReferer(url));
                }

                for (String key : header.keySet()) {
                    httpGet.setHeader(key,header.get(key));
                }
            }
        }

        CloseableHttpResponse response = null;

        // 执行请求
        try {
            response = httpClient.execute(httpGet);
        } catch (IOException e) {
            e.printStackTrace();
        }

        return response;
    }
```

第二步，将返回的 response 转换成 String 类型，使用 jsoup 将带有图片的链接全部过滤出来。

jsoup 是一款 Java 的 HTML 解析器，可直接解析某个 URL 地址、HTML 文本内容。它提供了一套非常省力的 API，可通过 DOM、CSS 以及类似于 jQuery 的操作方法来取出和操作数据。

```
    /**
     * 对 response 进行解析，解析出图片的 URL，并存放到 List 中
     * @param response
     * @param picParser
     * @return
     */
    private List<String> parseHtmlToImages(CloseableHttpResponse
response,PicParser picParser) {

        // 获取响应实体
        HttpEntity entity = response.getEntity();

        InputStream is = null;
        String html = null;

        try {
            is = entity.getContent();
            html = IOUtils.inputStream2String(is);
        } catch (IOException e) {
            e.printStackTrace();
        }

        Document doc = Jsoup.parse(html);

        List<String> urls = picParser.parse(doc);

        if (response != null) {
            try {
                EntityUtils.consume(response.getEntity());
                response.close();
            } catch (IOException e) {
                System.err.println("释放链接错误");
                e.printStackTrace();
            }
        }

        return urls;
    }
```

在这里，使用单独的 PicParser 来解析图片。

```
import com.cv4j.piccrawler.utils.Utils;
```

```
import com.saframework.tony.common.utils.Preconditions;
import lombok.extern.slf4j.Slf4j;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import java.util.ArrayList;
import java.util.List;

/**
 * Created by tony on 2017/11/14.
 */
@Slf4j
public class PicParser implements PageParser<List<String>>{

    @Override
    public List<String> parse(Document doc) {

        List<String> urls = new ArrayList<>();

        Elements media = doc.select("[src]");

        if (Preconditions.isNotBlank(media)) {

            for (Element src : media) {
                if (src.tagName().equals("img")) {

                    if (Preconditions.isNotBlank(src.attr("abs:src"))) { // 图
// 片的绝对路径不为空

                        String picUrl = src.attr("abs:src");
                        log.info(picUrl);
                        urls.add(picUrl);
                    } else if (Preconditions.isNotBlank(src.attr("src"))){ // 图
// 片的相对路径不为空

                        String picUrl = src.attr("src").replace("//","");
                        picUrl = "http://"+ Utils.tryToEscapeUrl(picUrl);
                        log.info(picUrl);
                        urls.add(picUrl);
                    }
                }
```

```
            }
        }

        return urls;
    }
}
```

第三步，下载这些图片，使用 Java 8 的 CompletableFuture 功能以及 Stream API 的并行功能。CompletableFuture 是 Java 8 新增的用于异步处理的类，并且 CompletableFuture 的性能要好于传统的 Future。在第 16 章中，会详细介绍 CompletableFuture 的用法。

```
/**
 * 下载多张图片
 * @param urls
 */
public void downloadPics(List<String> urls) {

    if (Preconditions.isNotBlank(urls)) {
        urls.stream().parallel().forEach(url->{

            try {
                CompletableFuture.runAsync(() -> downloadPic(url)).get();
            } catch (InterruptedException e) {
                e.printStackTrace();
            } catch (ExecutionException e) {
                e.printStackTrace();
            }
        });
    }
}
```

对 PicCrawler 感兴趣的读者，可以到 GitHub 上查看该项目的源码，该项目对 HttpClient 的封装比较完善。

14.4　小结

本章介绍了 HttpClient，以及如何使用 RxJava 来调用 HttpClient，还介绍了一个简单的图片爬虫程序。

在 Android 开发中，OkHttp 和 Retrofit 是目前比较流行的网络框架，本书第 12 章中已经讲述了 Retrofit 的用法。在服务端开发中，网络框架的选择范围会更广，开发者需要结合自身的业务场景来选择合适的网络框架，以及对网络框架的封装。

第 15 章
Spring Boot和RxJava 2

15.1 模拟 Task 任务

1. Spring

Spring 是一个开放源代码的设计层面框架，它解决的是业务逻辑层和其他各层的松耦合问题，因此它将面向接口的编程思想贯穿整个系统应用。Spring 是于 2003 年兴起的一个轻量级的 Java 开发框架，由 Rod Johnson 创建。简单来说，Spring 是一个分层的 JavaSE/EE full-stack（一站式）轻量级开源框架。

Rod Johnson 在 2002 年编著的 *Expert one on one J2EE design and development* 一书中，对 Java EE 系统框架臃肿、低效、脱离现实的种种现状提出了质疑，并积极寻求探索革新之道。以此书为指导思想，他编写了 interface21 框架，这是一个力图冲破 J2EE 传统开发困境，从实际需求出发，着眼于轻便、灵巧，易于开发、测试和部署的轻量级开发框架。Spring 框架即以 interface21 框架为基础，经过重新设计，并不断丰富其内涵，于 2004 年 3 月 24 日，发布了 1.0 正式版。同年他又推出了一部堪称经典的力作 *Expert one-on-one J2EE Development without EJB*，该书在 Java 界掀起了轩然大波，不断改变着 Java 开发者程序设计和开发的思考方式。在该书中，作者根据自己多年丰富的实践经验，对 EJB 的各种笨重臃肿的结构进行了逐一的分析和否定，并分别以简洁实用的方式替换之。至此一战功成，Rod Johnson 成为一个改变 Java 世界的大师级人物。

Spring 致力于 J2EE 应用的各层的解决方案，而不是仅仅专注于某一层的方案。可以说 Spring 是企业应用开发的“一站式”选择，并贯穿表现层、业务层及持久层。然而，Spring 并不想取

代那些已有的框架，而是与它们无缝整合。

Spring 的主要特点如下。

（1）方便解耦，简化开发。

通过 Spring 提供的 IoC 容器，我们可以将对象之间的依赖关系交由 Spring 进行控制，避免硬编码所造成的过度程序耦合。有了 Spring，用户不必再为单实例模式类、属性文件解析等这些很底层的需求编写代码，可以更专注于上层的应用。

（2）AOP 编程的支持。

通过 Spring 提供的 AOP 功能，可以方便地进行面向切面的编程，许多不容易用传统 OOP 实现的功能可以通过 AOP 轻松应付。

（3）声明式事务的支持。

在 Spring 中，我们可以从单调烦闷的事务管理代码中解脱出来，通过声明式方式灵活地进行事务的管理，提高开发效率和质量。

（4）方便程序的测试。

可以用非容器依赖的编程方式进行几乎所有的测试工作，在 Spring 里，测试不再是昂贵的操作，而是随手可做的事情。例如，Spring 对 JUnit4 支持，可以通过注解方便地测试 Spring 程序。

（5）方便集成各种优秀框架。

Spring 不排斥各种优秀的开源框架，相反，Spring 可以降低各种框架的使用难度，Spring 提供了对各种优秀框架（如 Struts、Hibernate、Hessian、Quartz 等）的直接支持。

（6）降低 Java EE API 的使用难度。

Spring 对很多难用的 Java EE API（如 JDBC、JavaMail、远程调用等）提供了一个薄薄的封装层，通过 Spring 的简易封装，这些 Java EE API 的使用难度大为降低。

（7）Java 源码是经典学习范例。

Spring 的源码设计精妙、结构清晰、匠心独运，处处体现着大师对 Java 设计模式的灵活运

用以及对 Java 技术的高深造诣。Spring 框架源码无疑是 Java 技术的最佳实践范例。如果想在短时间内迅速提高自己的 Java 技术水平和应用开发水平，学习和研究 Spring 源码将会使你收到意想不到的效果。

2. Spring Boot

Spring Boot 是由 Pivotal 团队提供的全新框架，其设计目的是用来简化新 Spring 应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot 致力于在蓬勃发展的快速应用开发领域成为领导者。

多年以来，Spring I/O 平台饱受非议的一点就是大量的 XML 配置以及复杂的依赖管理。在 2013 年的 SpringOne 2GX 会议上，Pivotal 的 CTO Adrian Colyer 回应了这些批评，并且特别提到该平台将来的目标之一就是实现免 XML 配置的开发体验。Spring Boot 所实现的功能超出了这个任务的描述，开发人员不仅不再需要编写 XML，而且在一些场景中甚至不需要编写烦琐的 import 语句。

然而，Spring Boot 并不是要成为 Spring I/O 平台里面众多“Foundation”层项目的替代者。Spring Boot 的目标不在于为已解决的问题域提供新的解决方案，而是为平台带来另一种开发体验，从而简化对这些已有技术的使用。对于已经熟悉 Spring 生态系统的开发人员来说，Boot 是一个很理想的选择；对于采用 Spring 技术的新人来说，Boot 提供了一种更简洁的方式来使用这些技术。

Spring Boot 的主要特点如下：

- ◎ 创建独立的 Spring 应用程序。
- ◎ 直接嵌入 Tomcat、Jetty 或 Undertow（无须部署 WAR 文件）。
- ◎ 提供“初始”的 POM 文件内容，以简化 Maven 配置。
- ◎ 尽可能自动配置 Spring。
- ◎ 提供生产就绪的功能，如指标、健康检查和外部化配置。
- ◎ 绝对无代码生成，也不需要 XML 配置。

这几年来，Spring Boot 以及 Spring Cloud 越来越受到欢迎，逐渐成为开发“微服务”架构的首选框架。

（1）模拟并发地执行任务。

本节的工程，可以模拟现实中任务顺序执行和并发执行的效果。

（2）添加 Spring Boot 配置。

为了在项目中引入 Spring Boot，需要在 build.gradle 中添加如下配置。

```
repositories {
    jcenter()
    mavenCentral()
    maven { url 'http://repo.spring.io/release' }
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.12'

    compile "io.reactivex.rxjava2:rxjava:2.1.9"
    compile "org.apache.httpcomponents:httpclient:4.5.2"
    ......

    compile 'org.springframework.boot:spring-boot-starter-web:1.5.8.RELEASE'
}
```

（3）创建 MockTask。

然后创建一个模拟的任务，其中 delayInSeconds 用来模拟任务所花费的时间，单位是秒。

```
import com.saframework.study.task.core.common.ITask;
import lombok.Getter;
import lombok.RequiredArgsConstructor;

import java.util.concurrent.TimeUnit;

/**
 * Created by tony on 2017/11/19.
 */
@Getter
@RequiredArgsConstructor
public class MockTask implements ITask {

    private final int delayInSeconds;

    private boolean started;
```

```
    private boolean finishedSuccessfully;

    private boolean interrupted;

    private long threadId;

    @Override
    public void execute() {
        try {
            this.threadId = Thread.currentThread().getId();
            this.started = true;
            TimeUnit.SECONDS.sleep(delayInSeconds);
            this.finishedSuccessfully = true;
        } catch (InterruptedException e) {
            this.interrupted = true;
        }
    }

    public static MockTask notDelayedTask() {
        return new MockTask(0);
    }

    public static MockTask fiveSecondsDelayedTask() {
        return new MockTask(5);
    }
}
```

（4）创建 ConcurrentTasksExecutor。

顺序执行比较简单，一个任务接着一个任务地完成即可，是单线程的操作。对于并发而言，在这里借助 RxJava 的 merge 操作符来将多个任务进行合并。还用到了 RxJava 的任务调度器 Scheduler，createScheduler()是按照所需的线程数来创建 Scheduler 的。

merge 操作符可以将多个 Observable 的输出合并，使得它们就像是一个单个的 Observable 一样。

```
import com.saframework.study.task.core.common.ITask;
import io.reactivex.Completable;
import io.reactivex.Scheduler;
import io.reactivex.schedulers.Schedulers;
```

```
import lombok.extern.slf4j.Slf4j;
import org.springframework.util.CollectionUtils;

import java.util.Arrays;
import java.util.Collection;
import java.util.List;
import java.util.concurrent.Executors;
import java.util.stream.Collectors;

/**
 * Created by tony on 2017/11/19.
 */
@Slf4j
public class ConcurrentTasksExecutor implements ITask {

    private final int numberOfConcurrentThreads;
    private final Collection<ITask> tasks;

    public ConcurrentTasksExecutor(int numberOfConcurrentThreads,
Collection<ITask> tasks) {

        if (numberOfConcurrentThreads<0) {
            throw new RuntimeException("Amount of threads must be higher than
zero.");
        }

        this.numberOfConcurrentThreads = numberOfConcurrentThreads;
        this.tasks = tasks;
    }

    public ConcurrentTasksExecutor(int numberOfConcurrentThreads, ITask...
tasks) {
        this(numberOfConcurrentThreads, tasks == null ? null :
Arrays.asList(tasks));
    }

    @Override
    public void execute() {
        if (isTasksCollectionEmpty()) {
            log.warn("There are no tasks to be executed.");
            return;
        }
```

```
        log.debug("Executing #{} tasks concurrent way.", tasks.size());
        Completable.merge(getAsConcurrentTasks()).blockingAwait();
    }

    private Scheduler createScheduler() {
        return
Schedulers.from(Executors.newFixedThreadPool(numberOfConcurrentThreads));
    }

    private List<Completable> getAsConcurrentTasks() {
        final Scheduler scheduler = createScheduler();

        return tasks.stream()
                .filter(task -> task != null)
                .map(task -> Completable
                        .fromAction(task::execute)
                        .subscribeOn(scheduler))
                .collect(Collectors.toList());
    }

    private boolean isTasksCollectionEmpty() {
        return CollectionUtils.isEmpty(tasks);
    }
}
```

（5）创建 Controller。

表现层使用 Spring MVC，会分别把 sequential 和 concurrent 执行任务的时间展示出来。

```
import com.saframework.study.task.core.common.ITask;
import com.saframework.study.task.core.impl.ConcurrentTasksExecutor;
import com.saframework.study.task.core.impl.MockTask;
import com.saframework.study.task.web.dto.ApiResponseDTO;
import com.saframework.study.task.web.dto.ErrorResponseDTO;
import lombok.extern.slf4j.Slf4j;
import org.springframework.http.HttpStatus;
import org.springframework.util.StopWatch;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.IntStream;
```

```
/**
 * Created by tony on 2017/11/19.
 */
@Slf4j
@RestController
@RequestMapping("/tasks")
public class TasksController {

    @GetMapping("/sequential")
    public ApiResponseDTO sequential(@RequestParam("task") int[]
taskDelaysInSeconds) {
        StopWatch watch = new StopWatch();
        watch.start();

        IntStream.of(taskDelaysInSeconds).mapToObj(MockTask::new).forEach(Mo
ckTask::execute);

        watch.stop();
        return new ApiResponseDTO(watch.getTotalTimeSeconds());
    }

    @GetMapping("/concurrent")
    public ApiResponseDTO concurrent(@RequestParam("task") int[]
taskDelaysInSeconds, @RequestParam("threads") int numberOfConcurrentThreads)
{
        StopWatch watch = new StopWatch();
        watch.start();

        List<ITask> delayedTasks =
IntStream.of(taskDelaysInSeconds).mapToObj(MockTask::new).collect(Collector
s.toList());
        new ConcurrentTasksExecutor(numberOfConcurrentThreads,
delayedTasks).execute();

        watch.stop();
        return new ApiResponseDTO(watch.getTotalTimeSeconds());
    }

    @ExceptionHandler({IllegalArgumentException.class})
    @ResponseStatus(HttpStatus.BAD_REQUEST)
    public ErrorResponseDTO handleException(IllegalArgumentException e) {
```

