# 探索专为 Android 而设计的 RxJava 2

RxJava 的下一代版本正在紧锣密鼓地开发当中。尽管在新版本中，很多操作符并未发生变化，但是创建可观察对象 (observable creation)、订阅管理 (subscription management) 以及背压 (backpressure) 都进行了彻头彻尾地改进。在本次[GOTO Copenhagen 2016](https://gotocon.com/cph-2016/) 的讲演中，Jake 将探讨 RxJava 2 进行了何种改进，以及这些改进背后的原因。您将学习到该如何将第三方库和应用同时迁移到 RxJava 2 当中，以及该如何在两个 RxJava 版本之间互相操作。

### 为什么要响应式？[(0:53)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,3,true);)

为什么突然之间，您听见大家都在谈论起响应式编程 (Reactive) 了呢？因为除非您可以完全用同步操作来编写整个应用，否则的话，应用中或多或少都会有异步源的存在，而这终将打破我们习惯的传统命令式编程风格。所谓的「打破」，并不是说命令式编程将被废除，而是说在某种意义上，它导致编程的复杂性极度增大，这个时候您就不会觉得命令式编程仍然是一个不错的选择了。

这里有一个例子可以帮助大家明白，为什么我认为这是一个大问题：

**interface** **UserManager** **{**

User **getUser();**

**void** **setName(**String name**);**

**void** **setAge(int** age**);**

**}**

UserManager um **=** **new** UserManager**();**

System**.**out**.**println**(**um**.**getUser**());**

um**.**setName**(**"Jane Doe"**);**

System**.**out**.**println**(**um**.**getUser**());**

我们从一个非常简单的类开始，这是一个用户对象，里面有一些相应的属性。如果我们只用面对同步操作的话，也就是只用操作一个单独的线程的话，这样我们便可以通过这种操作来输出我们所期望的结果：创建一个对象实例，然后输出用户的相关信息，还可以修改其中的一些属性，再输出用户的相关信息。

当我们以异步的方式开始建模的时候，就出大问题了。假设我们需要反映出服务器端当中属性的变更操作，那么下面的这两个方法就需要异步进行。那么我们该如何修改我们的代码，从而实现这个操作呢？

您可以选择什么都不做：去假设更新服务器的异步调用是永远成功的，这样您就可以在本地执行变更，这样当我们输出用户对象的时候，就可以立即反映出属性的变更操作。显然，这并不是一个好主意。因为网络是非常脆弱的，服务器很有可能会返回一个错误，因此您就必须在本地协调处理好相应的状态。

我们可以在每次异步调用成功的时候，提供一个 Runnable，这种做法很简单：这就是所谓****「响应式」****的概念，因为我们只在确保网络请求成功进行了数据变更的时候，才去更新显示的数据。

**interface** **UserManager** **{**

User **getUser();**

**void** **setName(**String name**,** Runnable callback**);**A

**void** **setAge(int** age**,** Runnable callback**);**B

**}**

UserManager um **=** **new** UserManager**();**

System**.**out**.**println**(**um**.**getUser**());**

um**.**setName**(**"Jane Doe"**,** **new** Runnable**()** **{**

**@Override** **public** **void** **run()** **{**

System**.**out**.**println**(**um**.**getUser**());**

**}});**

然而，我们并没有对任何可能发生的问题进行处理，比如说如果网络通讯失败的话。或许我们需要创建相应的监听器 (listener)，当错误发生的时候，我们可以做一些错误处理。

UserManager um **=** **new** UserManager**();**

System**.**out**.**println**(**um**.**getUser**());**

um**.**setName**(**"Jane Doe"**,** **new** UserManager**.**Listener**()** **{**

**@Override** **public** **void** **success()** **{**

System**.**out**.**println**(**um**.**getUser**());**

**}**

**@Override** **public** **void** **failure(**IOException e**)** **{**

*// TODO show the error...*

**}});**

我们可以向用户报告错误、可以选择自动重连网络。而这些操作，是人们需要向运行在单线程（传统上的 Android 主线程）之中的代码中混入异步代码的必由之路。

而随着异步代码的增多，问题也逐步显现。您必须要支持多个异步调用：比如说当用户在填写表单的时候，需要同时更改应用当中的多个属性，或者诸如此类的异步调用流 (flow)，也就是当第一个调用成功之后，就必须要触发下一个异步调用，而这些调用都有可能成功或者失败。