```
        return new ErrorResponseDTO(e.getMessage());
    }
}
```

（6）工程入口。

执行下面的程序就可以把整个工程运行起来。

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

/**
 * Created by tony on 2017/11/19.
 */
@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

如图 15-1 所示。

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v1.5.8.RELEASE)

2017-11-26 06:16:21.259  INFO 9462 --- [           main] com.saffframework.study.task.Application  : Starting Application on TonydeMacBook-Pro.local with PID 9462
 (/Users/tony/RxJavaInAction/Tasks/out/production/classes started by tony in /Users/tony/RxJavaInAction/Tasks)
2017-11-26 06:16:21.263  INFO 9462 --- [           main] com.saffframework.study.task.Application  : No active profile set, falling back to default profiles: default
2017-11-26 06:16:21.304  INFO 9462 --- [           main] ationConfigEmbeddedWebApplicationContext : Refreshing org.springframework.boot.context.embedded
.AnnotationConfigEmbeddedWebApplicationContext@31206beb: startup date [Sun Nov 26 06:16:21 CST 2017]; root of context hierarchy
2017-11-26 06:16:22.098  INFO 9462 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat initialized with port(s): 8080 (http)
2017-11-26 06:16:22.108  INFO 9462 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2017-11-26 06:16:22.108  INFO 9462 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet Engine: Apache Tomcat/8.5.23
2017-11-26 06:16:22.190  INFO 9462 --- [ost-startStop-1] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2017-11-26 06:16:22.191  INFO 9462 --- [ost-startStop-1] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 890 ms
2017-11-26 06:16:22.287  INFO 9462 --- [ost-startStop-1] o.s.b.w.servlet.ServletRegistrationBean  : Mapping servlet: 'dispatcherServlet' to [/]
2017-11-26 06:16:22.290  INFO 9462 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'characterEncodingFilter' to: [/*]
2017-11-26 06:16:22.290  INFO 9462 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
2017-11-26 06:16:22.290  INFO 9462 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'httpPutFormContentFilter' to: [/*]
2017-11-26 06:16:22.290  INFO 9462 --- [ost-startStop-1] o.s.b.w.servlet.FilterRegistrationBean   : Mapping filter: 'requestContextFilter' to: [/*]
2017-11-26 06:16:22.512  INFO 9462 --- [           main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded
.AnnotationConfigEmbeddedWebApplicationContext@31206beb: startup date [Sun Nov 26 06:16:21 CST 2017]; root of context hierarchy
2017-11-26 06:16:22.561  INFO 9462 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/tasks/concurrent],methods=[GET]}" onto public com.saffframework.study.task.web
.dto.ApiResponseDTO com.saffframework.study.task.web.controller.TasksController.concurrent(int[],int)
2017-11-26 06:16:22.562  INFO 9462 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/tasks/sequential],methods=[GET]}" onto public com.saffframework.study.task.web
.dto.ApiResponseDTO com.saffframework.study.task.web.controller.TasksController.sequential(int[])
2017-11-26 06:16:22.564  INFO 9462 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error]}" onto public org.springframework.http.ResponseEntity<java.util
.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)
2017-11-26 06:16:22.564  INFO 9462 --- [           main] s.w.s.m.m.a.RequestMappingHandlerMapping : Mapped "{[/error],produces=[text/html]}" onto public org.springframework.web.servlet
.ModelAndView org.springframework.boot.autoconfigure.web.BasicErrorController.errorHtml(javax.servlet.http.HttpServletRequest,javax.servlet.http.HttpServletResponse)
2017-11-26 06:16:22.582  INFO 9462 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/webjars/**] onto handler of type [class org.springframework.web
.servlet.resource.ResourceHttpRequestHandler]
2017-11-26 06:16:22.583  INFO 9462 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**] onto handler of type [class org.springframework.web.servlet
.resource.ResourceHttpRequestHandler]
2017-11-26 06:16:22.606  INFO 9462 --- [           main] o.s.w.s.handler.SimpleUrlHandlerMapping  : Mapped URL path [/**/favicon.ico] onto handler of type [class org.springframework.web
.servlet.resource.ResourceHttpRequestHandler]
2017-11-26 06:16:22.690  INFO 9462 --- [           main] o.s.j.e.a.AnnotationMBeanExporter        : Registering beans for JMX exposure on startup
2017-11-26 06:16:22.742  INFO 9462 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http)
2017-11-26 06:16:22.745  INFO 9462 --- [           main] com.saffframework.study.task.Application  : Started Application in 11.733 seconds (JVM running for 12.054)
```

图 15-1

程序运行起来之后，我们分别来体验顺序执行和并发执行的效果。

顺序执行多个任务，http://localhost:8080/tasks/sequential?task=3&task=2&task=5，每个任务所花费的时间分别是 3s、2s 和 5s。最后，一共花费了 10.002s，如图 15-2 所示。

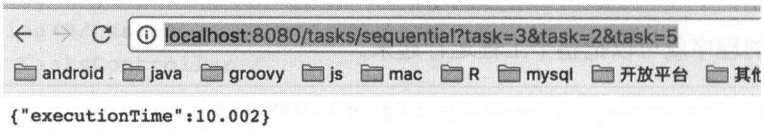


图 15-2

两个线程并发执行多个任务，http://localhost:8080/tasks/concurrent?task=3&task=2&task=5&threads=2。

每个任务所花费的时间同样分别是 3s、2s 和 5s。最后，一共花费了 7.008s，如图 15-3 所示。

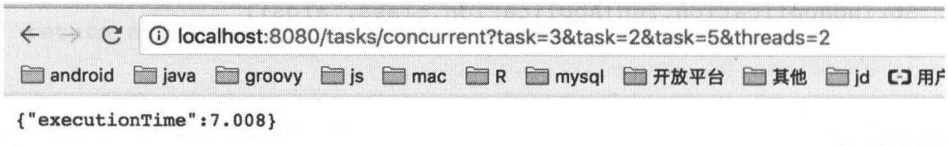


图 15-3

三个线程并发执行多个任务，http://localhost:8080/tasks/concurrent?task=3&task=2&task=5&threads=3。

与前面的区别是，三个线程并发执行。同样是三个任务，完成时间更快了。最后，一共花费了 5.001s，如图 15-4 所示。

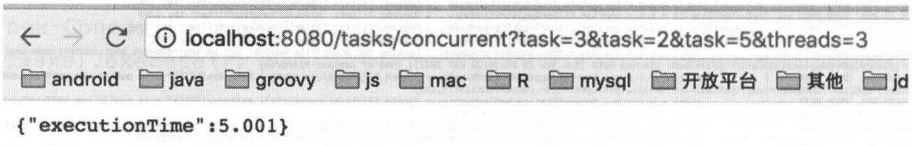


图 15-4

当然，即使是并发执行，线程数也不是越多越好，超过 3 个线程其实跟 3 个线程的执行速度是一样的，因为这里总共只有 3 个任务。

15.2　构建一个给爬虫使用的代理 IP 池

1. 设计一个代理 IP 池

做网络爬虫时，一般对代理 IP 的需求量比较大。因为在爬取网站信息的过程中，很多网站做了反爬虫策略，可能会对每个 IP 做频次控制。这样我们在爬取网站时就需要用到很多代理 IP。

代理 IP 的获取，可以从以下几个途径得到：

◎　从免费的网站上获取，质量很低，能用的 IP 极少。
◎　购买收费的代理服务，质量高很多。
◎　自己搭建代理服务器，稳定，但需要大量的服务器资源。

本节讲述的代理 IP 池是通过爬虫从多个免费网站上获取代理 IP 之后，再检查判断 IP 是否可用，可用就存放到 MongoDB 中，最后展示到前端页面上。

2. 数据存 MongoDB

MongoDB 是一个基于分布式文件存储的数据库。由 C++语言编写。旨在为 Web 应用提供可扩展的高性能数据存储解决方案。 MongoDB 是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。它支持的数据结构非常松散，是类似 JSON 的 BSON 格式，因此可以存储较为复杂的数据类型。MongoDB 最大的特点是它支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

它的特点是高性能、易部署、易使用，存储数据非常方便。主要功能特性有：

◎　面向集合存储，易存储对象类型的数据。
◎　模式自由。
◎　支持动态查询。
◎　支持完全索引，包含内部对象。
◎　支持查询。
◎　支持复制和故障恢复。
◎　使用高效的二进制数据存储，包括大型对象（如视频等）。
◎　自动处理碎片，以支持云计算层次的扩展性。
◎　支持 Ruby、Python、Java、C++、PHP、C#等多种语言。

◎ 文件存储格式为 BSON（一种 JSON 的扩展）。
◎ 可通过网络访问。

到目前为止，MongoDB 是一个新的和普遍使用的数据库。 它是一个基于文档的非关系数据库提供程序。虽然它比传统的数据库快 100 倍，但早期说它将广泛地取代传统的 RDBMS。但是，不可否认的是，在性能和可扩展性方面，MongoDB 有着明显的优势。关系数据库具有典型的架构设计，可以显示表的数量以及这些表之间的关系，而在 MongoDB 中则没有关系的概念。

MongoDB 的优点如下。

◎ MongoDB 的架构较少。它是一个文档数据库，它的一个集合持有不同的文档。
◎ 从一个文档到另一个的文档的数量、内容和大小可能有差异。
◎ MongoDB 中单个对象的结构很清晰。
◎ MongoDB 中没有复杂的连接。
◎ MongoDB 提供深度查询的功能，因为它支持对文档的强大的动态查询。
◎ MongoDB 很容易扩展。
◎ 它使用内部存储器来存储工作集，这是其快速访问的原因。

MongoDB 正因为它的使用方便、重量轻/轻量级、比 RDBMS 快得多等特性，被越来越多的互联网项目所采用。

3. 具体实现

获取代理的核心代码是 ProxyManager，它采用 RxJava 2 来实现，主要做了以下几件事。

（1）创建 ParallelFlowable，针对每一个提供免费代理 IP 的页面并行地抓取。本书的 10.2 节中曾经讲述过 ParallelFlowable 的内容。

```
Flowable.fromIterable(ProxyPool.proxyMap.keySet())
                .parallel()
```

（2）针对每一个页面进行抓取，返回 List<Proxy>对象。

```
map(new Function<String, List<Proxy>>() {
                    @Override
                    public List<Proxy> apply(String s) throws Exception {
```

```
                try {
                    return new ProxyPageCallable(s).call();
                } catch (Exception e) {
                    e.printStackTrace();
                }

                return null;
            }
        })
```

（3）对每一个页面获取的代理 IP 列表进行依次校验，判断是否可用。

```
flatMap(new Function<List<Proxy>, Publisher<Proxy>>() {
            @Override
            public Publisher<Proxy> apply(List<Proxy> proxies) throws
Exception {

                if (proxies == null) return null;

                List<Proxy> result = proxies
                        .stream()
                        .parallel()
                        .filter(new Predicate<Proxy>() {
                    @Override
                    public boolean test(Proxy proxy) {

                        HttpHost httpHost = new HttpHost(proxy.getIp(),
proxy.getPort(), proxy.getType());
                        return HttpManager.get().checkProxy(httpHost);
                    }
                }).collect(Collectors.toList());

                return Flowable.fromIterable(result);
            }
        })
```

（4）依次保存到 proxyList。

```
subscribe(new Consumer<Proxy>() {
            @Override
            public void accept(Proxy proxy) throws Exception {
```

```
                    log.debug("Result Proxy =
"+proxy.getType()+"://"+proxy.getIp()+":"+proxy.getPort());
                    proxy.setLastSuccessfulTime(new Date().getTime());
                    ProxyPool.proxyList.add(proxy);
                }
            });
```

完整的流程图如图 15-5 所示。

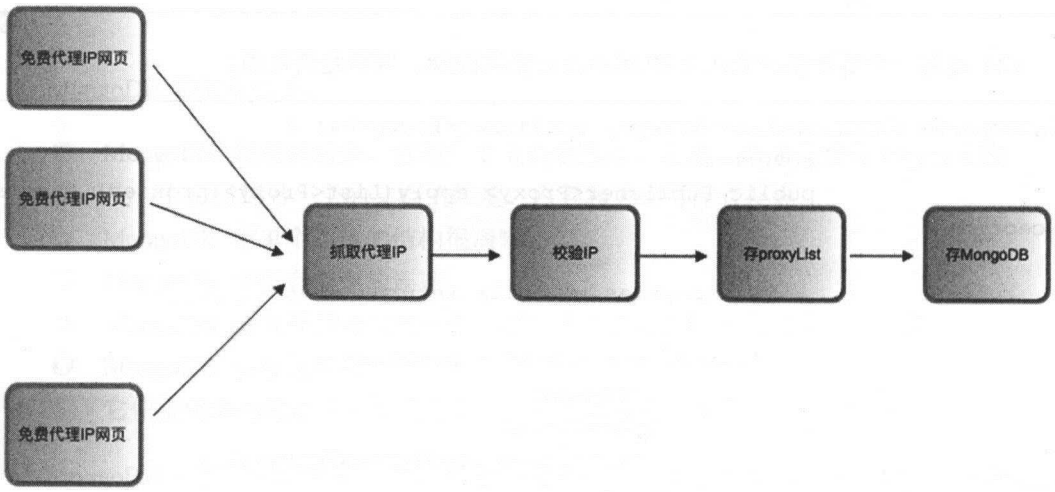


图 15-5

完整的 ProxyManager 代码如下。

```
import com.cv4j.proxy.domain.Proxy;
import com.cv4j.proxy.http.HttpManager;
import com.cv4j.proxy.task.ProxyPageCallable;
import com.safframework.tony.common.utils.Preconditions;
import io.reactivex.Flowable;
import io.reactivex.functions.Consumer;
import io.reactivex.functions.Function;
import lombok.extern.slf4j.Slf4j;
import org.apache.http.HttpHost;
import org.reactivestreams.Publisher;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
```

```
import java.util.function.Predicate;
import java.util.stream.Collectors;

/**
 * Created by tony on 2017/10/25.
 */
@Slf4j
public class ProxyManager {

    private ProxyManager() {
    }

    public static ProxyManager get() {
        return ProxyManager.Holder.MANAGER;
    }

    private static class Holder {
        private static final ProxyManager MANAGER = new ProxyManager();
    }

    /**
     * 抓取代理，成功的代理存放到 ProxyPool 中
     */
    public void start() {

        Flowable.fromIterable(ProxyPool.proxyMap.keySet())
                .parallel()
                .map(new Function<String, List<Proxy>>() {
                    @Override
                    public List<Proxy> apply(String s) throws Exception {

                        try {
                            return new ProxyPageCallable(s).call();
                        } catch (Exception e) {
                            e.printStackTrace();
                        }

                        return new ArrayList<Proxy>();
                    }
                })
                .flatMap(new Function<List<Proxy>, Publisher<Proxy>>() {
                    @Override
```

```
                public Publisher<Proxy> apply(List<Proxy> proxies) throws
Exception {
                    if (Preconditions.isNotBlank(proxies)) {
                        List<Proxy> result = proxies
                                .stream()
                                .parallel()
                                .filter(new Predicate<Proxy>() {
                                    @Override
                                    public boolean test(Proxy proxy) {
                                        HttpHost httpHost = new
HttpHost(proxy.getIp(), proxy.getPort(), proxy.getType());
                                        boolean result =
HttpManager.get().checkProxy(httpHost);
                                        if(result) log.info("checkProxy " +
proxy.getProxyStr() +", "+result);
                                        return result;
                                    }
                                }).collect(Collectors.toList());

                        return Flowable.fromIterable(result);
                    }

                    return Flowable.empty();
                }
            })
            .sequential()
            .subscribe(new Consumer<Proxy>() {
                @Override
                public void accept(Proxy proxy) throws Exception {

                    if (proxy!=null) {
                        log.info("accept " + proxy.getProxyStr());
                        proxy.setLastSuccessfulTime(new Date().getTime());
                        ProxyPool.proxyList.add(proxy);
                    }
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    log.error("ProxyManager is error:
"+throwable.getMessage());
                }
```

```
        });
    }
}
```

4. 定时任务

每隔几个小时跑一次定时任务，在完成抓取任务之后先删除旧的数据，然后再把新的数据插到 MongoDB 中。这里使用 Spring 自带的定时任务，使用起来很方便。在任务类上标注@Component 注解，然后在需要执行定时任务的方法上加上@Scheduled 即可。

```
import com.cv4j.proxy.ProxyManager;
import com.cv4j.proxy.ProxyPool;
import com.cv4j.proxy.domain.Proxy;
import com.cv4j.proxy.web.config.Constant;
import com.cv4j.proxy.web.dao.CommonDao;
import com.cv4j.proxy.web.dao.ProxyDao;
import com.cv4j.proxy.web.domain.JobLog;
import com.cv4j.proxy.web.domain.ProxyData;
import com.safframework.tony.common.utils.JodaUtils;
import com.safframework.tony.common.utils.Preconditions;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.CacheManager;
import org.springframework.scheduling.annotation.Scheduled;
import org.springframework.stereotype.Component;

import java.util.ArrayList;
import java.util.Date;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.atomic.AtomicInteger;

/**
 * Created by tony on 2017/11/22.
 */
@Component
@Slf4j
public class ScheduleJobs {


    public final static int JOB_STATUS_INIT = 0;
```

```
    public final static int JOB_STATUS_RUNNING = 1;
    public final static int JOB_STATUS_STOPPED = 2;

    protected AtomicInteger stat = new AtomicInteger(JOB_STATUS_INIT);

    @Autowired
    ProxyDao proxyDao;

    @Autowired
    CommonDao commonDao;

    @Autowired
    CacheManager cacheManager;

    ProxyManager proxyManager = ProxyManager.get();

    /**
     * 每隔几个小时跑一次任务
     */
    @Scheduled(cron="${cronJob.schedule}")
    public void cronJob() {

        //1.检查 job 的状态
        checkRunningStat();

        log.info("Job Start...");

        //2.获取目标网页的 URL
        ProxyPool.proxyMap = proxyDao.getProxyMap();

        //3.如果数据库里没取到，用默认内置的
        if(Preconditions.isBlank(ProxyPool.proxyMap)) {
            log.info("Job proxyDao.getProxyMap() is empty");
            ProxyPool.proxyMap = Constant.proxyMap;
        }

        //4.每次跑 job 先清空缓存中的内容
        if (cacheManager.getCache("proxys")!=null) {

            cacheManager.getCache("proxys").clear();
        }
```

```
        //5.创建一个日志对象，用于存储 job 的每次工作记录
        JobLog jobLog = new JobLog();
        jobLog.setJobName("ScheduleJobs.cronJob");
        jobLog.setStartTime(JodaUtils.formatDateTime(new Date()));

        //6.跑任务之前先清空 proxyList 中上一次 job 留下的 proxy 数据
        ProxyPool.proxyList.clear();

        //7.从数据库中选取 10 个代理作为种子代理，遇到 http 503 时使用代理来抓数据
        ProxyPool.addProxyList(getProxyList(proxyDao.takeRandomTenProxy()));
        log.info("Job ProxyPool.proxyList size = "+ProxyPool.proxyList.size());
        //8.正式开始，爬代理数据
        proxyManager.start();

        //9.爬完以后，把数据转换为 ProxyData 并存到数据库
        CopyOnWriteArrayList<ProxyData> list =
getProxyDataList(ProxyPool.proxyList);
        log.info("Job ProxyData list size = "+list.size());
        if (Preconditions.isNotBlank(list)) {

            // 10. list 的数量<=15 时，不删除数据库里的老数据
            if (list.size()>15) {
                proxyDao.deleteAll();
                log.info("Job after deleteAll");
            }

            //11. 然后再进行插入新的 proxy
            for (ProxyData p:list) {
                proxyDao.saveProxy(p);
            }
            log.info("Job save count = "+list.size());

            jobLog.setResultDesc(String.format("success save count = %s",
list.size()));
            jobLog.setEndTime(JodaUtils.formatDateTime(new Date()));
            commonDao.saveJobLog(jobLog);

        } else {
            log.info("Job proxyList is empty...");
        }

        //12. 设置 job 状态为停止
```

```
        stop();

        log.info("Job End...");
    }

    private void checkRunningStat() {
        while (true) {

            int statNow = getJobStatus();

            //如果已经在运行了，就抛出异常，结束循环
            if (statNow == JOB_STATUS_RUNNING) {
                throw new IllegalStateException("Job is already running!");
            }

            //如果还没在运行，就设置为运行状态，结束循环
            if (stat.compareAndSet(statNow, JOB_STATUS_RUNNING)) {
                break;
            }
        }
    }

    public int getJobStatus() {

        return stat.get();
    }

    public void stop() {
        //状态从 JOB_STATUS_RUNNING 更新为 JOB_STATUS_STOPPED，代表停止 job
        stat.compareAndSet(JOB_STATUS_RUNNING, JOB_STATUS_STOPPED);
    }

    private List<Proxy> getProxyList(List<ProxyData> list) {
        List<Proxy> resultList = new ArrayList<>();

        Proxy proxy = null;
        for(ProxyData proxyData : list) {
            proxy = new Proxy();
            proxy.setType(proxyData.getProxyType());
            proxy.setIp(proxyData.getProxyAddress());
            proxy.setPort(proxyData.getProxyPort());
```

```
            resultList.add(proxy);
        }

        return resultList;
    }

    private CopyOnWriteArrayList<ProxyData> getProxyDataList(List<Proxy> list)
{
        CopyOnWriteArrayList<ProxyData> resultList = new
CopyOnWriteArrayList<>();

        ProxyData proxyData = null;
        for(Proxy proxy : list) {
            proxyData = new ProxyData();
            proxyData.setProxyType(proxy.getType());
            proxyData.setProxyAddress(proxy.getIp());
            proxyData.setProxyPort(proxy.getPort());

            resultList.add(proxyData);
        }

        return resultList;
    }
}
```

另外，不要忘记在项目的 Application 上加上@EnableScheduling，否则无法执行定时任务。

5. 展示到前端

整个项目使用 Spring Boot 搭建，运行起来之后本地访问地址：http://localhost:8080/proxypool/ proxy_list。

预览效果如图 15-6 所示。

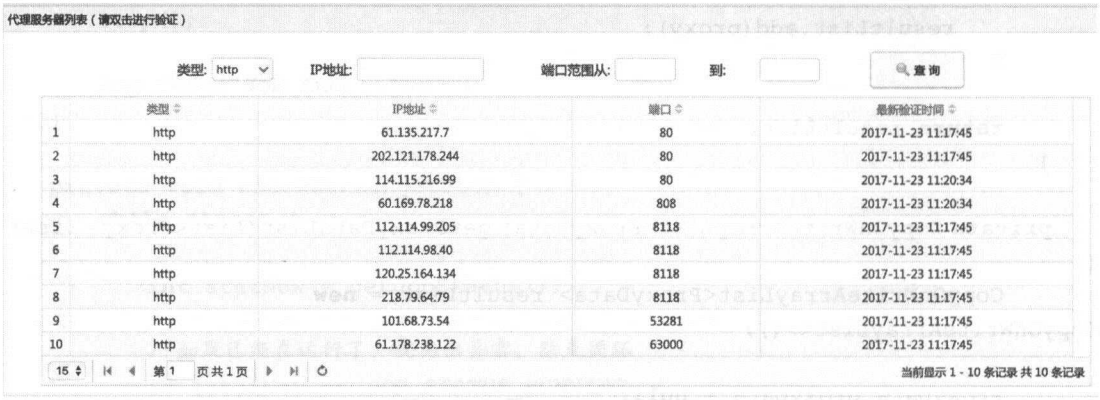


图 15-6

在使用前，还可以再做一次检测，只需双击某个代理 IP 即可，如图 15-7 所示。

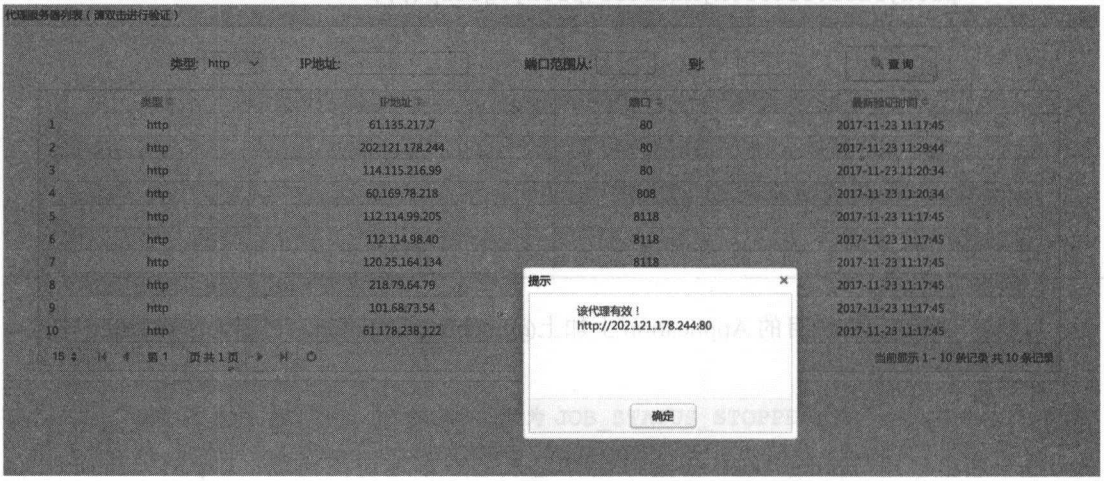


图 15-7

在第二次检测时，已经失效的 IP 会被 ProxyPool 删除。

在做爬虫项目时，自己维护一个可用的代理 IP 池是很有必要的事情，当然要想追求更高稳定性的代理 IP 还是考虑购买比较好。

本节的工程可以在 GitHub 上找到。笔者自己在做项目时也会用到它。

https://github.com/fengzhizi715/ProxyPool

15.3　小结

本章介绍了两个 Spring Boot 和 RxJava 整合的例子。这两个例子都非常具有代表性，RxJava 的操作符、异步、并行等给开发带来了便利，以及思维上的转变，再结合 Spring Boot 可以大大提高开发的效率。

另外，Spring 5 以及 Spring Boot 2 中会增加 Spring Webflux，它是一个 Reactive 风格编程的异步非阻塞开发框架。Spring Webflux 使用的是 Reactor，但是也支持使用 RxJava 以及 Java 8 的 CompletableFuture。

第16章

Java 8的函数式编程

16.1 Java 8 的新变化

Java 8 是自 Java 5 发布以来 Java 语言最大的一次版本升级，Java 8 带来了很多新特性，比如编译器、类库、开发工具和 JVM（Java 虚拟机）。除此之外，还有编程思想上的变化，传统的 Java 推崇的是面向对象的编程方式，Java 8 之后开始支持函数式编程，函数式编程也是未来 Java 语言的发展方向。

图 16-1 是笔者整理出来的一张思维导图，大致涵盖了 Java 8 的新特性。

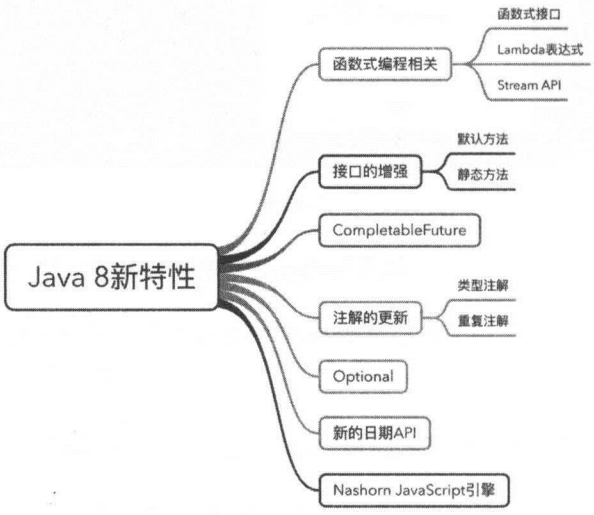


图 16-1

因为本书不是专门介绍 Java 8 的书籍，所以在本章接下来的各个小节中，会着重介绍函数式编程相关的内容以及 CompletableFuture。如果你对 Java 8 的各个特性都非常感兴趣，推荐你阅读《Java 8 实战》。

16.2　函数是一等公民

Java 8 引入了函数式编程。函数式编程的重点在于函数，函数变成了 Java 世界里的一等公民。函数和其他值一样，可以到处被定义，既可以作为参数传入另一个函数，也可以作为函数的返回值，返回给调用者。利用这些特性，既可以灵活组合已有函数形成新的函数，又可以在更高层次上对问题进行抽象，也就是使用高阶函数。

在数学和计算机科学中，高阶函数是至少满足下列任意条件的函数：

◎　接受一个或多个函数作为输入；
◎　输出一个函数。

高阶函数，又称算子（运算符）或泛函，是包含多于一个箭头的函数。微积分中的导数就是常见的例子，因为它映射一个函数到另一个函数。在无类型 Lambda 演算，所有函数都是高阶的；在有类型 Lambda 演算（大多数函数式编程语言都是从中演化而来的）中，高阶函数一般是那些函数别包含多于一个箭头的函数。在函数式编程中，返回另一个函数的高阶函数被称为柯里化函数。在很多函数式编程语言中都能找到的 map 函数是高阶函数的一个例子。它接受一个函数 f 作为参数，列表并应用 f 到它的每个元素的一个函数。

使用高阶函数之前，求和、求平方和、求立方和的写法如下：

```
public class TestHighOrderFunction {

  public static int identity(int x) {
    return x;
  }

  public static int sum_integers(int a, int b) {
    int sum = 0;
    for (int i = a; i <= b; i++) {
      sum += identity(i);
    }
```

```
    return sum;
  }

  public static int square(int x) {
    return x * x;
  }

  public static int sum_square(int a, int b) {
    int sum = 0;
    for (int i = a; i <= b; i++) {
      sum += square(i);
    }
    return sum;
  }

  public static double cube(int x) {
    return x * x * x;
  }

  public static int sum_cubes(int a, int b) {
    int sum = 0;
    for (int i = a; i <= b; i++) {
      sum += cube(i);
    }
    return sum;
  }

  public static void main(String[] a) {

    System.out.println(sum_integers(1, 10)); // return 55
    System.out.println(sum_square(1, 10));  // return 385
    System.out.println(sum_cubes(1, 10));  // return 3025
  }
}
```

在以 sum_开头的方法里，代码很类似，三者唯一的区别在于：

```
sum += identity(i);
sum += square(i);
sum += cube(i);
```

在软件工程里有 DRY（don't repeat yourself）准则。下面看看如何使用高阶函数优化刚才的

这些代码：

```
public interface Function {
    int opera(int a);
}

public static void main(String[] a) {

   Function identity = x->x;
   Function square = x->x*x;
   Function cube = x -> x*x*x;
   System.out.println(sum(identity, 1,10)); // return 55
   System.out.println(sum(square, 1,10)); // return 385
   System.out.println(sum(cube, 1,10));   // return 3025
}

public static int sum(Function term, int a, int b) {

    int sum = 0;
    for (int i = a; i <= b; i++) {
       sum += term.opera(i);
    }
    return sum;
}
```

得到的结果与上面的 TestHighOrderFunction 类中运行的结果相同。不过，这里的 sum 方法中使用了：

```
  sum += term.opera(i);
```

取代了原先的代码。term.opera(i)对应的是 identity(i)、square(i)、cube(i)，在这里 Function 函数被当作参数进行传递。这就是高阶函数的特性。

对于 for 循环，我们也能用更优雅的方式进行优化，下面使用了递归的方式。

```
public interface Function {
    int opera(int a);
}

public static void main(String[] a) {
```

```
    Function identity = x->x;
    Function square = x->x*x;
    Function cube = x -> x*x*x;
    Function inc = x->x+1; // 定义 next 函数
    System.out.println(sum(identity, 1,inc,10)); // return 55
    System.out.println(sum(square, 1,inc,10)); // return 385
    System.out.println(sum(cube, 1,inc,10));  // return 3025
}

public static int sum(Function term, int a, Function next, int b) {

    if (a>b) {
        return 0;
    } else {
        return term.opera(a) + sum(term, next.opera(a), next, b);
    }
}
```

16.3 Lambda 表达式

在 Java 8 之前我们使用 Thread 时，可能是这样的写法：

```
new Thread(new Runnable() {
     public void run() {
          System.out.println("test");
     }
}).start();
```

由于 Java 8 引入了 Lambda 表达式，因此可以这样写：

```
new Thread(()->System.out.println("test"));
```

Lambda 表达式源于 Lambda 演算。Lambda 演算可以被称为最小的通用程序设计语言，它包括一条变换规则（变量替换）和一条函数定义方式，Lambda 演算的通用之处在于，任何一个可计算函数都能用这种形式来表达和求值。因而，它是等价于图灵机的。尽管如此，Lambda 演算强调的是变换规则的运用，而非实现它们的具体机器。可以认为这是一种更接近软件而非硬件的方式。

当单击 Runnable 的源码时，发现 Runnable 使用了@FunctionalInterface，这在 Java 8 之前是

没有的。

```
@FunctionalInterface
public interface Runnable {
    /**
     * When an object implementing interface <code>Runnable</code> is used
     * to create a thread, starting the thread causes the object's
     * <code>run</code> method to be called in that separately executing
     * thread.
     * <p>
     * The general contract of the method <code>run</code> is that it may
     * take any action whatsoever.
     *
     * @see     java.lang.Thread#run()
     */
    public abstract void run();
}
```

@FunctionalInterface 是 Java 8 为函数式接口引入的一个新的注解，表明该接口是函数式接口，它只包含唯一一个抽象方法。任何可以接受一个函数式接口实例的地方，都可以用 Lambda 表达式。

再来看一个匿名函数的例子。

```
button.setOnClickListener(new Button.OnClickListener(){
    public void onClick(View v) {
        Log.i(TAG,"点击 button");
    }
});
```

将匿名函数改成 Lambda 表达式。

```
button.setOnClickListener((v)-> Log.i(TAG,"点击 button"));
```

这样改造的好处在于，Lambda 对象的创建是通过字节码指令 invokedynamic 来完成的，减少了类型和实例的创建消耗。而匿名类需要创建新的对象。

1. JDK 中的函数式接口举例

java.lang.Runnable,

java.awt.event.ActionListener,

java.util.Comparator,

java.util.concurrent.Callable

java.util.function 包下的接口，如 Consumer、Predicate、Supplier 等。

2. 简化的 Lambda——方法引用（Method Reference）

Lambda 已经简化了代码的写法，然而方法引用进一步简化了 Lambda 的写法。方法引用的使用方式是类名::方法名，如表 16-1 所示。

表 16-1

类　型	使用方式	备　注
引用静态方法	ContainingClass::staticMethodName	Integer::valueOf 是 i->Integer.valueOf(i)的简化写法
引用特定对象的实例方法	containingObject::instanceMethodName	s::toString()是()->s.toString()的简化写法
引用特定类型的任意对象的实例方法	ContainingType::methodName	System.out::println 是(s)->System.out.println(s)的简化写法，其中 System.out 表示的是 PrintStream 对象
引用构造函数	ClassName::new	String::new 是()->new String()的简化写法

下面来看一个简单的例子，对某些 User 对象按照 name 进行排序，最初我们会这样写：

```
User u1 = new User("tony");
User u2 = new User("cafei");
User u3 = new User("aaron");

List<User> users = Arrays.asList(u1,u2,u3);

Collections.sort(users, new Comparator<User>(){

 @Override
 public int compare(User u1, User u2) {
  return u1.getName().compareTo(u2.getName());
 }

});
```

在 Java 8 以后，Comparator 增加了一个静态方法 comparing(Function<? super T, ? extends U> keyExtractor)，我们可以把排序的写法简化为：

```
Collections.sort(users, Comparator.comparing((User u)->u.getName()));
```