**public** **final** **class** **UserActivity** **extends** Activity **{**

**private** **final** UserManager um **=** **new** UserManager**();**

**@Override** **protected** **void** **onCreate(**Bundle savedInstanceState**)** **{**

**super.**onCreate**(**savedInstanceState**);**

setContentView**(**R**.**layout**.**user**);**

TextView tv **=** **(**TextView**)** findViewById**(**R**.**id**.**user\_name**);**

tv**.**setText**(**um**.**getUser**().**toString**());**

um**.**setName**(**"Jane Doe"**,** **new** UserManager**.**Listener**()** **{**

**@Override** **public** **void** **success()** **{**

tv**.**setText**(**um**.**getUser**().**toString**());**

**}**

**@Override** **public** **void** **failure(**IOException e**)** **{**

*// TODO show the error...*

**}**

**});**

**}}**

我们还要记住，我们是在 Android 的运行环境当中，因此就需要有很多的考虑。例如，在这个成功回调当中，我们可能会直接将信息填充到 UI 上，但是问题是，Android 上 Activity 的存在是非常的短暂的，它们随时都可能会消失。如果这个异步操作在 UI 消失之后仍然还对 UI 进行了操作，那么就会遇到麻烦。

因此我们****有必要****去解决这个问题。这里有很多方法：我们可以在进行视图修改之前进行一些状态检查；还可以创建一个匿名类型 (anonymous type)，虽然这可能会导致暂时的内存泄漏，但是这可以让我们始终保持对这个 Activity 的引用；如果这个 Activity 消失，那么异步调用仍然可以在后台进行。

最后一个问题是，我们没有定义这些回调能在哪些线程上运行。它们可能会在后台线程当中返回，因此我们需要让这些线程跳回到主线程当中。

**public** **final** **class** **UserActivity** **extends** Activity **{**

**private** **final** UserManager um **=** **new** UserManager**();**

**@Override** **protected** **void** **onCreate(**Bundle savedInstanceState**)** **{**

**super.**onCreate**(**savedInstanceState**);**

setContentView**(**R**.**layout**.**user**);**

TextView tv **=** **(**TextView**)** findViewById**(**R**.**id**.**user\_name**);**

tv**.**setText**(**um**.**getUser**().**toString**());**

um**.**setName**(**"Jane Doe"**,** **new** UserManager**.**Listener**()** **{**

**@Override** **public** **void** **success()** **{**

runOnUiThread**(new** Runnable**()** **{**

**@Override** **public** **void** **run()** **{**

**if** **(**isDestroyed**())** **{**

tv**.**setText**(**um**.**getUser**().**toString**());**

**}**

**}**

**});**

**}**

**@Override** **public** **void** **failure(**IOException e**)** **{**

*// TODO show the error...*

**}**

**});**

**}}**

我们用一堆与代码的实际意图无关的东西堵塞了这个 Activity，而我们仅仅只是简单地启动这个异步操作并对异步结果进行处理而已，并且这个异步请求还是立即调用的。我们并没有处理诸如禁用用户输入、处理按钮点击，以及多个文本框处理等情况。类似的代码只能用来处理简单的情况，一旦您把它放到真实的应用当中，把这些问题一复合起来，您就不得疲于管理一堆堆的状态和进行一堆堆的 Activity 检查。

### 用响应式的思路解决问题 [(6:16)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,30,true);)

就某种意义而言，应用当中的所有操作都可以是异步的。比如说网络操作，我们向服务器发送请求，然后在响应了很长时间之后得到返回结果。这时候我们不能阻塞主线程，所以它必须在后台线程当中完成。此外还有类似文件系统之类的东西，不管是数据库存储也好，还是偏好设置共享也好，都不能阻塞主线程。