如果使用方法引用，代码可更加简化：

```
Collections.sort(users,Comparator.comparing(User::getName));
```

16.4　Java 8 新增的 Stream

16.4.1　Stream 的特性

Stream 是 Java 8 新增的接口，Stream 可以认为是一个高级版本的 Iterator。它代表着数据流，流中的数据元素的数量可以是有限的，也可以是无限的。

Stream 与 Iterator 的差别如下。

- ◎ 无存储：Stream 是基于数据源的对象，它本身不存储数据元素，而是通过管道将数据源的元素传递给操作。
- ◎ 函数式编程：对 Stream 的任何修改都不会修改背后的数据源，比如对 Stream 执行 filter 操作并不会删除被过滤的元素，而是会产生一个不包含被过滤元素的新的 Stream。
- ◎ 延迟执行：Stream 的操作由零个或多个中间操作（intermediate operation）和一个结束操作（terminal operation）两部分组成。只有执行了结束操作，Stream 定义的中间操作才会依次执行，这就是 Stream 的延迟特性。
- ◎ 可消费性：Stream 只能被“消费”一次，一旦遍历过就会失效。就像容器的迭代器那样，想要再次遍历则必须重新生成一个新的 Stream。

16.4.2　Java 8 新增的函数式接口

Stream 的操作是建立在函数式接口的组合之上的。Java 8 中新增的函数式接口都在 java.util.function 包下。这些函数式接口可以有多种分类方式，如图 16-2 和图 16-3 所示。

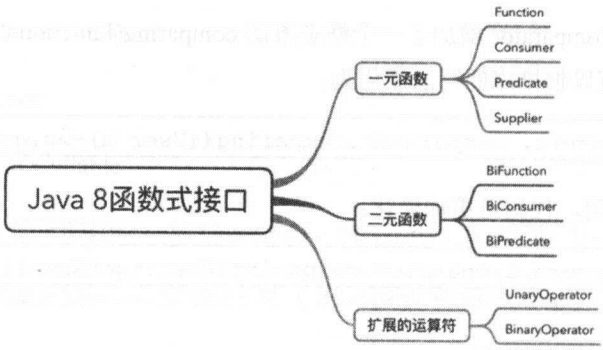


图 16-2

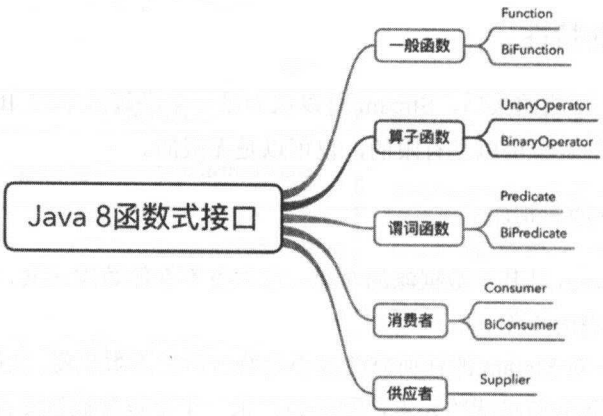


图 16-3

1. Function

Function 是从 T 到 R 的一元映射函数。将参数 T 传递给一个函数，返回 R。即 R = Function(T)。

```
@FunctionalInterface
public interface Function<T, R> {

    /**
     * Applies this function to the given argument.
     *
     * @param t the function argument
     * @return the function result
     */
    R apply(T t);
```

```
    /**
     * Returns a composed function that first applies the {@code before}
     * function to its input, and then applies this function to the result.
     * If evaluation of either function throws an exception, it is relayed to
     * the caller of the composed function.
     *
     * @param <V> the type of input to the {@code before} function, and to the
     *           composed function
     * @param before the function to apply before this function is applied
     * @return a composed function that first applies the {@code before}
     * function and then applies this function
     * @throws NullPointerException if before is null
     *
     * @see #andThen(Function)
     */
    default <V> Function<V, R> compose(Function<? super V, ? extends T> before)
{
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    /**
     * Returns a composed function that first applies this function to
     * its input, and then applies the {@code after} function to the result.
     * If evaluation of either function throws an exception, it is relayed to
     * the caller of the composed function.
     *
     * @param <V> the type of output of the {@code after} function, and of the
     *           composed function
     * @param after the function to apply after this function is applied
     * @return a composed function that first applies this function and then
     * applies the {@code after} function
     * @throws NullPointerException if after is null
     *
     * @see #compose(Function)
     */
    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)
{
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }
```

```
    /**
     * Returns a function that always returns its input argument.
     *
     * @param <T> the type of the input and output objects to the function
     * @return a function that always returns its input argument
     */
    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

Function 默认实现了 3 个 default 方法，分别是 compose、andThen 和 identity，如表 16-2 所示。

表 16-2

方法名	对应函数	描述
compose	V=Function(ParamFunction(T))	它体现了嵌套关系
andThen	V= ParamFunction(Function(T))	转换了嵌套的顺序
identity	Function(T)=T	传递自身的函数调用

compose 和 andThen 对于两个函数 f 和 g 来说，f.compose(g)等价于 g.andThen(f)。

2. Predicate

Predicate 是一个谓词函数，主要作为一个谓词演算推导真假值存在，返回布尔值。Predicate 等价于一个 Function 的 boolean 型返回值的子集。

```
@FunctionalInterface
public interface Predicate<T> {

    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches the predicate,
     * otherwise {@code false}
     */
    boolean test(T t);

    /**
```

```
     * Returns a composed predicate that represents a short-circuiting logical
     * AND of this predicate and another.  When evaluating the composed
     * predicate, if this predicate is {@code false}, then the {@code other}
     * predicate is not evaluated.
     *
     * <p>Any exceptions thrown during evaluation of either predicate are relayed
     * to the caller; if evaluation of this predicate throws an exception, the
     * {@code other} predicate will not be evaluated.
     *
     * @param other a predicate that will be logically-ANDed with this
     *              predicate
     * @return a composed predicate that represents the short-circuiting logical
     * AND of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }

    /**
     * Returns a predicate that represents the logical negation of this
     * predicate.
     *
     * @return a predicate that represents the logical negation of this
     * predicate
     */
    default Predicate<T> negate() {
        return (t) -> !test(t);
    }

    /**
     * Returns a composed predicate that represents a short-circuiting logical
     * OR of this predicate and another.  When evaluating the composed
     * predicate, if this predicate is {@code true}, then the {@code other}
     * predicate is not evaluated.
     *
     * <p>Any exceptions thrown during evaluation of either predicate are relayed
     * to the caller; if evaluation of this predicate throws an exception, the
     * {@code other} predicate will not be evaluated.
     *
     * @param other a predicate that will be logically-ORed with this
```

```
     *              predicate
     * @return a composed predicate that represents the short-circuiting logical
     * OR of this predicate and the {@code other} predicate
     * @throws NullPointerException if other is null
     */
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }

    /**
     * Returns a predicate that tests if two arguments are equal according
     * to {@link Objects#equals(Object, Object)}.
     *
     * @param <T> the type of arguments to the predicate
     * @param targetRef the object reference with which to compare for equality,
     *               which may be {@code null}
     * @return a predicate that tests if two arguments are equal according
     * to {@link Objects#equals(Object, Object)}
     */
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
                ? Objects::isNull
                : object -> targetRef.equals(object);
    }
}
```

Predicate 的默认方法是 and、negate、or。

3. Consumer

Consumer 是从 T 到 void 的一元函数，接受一个入参但不返回任何结果的操作。

```
@FunctionalInterface
public interface Consumer<T> {

    /**
     * Performs this operation on the given argument.
     *
     * @param t the input argument
     */
    void accept(T t);
```

```
    /**
     * Returns a composed {@code Consumer} that performs, in sequence, this
     * operation followed by the {@code after} operation. If performing either
     * operation throws an exception, it is relayed to the caller of the
     * composed operation.  If performing this operation throws an exception,
     * the {@code after} operation will not be performed.
     *
     * @param after the operation to perform after this operation
     * @return a composed {@code Consumer} that performs in sequence this
     * operation followed by the {@code after} operation
     * @throws NullPointerException if {@code after} is null
     */
    default Consumer<T> andThen(Consumer<? super T> after) {
        Objects.requireNonNull(after);
        return (T t) -> { accept(t); after.accept(t); };
    }
}
```

Consumer 的默认方法是 andThen。

4. Supplier

Supplier 是表示结果的供应者。

```
@FunctionalInterface
public interface Supplier<T> {

    /**
     * Gets a result.
     *
     * @return a result
     */
    T get();
}
```

Supplier 的用法如下：

```
        f(x,y,z) = (x+y)×z
        Supplier<String> supplier = new Supplier<String>() {
            @Override
            public String get() {
```

```
            return "hello suppiler";
        }
    };
    System.out.println(supplier.get());
```

或者：

```
Supplier<User> userSupplier = User::new;
userSupplier.get();   // new User
```

Java 8 新增了 CompletableFuture，它的很多方法的入参都用到了 Supplier。

16.4.3 Stream 用法

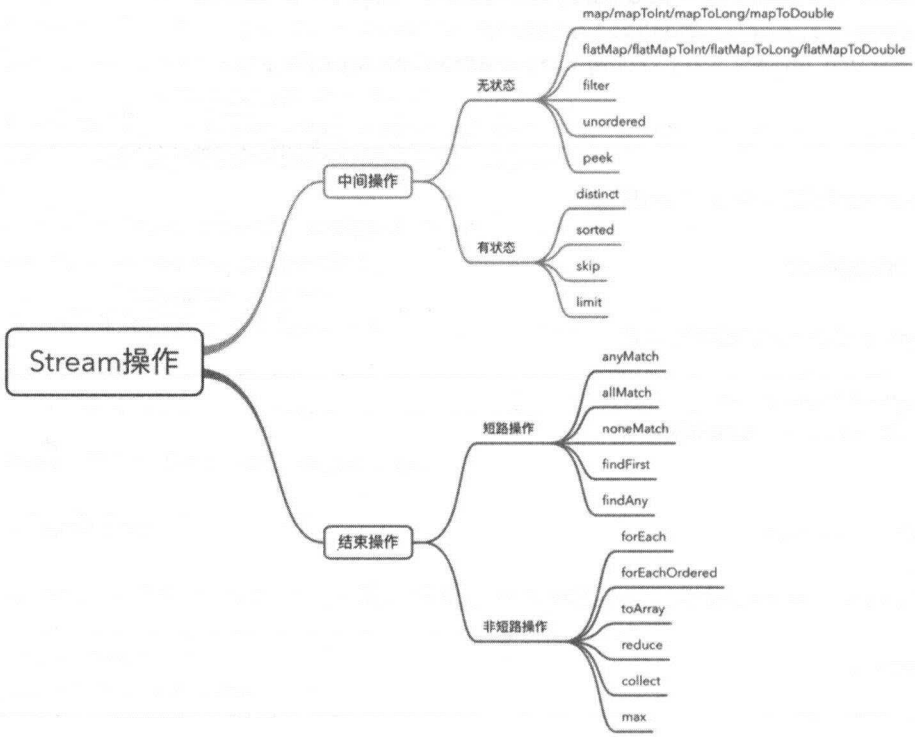


图 16-4

1. Stream 的创建

Java 8 有多种方式来创建 Stream：

- 通过集合的 stream()方法或者 parallelStream()。
- 使用流的静态方法，比如 Stream.of(Object[]), IntStream.range(int, int) 或者 Stream.iterate(Object, UnaryOperator)。
- 通过 Arrays.stream(Object[])方法。
- BufferedReader.lines()从文件中获得行的流。
- Files 类的操作路径的方法，如 list、find、walk 等。
- 随机数流 Random.ints()。
- 其他一些类提供了创建流的方法，如 BitSet.stream(),Pattern.splitAsStream(java.lang.CharSequence)和 JarFile.stream()。

其实最终都是依赖底层的 StreamSupport 类来完成 Stream 创建。

2. 中间操作

中间操作又可以分为无状态的（Stateless）和有状态的（Stateful），无状态中间操作是指元素的处理不受前面元素的影响，而有状态的中间操作必须等到所有元素处理之后才知道最终结果。

Stream 的中间操作只是一种标记，只有执行了结束操作才会触发实际计算。熟悉 RxJava、Scala 的读者可以看到，Stream 中间操作的各个方法在 RxJava、Scala 中都可以找到熟悉的身影。

3. 结束操作

（1）短路操作

短路操作是指不用处理全部元素就可以返回结果。短路操作必须一个元素处理一次。

（2）非短路操作

非短路操作可以批量处理数据，但是需要处理完全部元素才会返回结果。

16.4.4　并行流

在创建 Stream 时，默认是创建串行流。但是可以使用 parallelStream()来创建并行流，或者用 parallel()将串行流转换成并行流。并行流也可以通过 sequential()转换成串行流。

Java 8 Stream 的并行流，本质上还是使用 Fork/Join 模型。

在 Java 开发中，如果使用了 Java 8，那么强烈建议使用 Stream。因为 Stream 的每个操作都可以依赖 Lambda 表达式，它是一种声明式的数据处理方式，并且 Stream 提高了数据处理效率和开发效率。

16.5 函数的柯里化

在计算机科学中，柯里化（Currying），是把接受多个参数的函数变换成接受一个单一参数（最初函数的第一个参数）的函数，并且返回接受余下的参数且返回结果的新函数的技术。这个技术由克里斯托弗·斯特雷奇以逻辑学家哈斯凯尔·加里命名的，尽管它是 Moses Schönfinkel 和戈特洛布·弗雷格发明的。

在函数式编程中，函数的概念与数学中函数的概念相同，类似于“映射”。高阶函数和柯里化是函数式编程的特性。

对于柯里化而言，首先我们来举个例子，先定义这样一个函数：

```
f(x,y,z) = (x+y)^z
```

当 x 是一个常量时，比如 x=4，可以用一个新的函数来代替：

```
f(4,y,z)=g(y,z)=(4+y)^z
```

新的函数 $g(y,z)$是由 $f(x,y,z)$转换而来的，它的参数 y、z 是原先函数的后两个参数。

我们再一次对 y 赋值，比如 y=5，函数再次变成：

```
f(4,5,z)=g(4,5)=(4+5)^z
```

我们可以理解为将原来的函数变量拆分开来调用：

```
f(x,y,z) -> f(x)(y)(z)
```

1. 借助 Java 8 实现柯里化

孔乙己中茴香豆的“茴”字有四种写法，我们也给出多种方式来实现柯里化。

◎ 第一种方式，嵌套多层 Function：

```
Function<Integer, Function<Integer, Function<Integer, Integer>>> currying =  x
-> y -> z -> (x+y)*z;

System.out.println(currying.apply(4).apply(5).apply(6)); //54
```

◎　第二种方式：

```
IntFunction<IntFunction<IntUnaryOperator>> f = x -> y -> z -> (x + y) * z;

System.out.println(f.apply(4).apply(5).applyAsInt(6)); //54
```

◎　第三种方式，需要先定义一个 TriFunction 函数接口：

```
@FunctionalInterface
public interface TriFunction<U, T, S, R> {

    /**
     * Applies this function to the given arguments.
     * @param <U>
     * @param <T>
     * @param <S>
     * @return the function result
     */
     R apply(T t, U u, S s);
}
```

然后借助 TriFunction 函数来实现柯里化：

```
TriFunction<Integer,Integer,Integer, Integer> triFunction = (x,y,z) -> (x+y)*z;

System.out.println(triFunction.apply(4,5,6)); //54
```

为什么要定义一个 TriFunction 呢？其实 Java 8 的 function 库中包含了 BiFunction 的函数接口，但它只能传两个参数。而 Java 的设计者们也不再定义三个及以上参数的函数接口。可以想象一下，要是真的定义 $n(n>=3)$个参数的函数接口，那么这个函数需要传递 $n+1$ 个参数，其中包含一个返回的参数类型。在 RxJava 1.x 中确实存在可以定义到 9 个参数的 Func9，但写起代码来还是很痛苦的。

◎　第四种方式，借助匿名内部类，每次调用都返回一个新的函数：

```
        Function<Integer, Function<Integer, Function<Integer, Integer>>>
currying = new Function<Integer, Function<Integer, Function<Integer,
Integer>>>() {
            @Override
            public Function<Integer, Function<Integer, Integer>> apply(Integer
x) {
                return new Function<Integer, Function<Integer, Integer>>() {
                    @Override
                    public Function<Integer, Integer> apply(Integer y) {
                        return new Function<Integer, Integer>() {
                            @Override
                            public Integer apply(Integer z) {
                                return (x + y) * z;
                            }
                        };
                    }
                };
            }
        };

        System.out.println(currying.apply(4).apply(5).apply(6));//54
```

2. 柯里化的好处

随着函数在 Java 8 中变成一等公民，自然而然会产生柯里化。柯里化的链式调用的确用起来很畅快。柯里化也可以延迟加载一个函数。

除此之外，柯里化在很多时候简化了函数式编程的复杂性，使得编程更加优雅。当然，若是在团队中使用，也需要充分考虑到团队中其他成员的接受度。

16.6 新的异步编程方式 CompletableFuture

1. Future

JDK 5 引入了 Future 模式。Future 接口是 Java 多线程 Future 模式的实现，在 java.util.concurrent 包中，可以来进行异步计算。

Future 模式是多线程设计中常用的一种设计模式。Future 模式可以理解成：我有一个任务，提交给了 Future，Future 替我完成这个任务。期间我自己可以去做任何想做的事情。一段时间

之后，我便可以从 Future 那儿取出结果。

Future 的接口很简单，只有五个方法。

```
public interface Future<V> {

    boolean cancel(boolean mayInterruptIfRunning);

    boolean isCancelled();

    boolean isDone();

    V get() throws InterruptedException, ExecutionException;

    V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException, TimeoutException;
}
```

Future 接口的方法介绍如下：

- boolean cancel (boolean mayInterruptIfRunning)：取消任务的执行。参数指定是否立即中断任务执行，或者等任务结束。
- boolean isCancelled ()：任务是否已经取消，任务正常完成前将其取消，则返回 true。
- boolean isDone ()：任务是否已经完成。需要注意的是，如果任务正常终止、异常或取消，都将返回 true。
- V get () throws InterruptedException, ExecutionException：等待任务执行结束，然后获得 V 类型的结果。InterruptedException 表示线程被中断异常，ExecutionException 表示任务执行异常。如果任务被取消，还会抛出 CancellationException。
- V get (long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException：同上面的 get 功能一样，只是多了设置超时时间。参数 timeout 指定超时时间，uint 指定时间的单位，在枚举类 TimeUnit 中有相关的定义。如果计算超时，则抛出 TimeoutException。

一般情况下，我们会结合 Callable 和 Future 一起使用，通过 ExecutorService 的 submit 方法执行 Callable，并返回 Future。

```
ExecutorService executor = Executors.newCachedThreadPool();
```

```
        Future<String> future = executor.submit(() -> { //Lambda 是一个 callable,
提交后便立即执行, //这里返回的是 FutureTask 实例
            System.out.println("running task");
            Thread.sleep(10000);
            return "return task";
        });

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }

        System.out.println("do something else");  //前面的 Callable 在其他线程中
//运行着, 可以做一些其他的事情

        try {
            System.out.println(future.get());  //等待 Future 的执行结果, 执行完毕
//之后打印出来
        } catch (InterruptedException e) {
        } catch (ExecutionException e) {

        } finally {
            executor.shutdown();
        }
```

比起 future.get()，其实更推荐使用 get (long timeout, TimeUnit unit) 方法，它设置了超时时间，可以防止程序无限制地等待 Future 的结果。

2. CompletableFuture 介绍

Future 模式的缺点如下：

- ◎ Future 虽然可以实现获取异步执行结果的需求，但是它没有提供通知的机制，我们无法得知 Future 什么时候完成。
- ◎ 要么使用阻塞，在 future.get()的地方等待 Future 返回的结果，这时又变成同步操作。要么使用 isDone()轮询地判断 Future 是否完成，这样会耗费 CPU 资源。

2. CompletableFuture 介绍

Netty、Guava 分别扩展了 Java 的 Future 接口，以方便异步编程。

Java 8 新增的 CompletableFuture 类正是吸收了所有 Google Guava 中 ListenableFuture 和 SettableFuture 的特征，还提供了其他强大的功能，让 Java 拥有了完整的非阻塞编程模型：Future、Promise 和 Callback（在 Java8 之前，只有无 Callback 的 Future）。

CompletableFuture 既能够将回调放到与任务不同的线程中执行，又能将回调作为继续执行的同步函数，在与任务相同的线程中执行。它避免了传统回调的最大问题，那就是能够将控制流分离到不同的事件处理器中。

CompletableFuture 弥补了 Future 模式的缺点。在异步的任务完成后，需要用其结果继续操作时，无须等待，可以直接通过 thenAccept、thenApply、thenCompose 等方式将前面异步处理的结果交给另外一个异步事件处理线程来处理。

3. CompletableFuture 特性

（1）CompletableFuture 的静态工厂方法如表 16-3 所示。

表 16-3

方法名	描　述
runAsync(Runnable runnable)	使用 ForkJoinPool.commonPool()作为它的线程池执行异步代码
runAsync(Runnable runnable, Executor executor)	使用指定的 thread pool 执行异步代码
supplyAsync(Supplier<U> supplier)	使用 ForkJoinPool.commonPool()作为它的线程池执行异步代码，异步操作有返回值
supplyAsync(Supplier<U> supplier, Executor executor)	使用指定的 thread pool 执行异步代码，异步操作有返回值

runAsync 和 supplyAsync 方法的区别是，runAsync 返回的 CompletableFuture 是没有返回值的。

```
CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
    System.out.println("Hello");
});

try {
    future.get();
} catch (InterruptedException e) {
    e.printStackTrace();
} catch (ExecutionException e) {
    e.printStackTrace();
```

```
        }

        System.out.println("CompletableFuture");
```

而 supplyAsync 返回的 CompletableFuture 是有返回值的，下面的代码打印了 Future 的返回值。

```
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello");

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }

        System.out.println("CompletableFuture");
```

（2）Completable 如表 16-4 所示。

表 16-4

方法名	描　述
complete(T t)	完成异步执行，并返回 Future 的结果
completeExceptionally(Throwable ex)	异步执行不正常的结束

future.get()在等待执行结果时，程序会一直 block，如果此时调用 complete(T t)，则会立即执行。

```
        CompletableFuture<String> future  = CompletableFuture.supplyAsync(() ->
"Hello");

        future.complete("World");

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
```

```
            e.printStackTrace();
        }
```

执行结果：

```
World
```

可以看到 future 调用 complete(T t)会立即执行。但是 complete(T t)只能调用一次，后续的重复调用会失效。

如果 future 已经执行完毕能够返回结果，此时再调用 complete(T t)则会无效。

```
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello");

        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        future.complete("World");

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
Hello
```

如果使用 completeExceptionally(Throwable ex)则抛出一个异常，而不是一个成功的结果。

```
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello");

        future.completeExceptionally(new Exception());
```

```
        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
java.util.concurrent.ExecutionException: java.lang.Exception
...
```

（3）转换。

我们可以通过CompletableFuture来异步获取一组数据，并对数据进行一些转换，类似RxJava、Scala的map、flatMap操作。

map转换如表16-5所示。

表16-5

方法名	描　述
thenApply(Function<? super T,? extends U> fn)	接受一个Function<? super T,? extends U>参数用来转换CompletableFuture
thenApplyAsync(Function<? super T,? extends U> fn)	接受一个Function<? super T,? extends U>参数用来转换CompletableFuture，使用ForkJoinPool
thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)	接受一个Function<? super T,? extends U>参数用来转换CompletableFuture，使用指定的线程池

thenApply的功能相当于将CompletableFuture<t>转换成Completablefuture<U>。

```
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello");

        future = future.thenApply(new Function<String, String>() {

            @Override
            public String apply(String s) {

                return s + " World";
```

```
        }
    }).thenApply(new Function<String, String>() {
        @Override
        public String apply(String s) {

            return s.toUpperCase();
        }
    });

    try {
        System.out.println(future.get());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
```

再用 Lambda 表达式简化一下：

```
    CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello")
            .thenApply(s -> s + " World").thenApply(String::toUpperCase);

    try {
        System.out.println(future.get());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
```

执行结果：

```
HELLO WORLD
```

下面的例子，展示了数据流的类型经历了如下转换：String → Integer →Double。

```
    CompletableFuture<Double> future = CompletableFuture.supplyAsync(() ->
"10")
            .thenApply(Integer::parseInt)
            .thenApply(i->i*10.0);
```

```
        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
100.0
```

flatMap 转换如表 16-6 所示。

表 16-6

方法名	描　述
thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)	在异步操作完成的时候对异步操作的结果进行一些操作，并且仍然返回 CompletableFuture 类型
thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn)	在异步操作完成的时候对异步操作的结果进行一些操作，并且仍然返回 CompletableFuture 类型。使用 ForkJoinPool
thenComposeAsync(Function<? super T, ? extends CompletionStage<U>> fn,Executor executor)	在异步操作完成的时候对异步操作的结果进行一些操作，并且仍然返回 CompletableFuture 类型。使用指定的线程池

thenCompose 可用于组合多个 CompletableFuture，将前一个结果作为下一个计算的参数，它们之间存在着先后顺序。

```
        CompletableFuture<String> future = CompletableFuture.supplyAsync(() ->
"Hello")
                .thenCompose(s -> CompletableFuture.supplyAsync(() -> s + "
World"));

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
Hello World
```

下面的例子展示了多次调用 thenCompose()：

```
    CompletableFuture<Double> future = CompletableFuture.supplyAsync(() ->
"100")
            .thenCompose(s -> CompletableFuture.supplyAsync(() -> s +
"100"))
            .thenCompose(s -> CompletableFuture.supplyAsync(() ->
Double.parseDouble(s)));

    try {
        System.out.println(future.get());
    } catch (InterruptedException e) {
        e.printStackTrace();
    } catch (ExecutionException e) {
        e.printStackTrace();
    }
```

执行结果：

```
100100.0
```

（4）组合，如表 16-7 所示。

表 16-7

方法名	描　述
thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)	当两个 CompletableFuture 都正常完成后，执行提供的 fn，用它来组合另外一个 CompletableFuture 的结果
thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)	当两个 CompletableFuture 都正常完成后，执行提供的 fn，用它来组合另外一个 CompletableFuture 的结果。使用 ForkJoinPool
thenCombineAsync(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn, Executor executor)	当两个 CompletableFuture 都正常完成后，执行提供的 fn，用它来组合另外一个 CompletableFuture 的结果。使用指定的线程池

现在有 CompletableFuture<T>、completablefuture<U>和一个函数(T,U)->v，thencompose 就是将 completablefuture<T>和 completablefuture<U>变为 completablefuture<V>。

```
        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() ->
"100");
        CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(()
-> 100);

        CompletableFuture<Double> future = future1.thenCombine(future2, (s, i)
-> Double.parseDouble(s + i));

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
100100.0
```

使用 thenCombine()之后，future1、future2 之间是并行执行的，最后再将结果汇总。这一点与 thenCompose()不同。

thenAcceptBoth 与 thenCombine 类似，但是返回的是 CompletableFuture 类型，如表 16-8 所示。

表 16-8

方法名	描　述
thenAcceptBoth(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)	当两个 CompletableFuture 都正常完成后，执行提供的 action，用它来组合另外一个 CompletableFuture 的结果
thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action)	当两个 CompletableFuture 都正常完成后，执行提供的 action，用它来组合另外一个 CompletableFuture 的结果。使用 ForkJoinPool
thenAcceptBothAsync(CompletionStage<? extends U> other, BiConsumer<? super T,? super U> action, Executor executor)	当两个 CompletableFuture 都正常完成后，执行提供的 action，用它来组合另外一个 CompletableFuture 的结果。使用指定的线程池

```
        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() ->
"100");
```

```
        CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(()
-> 100);

        CompletableFuture<Void> future = future1.thenAcceptBoth(future2, (s, i)
-> System.out.println(Double.parseDouble(s + i)));

        try {
            future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
100100.0
```

（5）计算结果完成时的处理。

当 CompletableFuture 完成计算结果后，我们可能需要对结果进行一些处理。

执行特定的 Action，如表 16-9 所示。

表 16-9

方法名	描　述
whenComplete(BiConsumer<? super T,? super Throwable> action)	当 CompletableFuture 完成计算结果时对结果进行处理，或者当 CompletableFuture 产生异常时对异常进行处理
whenCompleteAsync(BiConsumer<? super T,? super Throwable> action)	当 CompletableFuture 完成计算结果时对结果进行处理，或者当 CompletableFuture 产生异常时对异常进行处理。使用 ForkJoinPool
whenCompleteAsync(BiConsumer<? super T,? super Throwable> action, Executor executor)	当 CompletableFuture 完成计算结果时对结果进行处理，或者当 CompletableFuture 产生异常时对异常进行处理。使用指定的线程池

```
        CompletableFuture.supplyAsync(() -> "Hello")
                .thenApply(s->s+" World")
                .thenApply(s->s+ "\nThis is CompletableFuture demo")
                .thenApply(String::toLowerCase)
                .whenComplete((result, throwable) ->
System.out.println(result));
```

执行结果：

```
hello world
this is completablefuture demo
```

执行完 Action 可以做转换，如表 16-10 所示。

表 16-10

方法名	描　述
handle(BiFunction<? super T, Throwable, ? extends U> fn)	当 CompletableFuture 完成计算结果或者抛出异常时，执行提供的 fn
handleAsync(BiFunction<? super T, Throwable, ? extends U> fn)	当 CompletableFuture 完成计算结果或者抛出异常时，执行提供的 fn，使用 ForkJoinPool
handleAsync(BiFunction<? super T, Throwable, ? extends U> fn, Executor executor)	当 CompletableFuture 完成计算结果或者抛出异常时，执行提供的 fn，使用指定的线程池

```
        CompletableFuture<Double> future = CompletableFuture.supplyAsync(() ->
"100")
                .thenApply(s->s+"100")
                .handle((s, t) -> s != null ? Double.parseDouble(s) : 0);

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：

```
100100.0
```

在这里，handle()的参数是 BiFunction，apply()方法返回 R，相当于转换的操作。

```
@FunctionalInterface
public interface BiFunction<T, U, R> {

    /**
     * Applies this function to the given arguments.
```

```
     *
     * @param t the first function argument
     * @param u the second function argument
     * @return the function result
     */
    R apply(T t, U u);

    /**
     * Returns a composed function that first applies this function to
     * its input, and then applies the {@code after} function to the result.
     * If evaluation of either function throws an exception, it is relayed to
     * the caller of the composed function.
     *
     * @param <V> the type of output of the {@code after} function, and of the
     *           composed function
     * @param after the function to apply after this function is applied
     * @return a composed function that first applies this function and then
     * applies the {@code after} function
     * @throws NullPointerException if after is null
     */
    default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after)
{
        Objects.requireNonNull(after);
        return (T t, U u) -> after.apply(apply(t, u));
    }
}
```

而 whenComplete()的参数是 BiConsumer，accept()方法返回 void。

```
@FunctionalInterface
public interface BiConsumer<T, U> {

    /**
     * Performs this operation on the given arguments.
     *
     * @param t the first input argument
     * @param u the second input argument
     */
    void accept(T t, U u);

    /**
```

```
     * Returns a composed {@code BiConsumer} that performs, in sequence, this
     * operation followed by the {@code after} operation. If performing either
     * operation throws an exception, it is relayed to the caller of the
     * composed operation.  If performing this operation throws an exception,
     * the {@code after} operation will not be performed.
     *
     * @param after the operation to perform after this operation
     * @return a composed {@code BiConsumer} that performs in sequence this
     * operation followed by the {@code after} operation
     * @throws NullPointerException if {@code after} is null
     */
    default BiConsumer<T, U> andThen(BiConsumer<? super T, ? super U> after) {
        Objects.requireNonNull(after);

        return (l, r) -> {
            accept(l, r);
            after.accept(l, r);
        };
    }
}
```

所以，handle()相当于 whenComplete()+转换。

纯消费（执行 Action）如表 16-11 所示。

表 16-11

方法名	描　述
thenAccept(Consumer<? super T> action)	当 CompletableFuture 完成计算结果后，只对结果执行 Action，而不返回新的计算值
thenAcceptAsync(Consumer<? super T> action)	当 CompletableFuture 完成计算结果后，只对结果执行 Action，而不返回新的计算值，使用 ForkJoinPool
thenAcceptAsync(Consumer<? super T> action, Executor executor)	当 CompletableFuture 完成计算结果后，只对结果执行 Action，而不返回新的计算值

thenAccept()是只会对计算结果进行消费而不会返回任何结果的方法。

```
CompletableFuture.supplyAsync(() -> "Hello")
        .thenApply(s->s+" World")
        .thenApply(s->s+ "\nThis is CompletableFuture demo")
```

```
                .thenApply(String::toLowerCase)
                .thenAccept(System.out::print);
```

执行结果：

```
hello world
this is completablefuture demo
```

（6）Either。

Either 表示的是两个 CompletableFuture，当其中任意一个 CompletableFuture 计算完成的时候就会执行，如表 6-12 所示。

表 16-12

方法名	描　述
acceptEither(CompletionStage<? extends T> other, Consumer<? super T> action)	当任意一个 CompletableFuture 完成时，action 这个消费者就会被执行
acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action)	当任意一个 CompletableFuture 完成时，action 这个消费者就会被执行。使用 ForkJoinPool
acceptEitherAsync(CompletionStage<? extends T> other, Consumer<? super T> action, Executor executor)	当任意一个 CompletableFuture 完成时，action 这个消费者就会被执行。使用指定的线程池

```
        Random random = new Random();

        CompletableFuture<String> future1 =
CompletableFuture.supplyAsync(()->{

            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            return "from future1";
        });

        CompletableFuture<String> future2 =
CompletableFuture.supplyAsync(()->{
```

```
            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            return "from future2";
        });

        CompletableFuture<Void> future =
future1.acceptEither(future2,str->System.out.println("The future is "+str));

        try {
            future.get();
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果：The future is from future1 或者 The future is from future2。因为 future1 和 future2 的执行顺序是随机的。

applyToEither 与 acceptEither 类似，如表 16-13 所示。

表 16-13

方法名	描　述
applyToEither(CompletionStage<? extends T> other, Function<? super T,U> fn)	当任意一个 CompletableFuture 完成的时候，fn 会被执行，它的返回值会当作新的 CompletableFuture<U>的计算结果
applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn)	当任意一个 CompletableFuture 完成的时候，fn 会被执行，它的返回值会当作新的 CompletableFuture<U>的计算结果。使用 ForkJoinPool
applyToEitherAsync(CompletionStage<? extends T> other, Function<? super T,U> fn, Executor executor)	当任意一个 CompletableFuture 完成的时候，fn 会被执行，它的返回值会当作新的 CompletableFuture<U>的计算结果。使用指定的线程池

```
        Random random = new Random();

        CompletableFuture<String> future1 =
CompletableFuture.supplyAsync(()->{

            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            return "from future1";
        });

        CompletableFuture<String> future2 =
CompletableFuture.supplyAsync(()->{

            try {
                Thread.sleep(random.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            return "from future2";
        });

        CompletableFuture<String> future =
future1.applyToEither(future2,str->"The future is "+str);

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

执行结果也与上面的程序类似。

（7）其他方法。

allOf、anyOf 是 CompletableFuture 的静态方法。

allOf 如表 16-14 所示。

表 16-14

方法名	描　述
allOf(CompletableFuture<?>... cfs)	在所有 Future 对象完成后结束，并返回一个 future

allOf()方法所返回的 CompletableFuture，并不能组合前面多个 CompletableFuture 的计算结果，于是我们借助 Java 8 的 Stream 来组合多个 future 的结果。

```
        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() ->
"tony");

        CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() ->
"cafei");

        CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() ->
"aaron");

        CompletableFuture.allOf(future1, future2, future3)
                .thenApply(v ->
                Stream.of(future1, future2, future3)
                        .map(CompletableFuture::join)
                        .collect(Collectors.joining(" ")))
                .thenAccept(System.out::print);
```

执行结果：

```
tony cafei aaron
```

anyOf 如表 16-15 所示

表 16-15

方法名	描　述
anyOf(CompletableFuture<?>... cfs)	在任何一个 Future 对象结束后结束，并返回一个 future

```
        Random rand = new Random();
        CompletableFuture<String> future1 = CompletableFuture.supplyAsync(() ->
{
```

```
            try {
                Thread.sleep(rand.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "from future1";
        });
        CompletableFuture<String> future2 = CompletableFuture.supplyAsync(() ->
{
            try {
                Thread.sleep(rand.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "from future2";
        });
        CompletableFuture<String> future3 = CompletableFuture.supplyAsync(() ->
{
            try {
                Thread.sleep(rand.nextInt(1000));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            return "from future3";
        });

        CompletableFuture<Object> future =
CompletableFuture.anyOf(future1,future2,future3);

        try {
            System.out.println(future.get());
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (ExecutionException e) {
            e.printStackTrace();
        }
```

使用 anyOf()时，只要某一个 future 完成，就结束了。所以执行结果可能是“from future1”、“from future2”、“from future3”中的任意一个。

anyOf 和 acceptEither、applyToEither 的区别在于，后两者只能使用在两个 future 中，而 anyOf 可以使用在多个 future 中。

（8）CompletableFuture 异常处理。

CompletableFuture 在运行时如果遇到异常，则可以使用 get()并抛出异常进行处理，但这并不是一个最好的方法。CompletableFuture 本身也提供了几种方式来处理异常。

Exceptionally，如表 16-16 所示。

表 16-16

方法名	描　述
exceptionally(Function<Throwable,? extends T> fn)	只有当 CompletableFuture 抛出异常的时候，才会触发这个 exceptionally 的计算，调用 Function 计算值

```
        CompletableFuture.supplyAsync(() -> "hello world")
                .thenApply(s -> {
                    s = null;
                    int length = s.length();
                    return length;
                }).thenAccept(i -> System.out.println(i))
                .exceptionally(t -> {
                    System.out.println("Unexpected error:" + t);
                    return null;
                });
```

执行结果：

```
Unexpected error:java.util.concurrent.CompletionException:
java.lang.NullPointerException
```

对上面的代码稍微做一下修改，修复了空指针的异常。

```
        CompletableFuture.supplyAsync(() -> "hello world")
                .thenApply(s -> {
//                      s = null;
                    int length = s.length();
                    return length;
                }).thenAccept(i -> System.out.println(i))
                .exceptionally(t -> {
```

```
                System.out.println("Unexpected error:" + t);
                return null;
            });
```

执行结果：

```
11
```

whenComplete 在前面已经介绍过了，在这里跟 exceptionally 的作用差不多，可以捕获任意阶段的异常。如果没有异常，就执行 action。

```
    CompletableFuture.supplyAsync(() -> "hello world")
            .thenApply(s -> {
                s = null;
                int length = s.length();
                return length;
            }).thenAccept(i -> System.out.println(i))
            .whenComplete((result, throwable) -> {

                if (throwable != null) {
                   System.out.println("Unexpected error:"+throwable);
                } else {
                    System.out.println(result);
                }

            });
```

执行结果：

```
Unexpected error:java.util.concurrent.CompletionException:
java.lang.NullPointerException
```

与 whenComplete 相似的方法是 handle，handle 的用法在前面也已经介绍过。

4. CompletableFuture VS Java8 Stream VS RxJava 1 & RxJava 2

CompletableFuture 有很多特性与 RxJava 很像，所以将 CompletableFuture、Java 8 Stream 和 RxJava 做一个相互的比较，如表 16-17 所示。

表 16-17

	composable	lazy	resuable	async	cached	push	Back Pressure
CompletableFuture	支持	不支持	支持	支持	支持	支持	不支持
Stream	支持	支持	不支持	不支持	不支持	不支持	不支持
Observable(RxJava1)	支持	支持	支持	支持	支持	支持	支持
Observable(RxJava2)	支持	支持	支持	支持	支持	支持	不支持
Flowable(RxJava2)	支持	支持	支持	支持	支持	支持	支持

Java 8 提供了一种函数风格的异步和事件驱动编程模型 CompletableFuture，它不会造成堵塞。CompletableFuture 背后依靠的是 fork/join 框架来启动新的线程实现异步与并发。当然，我们也能通过指定线程池来做这些事情。

特别是对微服务架构而言，CompletableFuture 会有很大的作为。举一个具体的场景，电商的商品页面可能会涉及商品详情服务、商品评论服务、相关商品推荐服务等。获取商品的信息时（/productdetails?productid=xxx），需要调用多个服务来处理这一个请求并返回结果。这里可能会涉及并发编程，我们完全可以使用 Java 8 的 CompletableFuture 或者 RxJava 来实现。

16.7 小结

Java 8 算是自 Java 5 以来最大的一次更新，引入了函数式编程的思想。本章主要介绍了这些函数式编程的特性，也花了大力气着重介绍了新的异步编程方式 CompletableFuture。

希望读完本章之后，大家能够在新的项目中使用 Java 8 的新特性，无论是在服务端还是客户端的项目中。

第17章
Kotlin和RxJava

17.1 Kotlin 简介

2017 年 5 月 18 日谷歌在 Google I/O 2017 大会上宣布 Kotlin 成 Android 开发一级语言，并会在 Android Studio 3 中提供相关支持。谷歌称其“简洁、表现力强，具有类型安全和空值安全的特点，也可以与 Java 进行完整的互操作”。Kotlin 一夜之间成为 Android 开发中的热门话题。

Kotlin 是一个基于 JVM 的编程语言，它由 JetBrains 开发。Kotlin 既可以编译成 Java 字节码，也可以编译成 JavaScript，方便在没有 JVM 的设备上运行。

JetBrains，作为目前广受欢迎的 Java IDE IntelliJ IDEA 的开发商，在 Apache 许可下已经开源 Kotlin 编程语言。JetBrains 作为最智能的 Java IDE 的开发商，对 Java 的了解是毋庸置疑的，在使用 Java 的过程中，JetBrains 的工程师们发现了大量的问题，为了更高效地开发以及解决 Java 中的一些问题，JetBrains 开发了致力于替代 Java 的 Kotlin。

Kotlin 还可以基于 Native 编程，这不是类似 JNI 的概念，不仅仅是要与底层代码比如 C、C++ 交互，而且还要绕过 JVM 直接编译成机器码供系统运行。在编写本书时 Kotlin Native 还处于开发阶段，离商用还有一段时间。

1. 为何需要 Kotlin?

（1）空类型安全。

在 Java 中针对空指针，需要写很多防御性的代码，Kotlin 基于这个问题，提出了一个空安

全的概念，即每个属性默认不可为 null。

```
var a: String = "test kotlin"
a = null //编译错误
```

如果要允许为空，则需要手动声明一个变量为可空字符串类型，写为 String?

```
var a: String? = "test kotlin"
a = null //编译成功
```

（2）Lambda 表达式。

在 Kotlin 中可以直接使用 Lambda 表达式，从而简化代码量，提高代码的可读性。

（3）扩展函数。

Kotlin 的扩展函数功能可以使我们为现有的类添加新的函数，而不用修改原来的类，例如：

```
fun View.hideKeyboard() {
    clearFocus()
    (context.getSystemService(Context.INPUT_METHOD_SERVICE) as
InputMethodManager).hideSoftInputFromWindow(windowToken, 0)
}

fun View.showKeyboard() {
    requestFocus()
    (context.getSystemService(Context.INPUT_METHOD_SERVICE) as
InputMethodManager).showSoftInput(this, InputMethodManager.SHOW_IMPLICIT)
}
```

声明一个扩展函数很简单，只需在函数名前添加指定的类名即可。在调用时，该函数会以导入的方式添加到这个类中。

（4）类型推导。

Kotlin 会推断我们想要的类型（我们会觉得提高了代码的阅读性）：

```
val a = "kotlin"                          // type inferred to String
val b = 5                                 // type inferred to Int

val c: Double = 1.0                       // type declared explicitly
val d: List<String> = ArrayList()         // type declared explicitly
```

（5）胜任 Java 能做的所有事。

Kotlin 使用既可以进行 Android 开发，也可以做服务端开发，Kotlin 100%兼容所有 JVM 框架。使用 Kotlin 编写的代码，也可转换为 JavaScript 在 Node.js 或浏览器中运行。

（6）简洁优雅。

不用写分号，天然支持 Lambda 表达式，它的语法本身就比 Java 简洁许多。

2. Kotlin 概述

（1）使用 Kotlin 进行 Android 开发。

Kotlin 非常适合开发 Android 应用程序，它可将现代语言的所有优势带入 Android 平台而不会引入任何新的限制。

- ◎ 兼容性：Kotlin 与 JDK 6 完全兼容（Kotlin 可以让你选择生成 Java 6 或者 Java 8 兼容的字节码。可以为较高版本的平台生成更优化的字节码），保障了 Kotlin 应用程序可以在较旧的 Android 设备上运行而无任何问题。Kotlin 工具在 Android Studio 中完全支持，并且兼容 Android 构建系统。
- ◎ 性能：由于有非常相似的字节码结构，因此 Kotlin 应用程序的运行速度与 Java 类似。随着 Kotlin 对内联函数的支持，使用 Lambda 表达式的代码通常比用 Java 写的代码运行得更快。
- ◎ 互操作性：Kotlin 可与 Java 进行 100% 的互操作，允许在 Kotlin 应用程序中使用所有现有的 Android 库，包括注解处理数据绑定和 Dagger。
- ◎ 占用：Kotlin 具有非常紧凑的运行时库，可以通过使用 ProGuard 进一步减少。在实际应用程序中，Kotlin 运行时只增加几百个方法，.apk 文件不到 100KB。
- ◎ 编译时长：Kotlin 支持高效的增量编译，所以对于清理构建会有额外的开销，增量构建通常与 Java 一样快或者更快。
- ◎ 学习曲线：对于 Java 开发人员，Kotlin 入门很容易。包含在 Kotlin 插件中的自动 Java 到 Kotlin 的转换器有助于迈出第一步。

其实，在本书的很多章节中曾多次提到过 Kotlin，笔者喜欢用 Kotlin 来写一些工具类，当然也会用它来做一些具体的业务。

（2）使用 Kotlin 进行服务器端开发。

Kotlin 非常适合开发服务器端的应用程序，允许编写简明且表现力强的代码，同时保持与现有基于 Java 的技术栈的完全兼容性以及平滑的学习曲线。

- 表现力：Kotlin 的革新式语言功能，例如支持类型安全的构建器和委托属性，有助于构建强大且易于使用的抽象。
- 可伸缩性：Kotlin 对协程的支持有助于构建服务器端应用程序，伸缩到适度的硬件要求以应对大量的客户端。
- 互操作性：Kotlin 与所有基于 Java 的框架完全兼容，可以让你保持熟悉的技术栈，同时获得更现代化语言的优势。
- 迁移：Kotlin 支持大型代码库从 Java 到 Kotlin 的逐步迁移。你可以开始用 Kotlin 编写新代码，同时系统中较旧部分继续用 Java。
- 工具：除了很棒的 IDE 支持外，Kotlin 还为 IntelliJ IDEA Ultimate 的插件提供了框架特定的工具（例如 Spring）。
- 学习曲线：对于 Java 开发人员，Kotlin 入门很容易。包含在 Kotlin 插件中的自动 Java 到 Kotlin 的转换器有助于迈出第一步。

（3）Kotlin JavaScript 概述。

Kotlin 提供了 JavaScript 作为目标平台的能力。它通过将 Kotlin 转换为 JavaScript 来实现。目前的实现目标是 ECMAScript 5.1，但也有最终目标为 ECMAScript 2015 的计划。当你选择 JavaScript 目标时，作为项目一部分的任何 Kotlin 代码以及 Kotlin 附带的标准库都会转换为 JavaScript。但是，这不包括使用的 JDK 和任何 JVM 或 Java 框架或库。任何不是 Kotlin 的文件会在编译期间被忽略掉。

Kotlin 编译器努力遵循以下目标：

- 提供最佳大小的输出；
- 提供可读的 JavaScript 输出；
- 提供与现有模块系统的互操作性；
- 在标准库中提供相同的功能，无论是 JavaScript，还是 JVM 目标（尽最大可能程度）。

17.2　使用 Kotlin 来封装图像框架

1. Kotlin + RxJava 2 封装滤镜的操作

这里我们首先要介绍一个图像处理框架 cv4j。

cv4j 是一个高质量的实时图像处理和机器学习库，纯 Java 实现。该框架可以运行在 Java 桌面和 Android 平台。目前 cv4j 由贾志刚老师跟我一起维护，cv4j 的 GitHub 地址：https://github.com/imageprocessor/cv4j

cv4j 框架的一个重要特点是已经包含了几十款滤镜的功能。在 Android 开发中，可以直接拿来使用。

cv4j 最早使用滤镜的方式如下：

```
CV4JImage cv4jImage = new CV4JImage(bitmap);
CommonFilter filter = new NatureFilter();
Bitmap newBitMap =
filter.filter(cv4jImage.getProcessor()).getImage().toBitmap();
image.setImageBitmap(newBitMap);
```

后来增加了 RxJava 封装的版本之后，使用起来更加简单。

```
RxImageData.bitmap(bitmap).addFilter(new NatureFilter()).into(image);
```

来看一下使用滤镜的效果图，如图 17-1 所示。

图 17-1

下面还展示了多种滤镜的效果，只有第一张图是原图，其他都是使用滤镜的效果图，如图 17-2 和图 17-3 所示。

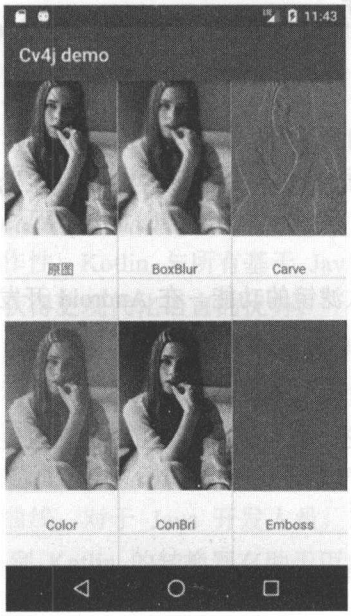


图 17-2

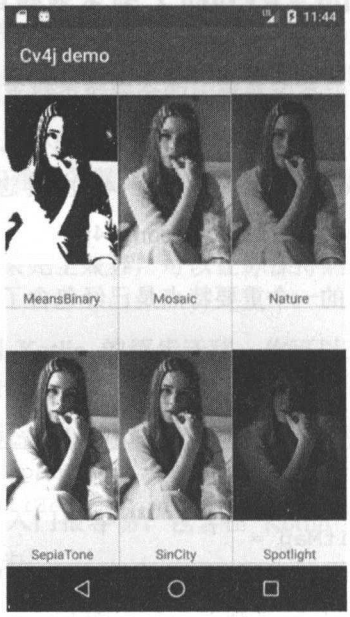


图 17-3

RxImageData 对滤镜功能做了封装，它最早是使用 RxJava，后来笔者用 Kotlin 重构了一下。Kotlin 的 Lambda 表达式，比起 Java 的 Lambda 表达式更加简洁。下面是 RxImageData 的源码，RxImageData 支持链式调用，支持添加多个滤镜，使用起来非常简单。

```
/*
 * Copyright (c) 2017-present, CV4J Contributors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
```

```
 * limitations under the License.
 */
package com.cv4j.rxjava

import android.app.Dialog
import android.graphics.Bitmap
import android.util.Log
import android.widget.ImageView
import com.cv4j.core.datamodel.CV4JImage
import com.cv4j.core.datamodel.ImageProcessor
import com.cv4j.core.filters.CommonFilter
import com.saffframework.utils.RxJavaUtils
import io.reactivex.Flowable
import io.reactivex.FlowableTransformer
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.schedulers.Schedulers
import org.jetbrains.annotations.NotNull
import java.util.*

class RxImageData private constructor(internal var image: CV4JImage?) {
    internal var flowable: Flowable<WrappedCV4JImage>
    internal var memCache: MemCache
    internal var useCache = true

    internal var imageView: ImageView? = null
    internal var filters: MutableList<CommonFilter>
    internal var wrappedCV4JImage: WrappedCV4JImage
    internal var mDialog: Dialog? = null

    private constructor(bytes: ByteArray?) : this(CV4JImage(bytes)) {}

    private constructor(bitmap: Bitmap?) : this(CV4JImage(bitmap)) {}

    init {
        filters = ArrayList<CommonFilter>()
        memCache = MemCache.instance

        wrappedCV4JImage = WrappedCV4JImage(image, filters)
        flowable = Flowable.just(wrappedCV4JImage)
    }

    fun dialog(dialog: Dialog?): RxImageData {
```

```
        if (dialog == null) {

            return this
        }

        this.mDialog = dialog
        this.mDialog?.show()
        return this
    }

    /**
     * 使用滤镜，支持链式调用多个滤镜
     * @param filter
     * *
     * @return
     */
    fun addFilter(filter: CommonFilter?): RxImageData {

        if (filter == null) {

            Log.e("RxImageData", "filter is null")
            return this
        }

        filters.add(filter)
        return this
    }

    /**
     * 判断是否使用缓存，默认情况下是使用缓存
     * 该方法需要在 into() 方法之前使用
     * @param useCache
     * *
     * @return
     */
    fun isUseCache(useCache: Boolean): RxImageData {

        this.useCache = useCache
        return this
    }
```

```
/**
 * RxImageData.bitmap(bitmap).addFilter(new ColorFilter()).into(view);
 * @param imageview
 */
fun into(imageview: ImageView?) {

    if (imageview == null) {

        Log.e("RxImageData", "imageview is null")
        return
    }

    this.imageView = imageview
    render()
}

/**
 * 渲染 imageview
 */
private fun render() {

    if (imageView == null) {
        return
    }

    if (filters.size == 0) {
        this.flowable.compose(RxJavaUtils.flowableToMain())
                .doFinally({
                    dismissDiaog()
                })
                .subscribe({ (cV4JImage) ->
                    imageView?.setImageBitmap(cV4JImage?.toBitmap())
                }, { t ->
                    t.printStackTrace()
                })
    } else if (filters.size == 1) {
        this.flowable
                .map({ (image1, filters1) ->
                    if (useCache) {

                        val sb = StringBuilder()
                        if (imageView?.context != null) {
```

```
                        sb.append(imageView!!.context.javaClass.simple
Name)
                    }

                    sb.append(filters1[0].javaClass.simpleName).append
(imageView?.id)

                    // 目前 key 采用 activity name + filter name + imageView
// id
                    val key = Utils.md5(sb.toString())

                    if (memCache.get(key) == null) {

                        val imageProcessor =
filters1[0].filter(image?.processor)
                        memCache.put(key,
imageProcessor.image.toBitmap())

                        imageProcessor
                    } else {

                        image!!.processor.image.setBitmap(memCache.get
(key))

                        image!!.processor
                    }
                } else {

                    filters1[0].filter(image?.processor)
                }
            })
            .compose(RxJavaUtils.flowableToMain())
            .doFinally({
                dismissDiaog()
            })
            .subscribe({ processor ->
                imageView?.setImageBitmap(processor.image.toBitmap())
            }, { t ->
                t.printStackTrace()
            })

    } else {
```

```
            this.flowable
                    .map({ (image1, filters1) -> filters1 })
                    .map { filter(image!!.processor) }
                    .compose(RxJavaUtils.flowableToMain())
                    .doFinally({
                        dismissDiaog()
                    })
                    .subscribe({ processor ->
                        imageView?.setImageBitmap(processor.image.toBitmap())
                    }, { t ->
                        t.printStackTrace()
                    })
        }
    }


    private fun filter(imageData: ImageProcessor): ImageProcessor = if
(filters.size > 0) filter(imageData, filters.size) else imageData

    private fun filter(imageData: ImageProcessor, size: Int): ImageProcessor {
        var imageData = imageData

        if (size == 1) {
            val filter = filters[0]
            return filter.filter(imageData)
        }

        val filter = filters[size - 1]
        imageData = filter.filter(imageData)

        return filter(imageData, size - 1)
    }

    private fun dismissDiaog() {

        if (mDialog != null) {
            mDialog!!.dismiss()
            mDialog = null
        }
    }

    /**
```

```
     * 释放资源
     */
    fun recycle() {

        image?.recycle()
    }

    companion object {

        @JvmStatic
        fun bytes(@NotNull bytes: ByteArray?): RxImageData {

            return RxImageData(bytes)
        }

        @JvmStatic
        fun bitmap(@NotNull bitmap: Bitmap?): RxImageData {

            return RxImageData(bitmap)
        }

        @JvmStatic
        fun image(@NotNull image: CV4JImage?): RxImageData {

            return RxImageData(image)
        }
    }
}
```

2. DSL 风格的编程

DSL 是领域专用语言（Domain Specific Language，DSL），其基本思想是“求专不求全”，不像通用目的语言那样目标范围涵盖一切软件问题，而是专门针对某一特定问题的计算机语言。前几年随着被誉为“Web 开发领域专用语言”的 Ruby on Rails 而迅速走红，DSL 又一次成为人们讨论的热点话题。很多人都认为，DSL 将会是软件业的“next big thing”。

DSL 编程是一种声明式编程，直观、容易理解和使用。相信做过 Android 开发的读者在使用 Android Studio 时，一定都亲身感受到 Gradle 的巨大威力。和 Maven 相比，Gradle 的 DSL 是如此简洁。同样，Kotlin DSL 也会帮助我们减少代码量。

在 Kotlin 领域，比较著名的 DSL 框架是 Anko。

3. Anko

Anko 是一个 DSL，它是由 JetBrains 出品的，用 Kotlin 开发的安卓框架。它主要的目的是用来替代以前 XML 的方式来使用代码生成 UI 布局。

先来看一个直观的例子：

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_height="match_parent"
    android:layout_width="match_parent">

    <EditText
        android:id="@+id/todo_title"
        android:layout_width="match_parent"
        android:layout_heigh="wrap_content"
        android:hint="@string/title_hint" />

    <!-- Cannot directly add an inline click listener as onClick delegates
implementation to the activity -->
    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/add_todo" />

</LinearLayout>
```

使用 Anko 之后，可以用代码实现布局，并且 button 还绑定了点击事件。

```
verticalLayout {
    var title = editText {
        id = R.id.todo_title
        hintResource = R.string.title_hint
    }
    button {
        textResource = R.string.add_todo
        onClick { view -> {
                // do something here
                title.text = "Foo"
```

```
            }
        }
    }
}
```

可以看到 DSL 的一个主要优点在于，它只需用很少的时间即可理解和传达某个领域的详细信息。

4. 封装 cv4j 图像处理框架

现在使用 Kotlin 的项目，还多了一种方式来使用滤镜的功能。

```
        cv4j {
            bitmap = BitmapFactory.decodeResource(resources,
R.drawable.test_io)

            filter = NatureFilter()

            imageView = image
        }
```

比起之前通过代码来实现滤镜，这个 DSL 是不是更加直观呢？那它是如何实现的呢？其实是对 RxImageData 的进一步封装。

```
/*
 * Copyright (c) 2017-present, CV4J Contributors.
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */
package com.cv4j.rxjava
```

```
import android.app.Dialog
import android.graphics.Bitmap
import android.widget.ImageView
import com.cv4j.core.datamodel.CV4JImage
import com.cv4j.core.filters.CommonFilter

/**
 * only for Kotlin code,this class provides the DSL style for cv4j
 */
class Wrapper {

    var bitmap:Bitmap? = null

    var cv4jImage: CV4JImage? = null

    var bytes:ByteArray? = null

    var useCache:Boolean = true

    var imageView: ImageView? = null

    var filter: CommonFilter? = null

    var dialog: Dialog? = null
}

fun cv4j(init: Wrapper.() -> Unit) {
    val wrap = Wrapper()

    wrap.init()

    render(wrap)
}

private fun render(wrap: Wrapper) {

    if (wrap.bitmap!=null) {

        RxImageData.bitmap(wrap.bitmap).dialog(wrap.dialog).addFilter(wrap.f
ilter).isUseCache(wrap.useCache).into(wrap.imageView)

    } else if (wrap.cv4jImage!=null) {
```

```
        RxImageData.image(wrap.cv4jImage).dialog(wrap.dialog).addFilter(wrap
.filter).isUseCache(wrap.useCache).into(wrap.imageView)

    } else if (wrap.bytes!=null) {

        RxImageData.bytes(wrap.bytes).dialog(wrap.dialog).addFilter(wrap.fil
ter).isUseCache(wrap.useCache).into(wrap.imageView)

    }

}
```

程序的最终效果图如图 17-4 所示。

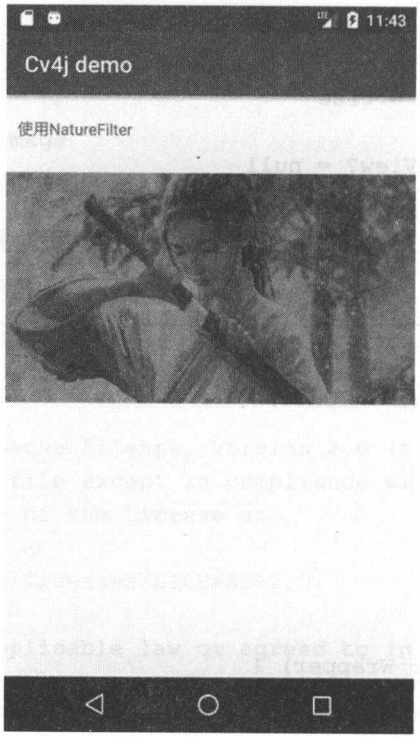


图 17-4

17.3　小结

Kotlin 相对 Java 有很多优势，比如代码更加简洁、安全，Kotlin 作为一门新的语言也有自己的特色，比如类型推断、多范式支持、可控性表达、扩展函数、DSL 支持等。

Kotlin 从一开始就支持 Lambda、高阶函数、Streams API 等函数式编程的特性。而且可以与 Java 无缝交互。未来很看好 Kotlin 的发展，以及它在 Android 开发和服务端领域开发的表现。

第 18 章
展望未来

18.1 期待已久的 Java 9

2017 年 9 月 21 日终于迎来了期待已久的 Java 9。

笔者整理出了一张思维导图，大致涵盖了 Java 9 的新特性，如图 18-1 所示。

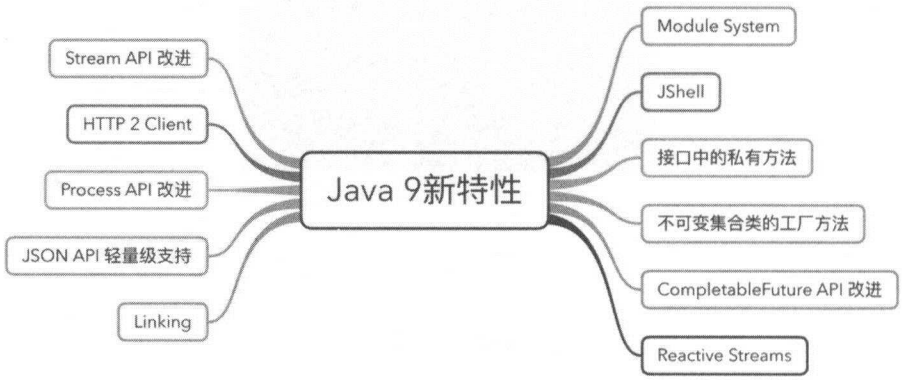


图 18-1

Java 9 最大的改变是引入了模块化。在 Java 9 之前，我们一般使用 Monolithic Jars 来开发基于 Java 语言的应用程序。这种体系架构有许多局限性和缺点，为了解决这些问题，Java 9 引入了 Module System。

我们在开发程序时，当代码库越来越大、越来越复杂时就会遇到两个基本问题：① 很难真正地对代码进行封装；② 系统并没有对不同部分（也就是 JAR 文件）之间的依赖关系有明确的概念，每一个公共类都可以被类路径下的任何其他的公共类所访问到，这样就会导致无意中使用了并不想被公开访问的 API。模块化的 JAR 文件都包含一个额外的模块描述器。在这个模块描述器中，对其他模块的依赖是通过“requires”来表示的。另外，“exports”语句控制着哪些包可以被其他模块访问到。所有不被导出的包默认都封装在模块的里面。我们既可以使用 JDK 的 Modules，也可以创建我们自己的 modules。

从函数响应式编程的角度看，最大的特色是增加了 Reactive Streams。响应式编程由于其便利性在应用程序开发中变得非常流行。RxJava2、Play、Akka 等框架都已经集成了 Reactive Streams 并且受益良多。JDK 9 也引入了一个新的 Reactive Streams API。JDK 9 的 Reactive Streams API 是一个发布订阅型框架，有了它使用 Java 语言就可以很容易地实现异步的、可拓展的和并行的应用。JDK 9 引入了下面这些 API，以便在基于 Java 语言的应用中开发 Reactive Streams：

- java.util.concurrent.Flow
- java.util.concurrent.Flow.Publisher
- java.util.concurrent.Flow.Subscriber
- java.util.concurrent.Flow.Processor

其中，Publisher、Subscriber、Processor 在 RxJava 中也有相同的概念。并且，Processor 也是支持背压的。

除此之外，Java 9 对 Stream API、CompletableFuture API 都有改进。

- 在 Stream 接口中新增了 4 个方法以及其重载方法：dopWhile、takeWhile、ofNullable、iterate。
- 对于 CompletableFuture API 的更新，主要是支持一些延时和超时操作，实用方法和更好的子类化。

其他新增的功能就不一一介绍了，感兴趣的读者可以直接访问 Oracle 的官网查看关于 Java 9 更新的内容。最后，RxJava 也从 2.1.3 版本开始兼容 Java 9。

18.2 其他的 Reactive Streams 项目

1. Reactor 3

本书所讲的 RxJava 库是 JVM 上响应式编程的先驱，也是响应式流规范（Reactive Streams）的基础。RxJava 2 在 RxJava 的基础上做了很多更新。不过 RxJava 库也有其不足的地方。RxJava 产生于响应式流规范之前，虽然可以和响应式流的接口进行转换，但是由于底层实现的原因，使用起来并不是很直观。RxJava 2 在设计和实现上考虑到了与规范的整合，不过为了保持与 RxJava 的兼容性，很多地方在使用时也并不直观。

Reactor 则是完全基于响应式流规范设计和实现的库，没有 RxJava 那样的历史包袱，在使用上更加直观易懂。Reactor 也是 Spring 5 中响应式编程的基础。学习和掌握 Reactor 可以更好地理解 Spring 5 中的相关概念。

Flux 和 Mono 是 Reactor 中的两个基本概念。Flux 表示的是包含 0 到 n 个元素的异步序列。在该序列中可以包含三种不同类型的消息通知：元素的消息、序列结束的消息和序列出错的消息。当消息通知产生时，会调用订阅者中对应的方法 onNext()、onComplete()和 onError()。Mono 表示的是包含 0 或者 1 个元素的异步序列。该序列中同样可以包含与 Flux 相同的三种类型的消息通知。Flux 和 Mono 之间可以进行转换。对一个 Flux 序列进行计数操作，得到的结果是一个 Mono 对象。把两个 Mono 序列合并在一起，得到的是一个 Flux 对象。

Flux 类似于 RxJava 的 Observable，它可以触发零到多个事件，并根据实际情况结束处理或触发错误。Mono 最多只触发一个事件，它跟 RxJava 的 Single 和 Maybe 类似，所以可以把 Mono 用在异步任务完成时发出通知。

从设计概念方面来看，Reactor 看起来有点像 RxJava，不过这决不只是个巧合。这样的设计是为了能够给复杂的异步逻辑提供一套原生的具有 Rx 操作风格的响应式流 API。所以说 Reactor 扎根于响应式流，同时在 API 方面又尽可能向 RxJava 靠拢。

2. Akka Streams

Akka Streams 是以 Akka（Actor）为基础的 Reactive Streams 的实现。Akka Streams 在 Akka 现有的角色模型之上提供了一种更高层级的抽象。

Akka Streams 的目的是提供一个直观的、安全的方式来定制（formulate）流处理，使我们

在资源限制使用的情况下（即控制内存溢出的情况）高效处理。那么是如何实现的呢？Akka stream 实现了一个有 “back pressure” 的特性，它来源于 Reactive Streams，Akka 是该规范的初始成员。

3. Vert.x

Vert.x 是一个异步网络应用开发框架，用来开发高并发、异步、可伸缩、多语言支持的 Web 应用。它类似于 Node.js，不仅支持 JavaScript，还支持 Java、Groovy、Python、Ruby 等其他语言。Netty 作为一个核心处理引擎，改变了当前 J2EE 中常见的阻塞式模型应用的开发，带领大家进入了一个新的 Web 应用开发领域。

首先，Vert.x 非常轻量，可以嵌入我们当前的应用中而不需要改变现有的结构；另一个重要的描述是响应式 —— Vert.x 就是为构建响应式应用（系统）而设计的。Vert.x 是事件驱动的，同时也是非阻塞的。

Vert.x 有一个库 Vert.x Reactive Streams，它提供了 Vert.x 上 Reactive Streams 标准的实现。在处理流式数据方面，Vert.x 有自己的机制，通过这三个类：io.vertx.core.streams.ReadStream、io.vertx.core.streams.WriteStream 和 io.vertx.core.streams.Pump，可以将数据从一个流泵到另一个流时，实现流量控制。这个库为可读流、可写流都提供了实现，这两者分别扮演了 Reactive Streams 中发布者和订阅者的角色。我们可以以对待 Vert.x 中读写流的方式来处理任意 Reactive Streams 的发布者和订阅者。

在 Vert.x 3.5 版本中，它新增了对 RxJava 2 的支持。

4. Slick

Slick（Scala language-integrated connection kit）是 Scala 的一个 FRM（Functional Relational Mapper），即函数式的关系数据库编程工具库。Slick 的主要目的是使关系数据库能更容易、更自然地融入函数式编程模式，它可以让开发者像对待 Scala 集合一样来处理关系数据库表。也就是说，可以用 Scala 集合的那些丰富的操作函数来处理库表数据。Slick 把数据库编程融入到 Scala 编程中，编程人员可以不需要编写 SQL 代码。

Slick 可以把数据库表当作 Scala 语言中的集合来对待。除了能实现 FP 的函数组合外，又避免了嵌入 SQL 语句式的数据库编程，而且也实现了类型安全（type safe），可以由编译器（compiler）在编译时来捕捉语法错误。另一方面，与传统的 ORM 库相比，Slick 可以实现更高

效率的关系表数据提取。

FRM 相比传统的 ORM 最明显的优势是，FRM 是基于多线程的 Future 的数据查询，而 ORM 是单线程的线性执行。

18.3 小结

Java 9 引入了新的 Reactive Streams API，它与 Reactive Streams 项目所创建的接口兼容。由于 Reactive Streams 已经是 JDK 的组成部分，相信它在服务器端会得到更广泛的采用。

最近 Spring 5 也正式发布了。Spring Framework 5.0 的亮点绝对是响应式编程，这是一个重要的思想转变。Spring 的一些子项目包括 Spring Data、Spring Security、Spring Integration 等版本都会提供的响应式编程的功能。

对于 Android 开发而言，响应式是一种资源受限的环境，RxJava 2 是目前看上去最好用的库。

未来，在开发服务端领域，除了选择 Spring 5 外，还可以考虑使用 Vert.x，它是一个理念非常先进的框架。

附录A

RxJava常用的操作符列表

All：判断 Observable 发射的所有的数据项是否都满足某个条件。

Amb：给定多个 Observable，只让第一个发射数据的 Observable 发射全部数据。

And/Then/When：通过模式（And 条件）和计划（Then 次序）组合两个或多个 Observable 发射的数据集

Average：计算 Observable 发射的数据序列的平均值，然后发射这个结果。

Buffer：缓存，可以简单理解为缓存，它定期从 Observable 收集数据到一个集合，然后把这些数据集合打包发射，而不是一次发射一个。

Catch：捕获，继续序列操作，将错误替换为正常的数据，从 onError 通知中恢复。

CombineLatest：当两个 Observables 中的任何一个发射了一个数据时，通过一个指定的函数组合每个 Observable 发射的最新数据（一共两个数据），然后发射这个函数的结果。

Concat：不交错地连接多个 Observable 的数据。

Connect：指示一个可连接的 Observable 开始发射数据给订阅者。

Contains：判断 Observable 是否会发射一个指定的数据项。

Count：计算 Observable 发射的数据个数，然后发射这个结果。

Create：通过调用观察者的方法从头创建一个 Observable。

Debounce：只有在空闲了一段时间后才发射数据，简单来说，就是如果一段时间没有操作，就执行一次操作。

DefaultIfEmpty：发射来自原始 Observable 的数据，如果原始 Observable 没有发射数据，就发射一个默认数据。

Defer：在观察者订阅之前不创建这个 Observable，为每一个观察者创建一个新的 Observable。

Delay：延迟一段时间发射结果数据。

Distinct：去重，过滤掉重复数据项。

Do：注册一个动作占用一些 Observable 的生命周期事件，相当于 Mock 某个操作。

Materialize/Dematerialize：将发射的数据和通知都当作数据发射，或者反过来。

ElementAt：取值，取特定位置的数据项。

Empty/Never/Throw：创建行为受限的特殊 Observable。

Filter：过滤，过滤掉没有通过谓词测试的数据项，只发射通过测试的。

First：首项，只发射满足条件的第一条数据。

flatMap：扁平映射，将 Observable 发射的数据转换为 Observables 集合，然后将这些 Observable 发射的数据平坦化地放进一个单独的 Observable，可以认为是一个将嵌套的数据结构展开的过程。

From：将其他对象或数据结构转换为 Observable。

GroupBy：分组，将原来的 Observable 拆分为 Observable 集合，将原始 Observable 发射的数据按 Key 分组，每一个 Observable 发射一组不同的数据。

IgnoreElements：忽略所有的数据，只保留终止通知（onError 或 onCompleted）。

Interval：创建一个定时发射整数序列的 Observable。

Join：无论何时，如果一个 Observable 发射了一个数据项，只要在另一个 Observable 发射的数据项定义的时间窗口内，就将两个 Observable 发射的数据合并发射。

Just：将对象或者对象集合转换为一个会发射这些对象的 Observable。

Last：末项，只发射最后一条数据。

Map：映射，对序列的每一项都应用一个函数变换 Observable 发射的数据，实质是对序列中的每一项执行一个函数，函数的参数就是这个数据项。

Max：计算并发射数据序列的最大值。

Merge：将两个 Observable 发射的数据组合并成一个。

Min：计算并发射数据序列的最小值。

ObserveOn：指定观察者观察 Observable 的调度程序（工作线程）。

Publish：将一个普通的 Observable 转换为可连接的。

Range：创建发射指定范围的整数序列的 Observable。

Reduce：按顺序对数据序列的每一项数据应用某个函数，然后返回这个值。

RefCount：使一个可连接的 Observable 表现得像一个普通的 Observable。

Repeat：创建重复发射特定的数据或数据序列的 Observable。

Replay：确保所有的观察者收到同样的数据序列，即使他们在 Observable 开始发射数据之后才订阅。

Retry：重试，如果 Observable 发射了一个错误通知，重新订阅它，期待它正常终止辅助操作。

Sample：取样，定期发射最新的数据，等同于数据抽样，有的实现中叫作 ThrottleFirst。

Scan：扫描，对 Observable 发射的每一项数据应用一个函数，然后按顺序依次发射这些值。

SequenceEqual：判断两个 Observable 是否按相同的数据序列。

Serialize：强制 Observable 按次序发射数据并且功能是有效的。

Skip：跳过前面的若干项数据。

SkipLast：跳过后面的若干项数据。

SkipUntil：丢弃原始 Observable 发射的数据，直到第二个 Observable 发射了一个数据，然后发射原始 Observable 的剩余数据。

SkipWhile：丢弃原始 Observable 发射的数据，直到一个特定的条件为假，然后发射原始 Observable 剩余的数据。

Start：创建发射一个函数返回值的 Observable。

StartWith：在发射原来的 Observable 的数据序列之前，先发射一个指定的数据序列或数据项。

Subscribe：收到 Observable 发射的数据和通知后执行的操作。

SubscribeOn：指定 Observable 应该在哪个调度程序上执行。

Sum：计算并发射数据序列的和。

Switch：将一个发射 Observable 序列的 Observable 转换为这样一个 Observable，即它逐个发射那些 Observable 最近发射的数据。

Take：只保留前面的若干项数据。

TakeLast：只保留后面的若干项数据。

TakeUntil：发射来自原始 Observable 的数据，直到第二个 Observable 发射了一个数据或一个通知。

TakeWhile：发射原始 Observable 的数据，直到一个特定的条件为真，然后跳过剩余的数据。

TimeInterval：将一个 Observable 转换为发射两个数据之间所耗费时间的 Observable。

Timeout：添加超时机制，如果过了指定的一段时间没有发射数据，就发射一个错误通知。

Timer：创建在一个指定的延迟之后发射单个数据的 Observable。

Timestamp：给 Observable 发射的每个数据项添加一个时间戳。

To：将 Observable 转换为其他对象或数据结构。

Using：创建一个只在 Observable 生命周期内存在的一次性资源。

Window：窗口，定期将来自 Observable 的数据拆分成一些 Observable 窗口，然后发射这些窗口，而不是每次发射一项。类似于 Buffer，但 Buffer 发射的是数据，Window 发射的是 Observable，每一个 Observable 发射原始 Observable 数据的一个子集。

Zip：打包，使用一个指定的函数将多个 Observable 发射的数据组合在一起，然后将这个函数的结果作为单项数据发射。

附录B
RxJava中常用的library

B.1 Android 相关的 library

1. RxLifecycle

用途：配合 Activity/Fragment 生命周期来管理订阅。

地址：https://github.com/trello/RxLifecycle

2. RxLifecycle

用途：配合 Activity/Fragment 生命周期来管理订阅（知乎版）。

地址：https://github.com/zhihu/RxLifecycle

3. AutoDispose

用途：Uber 开发的开源库，类似于 RxLifecycle，它与 RxLifecycle 的区别是不仅可以在 Android 平台上使用，还可以在 Java 平台上使用，适用的范围更加宽广。

地址：https://github.com/uber/AutoDispose

4. RxBinding

用途：Android UI widgets 的 Rx 扩展。

地址：https://github.com/JakeWharton/RxBinding

5. Retrofit

用途：在 Android 开发中非常流行的网络框架，底层依赖 OkHttp。Retrofit 提供了 RxJava 的适配器。

地址：https://github.com/square/retrofit

6. sqlbrite

用途：SQLiteOpenHelper 的 Rx 封装。

地址：https://github.com/square/sqlbrite

7. Android-ReactiveLocation

用途：Google Play Services API 的 Rx 封装。

地址：https://github.com/mcharmas/Android-ReactiveLocation

8. RxLocation

用途：Location API 的 Rx 封装。

地址：https://github.com/patloew/RxLocation

9. rx-preferences

用途：SharedPreferences 的 Rx 封装。

地址：https://github.com/f2prateek/rx-preferences

10. RxPermissions

用途：Android runtime permissions 的 Rx 封装。

地址：https://github.com/tbruyelle/RxPermissions

11. ReactiveNetwork

用途：用于网络状态监听。

地址：https://github.com/pwittchen/ReactiveNetwork

12. RxDownload

用途：基于 RxJava 打造的下载工具，支持多线程下载和断点续传。

地址：https://github.com/ssseasonnn/RxDownload

B.2 Java 服务端相关的 library

1. Hystrix

用途：它是 Spring Cloud 的标准组件，具备拥有回退机制和断路器功能的线程和信号隔离、请求缓存和请求打包，以及监控和配置等功能。Hystrix 中使用了大量的 RxJava。

地址：https://github.com/Netflix/Hystrix/

2. rxjava-jdbc

用途：使用 RxJava 流式处理 JDBC 连接，还支持语句的函数式组合。

地址：https://github.com/davidmoten/rxjava-jdbc

3. Camel RX

用途：一个用于 Apache Camel 的 RxJava 兼容层。

地址：http://camel.apache.org/rx.html

4. vertx-rx

用途：Vertx 的 RxJava 模块。

地址：https://github.com/vert-x3/vertx-rx

5. async-http-client

用途：异步 Http 和 WebSocket 客户端的库，由 Java 实现，支持 RxJava 的扩展。

地址：https://github.com/AsyncHttpClient/async-http-client

電子工業出版社 PUBLISHING HOUSE OF ELECTRONICS INDUSTRY http://www.phei.com.cn
Broadview WWW.BROADVIEW.COM.CN
博文视点 · IT出版旗舰品牌

十载耕耘奠定专业地位

博文视点诚邀精锐作者加盟

以书为证彰显卓越品质

《C++Primer（中文版）（第5版）》、《淘宝技术这十年》、《代码大全》、《Windows内核情景分析》、《加密与解密》、《编程之美》、《VC++深入详解》、《SEO实战密码》、《PPT演义》……

“圣经”级图书光耀夺目，被无数读者朋友奉为案头手册传世经典。

潘爱民、毛德操、张亚勤、张宏江、咎辉Zac、李刚、曹江华……

“明星”级作者济济一堂，他们的名字熠熠生辉，与IT业的蓬勃发展紧密相连。

十年的开拓、探索和励精图治，成就**博**古通今、**文**圆质方、**视**角独特、**点**石成金之计算机图书的风向标杆：博文视点。

“凤翱翔于千仞兮，非梧不栖”，博文视点欢迎更多才华横溢、锐意创新的作者朋友加盟，与大师并列于IT专业出版之巅。

英雄帖

江湖风云起，代有才人出。
IT界群雄并起，逐鹿中原。
博文视点诚邀天下技术英豪加入，
指点江山，激扬文字
传播信息技术，分享IT心得

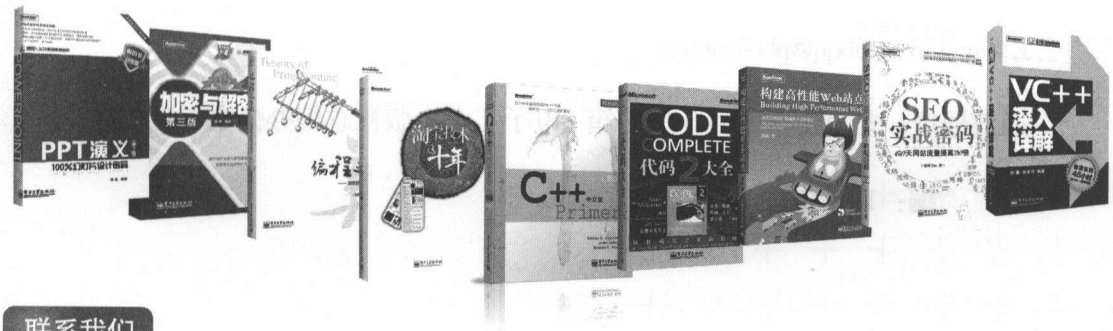
• 专业的作者服务 •

博文视点自成立以来一直专注于IT专业技术图书的出版，拥有丰富的与技术图书作者合作的经验，并参照IT技术图书的特点，打造了一支高效运转、富有服务意识的编辑出版团队。我们始终坚持：

善待作者——我们会把出版流程整理得清晰简明，为作者提供优厚的稿酬服务，解除作者的顾虑，安心写作，展现出最好的作品。

尊重作者——我们尊重每一位作者的技术实力和生活习惯，并会参照作者实际的工作、生活节奏，量身制定写作计划，确保合作顺利进行。

提升作者——我们打造精品图书，更要打造知名作者。博文视点致力于通过图书提升作者的个人品牌和技术影响力，为作者的事业开拓带来更多的机会。



联系我们

博文视点官网：http://www.broadview.com.cn　　CSDN官方博客：http://blog.csdn.net/broadview2006/
投稿电话：010-51260888　88254368　　投稿邮箱：jsj@phei.com.cn

新浪微博 weibo.com　@博文视点Broadview　　微信公众账号　博文视点Broadview

出版编辑联络：安 娜
微信&QQ：80303489
邮箱：anna@phei.com.cn

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

目前响应式编程越来越流行了，这不仅是一个简单的函数库，更是一种编程理念的突破。本书深入浅出，通过实例介绍了RxJava，整本书从一些基本的编程接口展开，逐步讲解了如何解决一些生产环境的实战问题。即便刚开始接触响应式编程的读者，读完本书后也可以成为立即提交生产代码的程序员。

——Branch Metrics首席数据科学家　余侃

RxJava，一个风靡全球的响应式编程开源库，各大IT公司(包括Google、Facebook、Uber)和许多创业公司都在应用，一个小巧的库能帮助解决复杂的异步响应问题。本书深入浅出地介绍了RxJava的方方面面，一定能帮助你真正理解RxJava编程的精髓。

——Google Lens tech leader　雷加贝

博文视点Broadview
新浪微博 weibo.com
@博文视点Broadview

责任编辑：安　娜
封面设计：侯士卿

上架建议：移动开发

ISBN 978-7-121-33722-2

定价：79.00元