用户同样也是一种异步数据源。我们在 UI 中向用户展示数据，然后他们通过按钮点击或者文本框输入等操作来做出回应。这些操作都是异步发生的。我们并不能同步获取用户的输入，而必须等待，直到用户进行了操作。

很多人认为，他们可以编写出单线程的应用程序，这个程序默认在主线程上运行，但是实际上，UI 就是一个很大的异步源。您必须要通过有意义的方式来对用户的输入做出回应。

应用在不同时间当中都会有不同的操作流发生，因此应用都必须全盘接受并做出回应，并且您不能让主线程被阻塞。此外当某些数据出乎意料地从异步源当中出现的时候，用户是不希望应用没有对其作出回应或者发生崩溃的。总而言之，代码需要对这一切负责。

这就带来了极高的复杂度：您必须要在 Fragment 当中对所有的状态进行维护，而这些多个异步源可能会随时生成和接受数据，因此就必须进行协调管理。另外 Android 平台本身的因素都需要考虑到，因为这是一个充满了异步操作的平台，比如说推送通知、系统广播、配置更改等等。用户还可以随时对设备进行旋转，如果您的代码没有对此进行处理的话，那么应用很可能就会发生崩溃。

### 网络请求 [(9:19)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,37,true);)

****除非您可以让整个系统全都以同步方式运行，否则随便一个异步源都会中断命令式编程****。

不存在异步网络请求的应用是很难找到的。通常情况下，我们都会使用硬盘（数据库），而这本质上就是一种异步源，此外 UI 本身也应该被认作为一种异步源，所以一般情况下，Android 当中的大部分操作基本都是异步的。如果您坚持使用传统的命令式编程和状态管理技术，那么最终遭殃的始终是自己。

我们应该设想一种模型，我们所编写的代码应该位于模型与异步源的中间，作为状态仲裁器 (state arbiter) 来使用，而不是试图去协调所有异步源的操作，我们可以直接与异步操作建立关联，从而消除我们进行管理的难度。我们可以将 UI 的更变直接与数据库变更联系起来，建立订阅 (subscribe) 关系，只有在数据发生变更的时候才做出回应。我们可以在按钮被点击时，直接让数据库更变和网络调用立即响应，而不是接收这个点击事件，然后再将其分配到合适的地方。

同样地，当网络响应回复之后，如果可以直接进行数据变更就非常好。我们已经知道，当数据变更的时候，UI 就会随之发生变更，因此这样我们就摈弃了自己对这个过程进行管理的责任。如果 Android 平台发生了某些异步操作，比如说设备旋转，这个工作方式也能很好地进行，因为它将会自动反映在 UI 之上，或者也可以自动启动一些后台任务。

这样，我们最终便可以舍弃了很多以前必须要编写的代码，不再劳心去维护这些状态。尽管我们仍然需要编写代码，但是我们的重点放在了该如何用一种有意义的方式将这些操作连接在一起，而不是试图去管理所有的状态和事件。

### RxJava [(11:14)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,40,true);)

这就是我们需要 RxJava 的原因所在。实际上，它已经变成了一个专为 Android 而生的响应式第三方库，主要是因为它是第一个全方位可用于 Java 支持的响应式第三方库。RxJava 2 保留了我们需要在 Android 上支持的所有 Java 版本。

归结起来，它由三个主要的部分组成：

* 用于表示数据源的一组类
* 用于侦听数据源的一组类
* 用于修改与合并数据的一组方法

当您开始侦听数据源的时候，它将开始或者停止执行某些操作。您可以将其想象为一种网络请求，除非您开始对网络回应进行侦听，否则的话这个网络请求将永远不会触发。并且如果您在数据源任务完成之前取消了对数据源的订阅，那么它很可能会取消这个网络请求。

这些操作可以是同步的，也可以是异步的，因此您可以构造出一个类似网络请求的操作，它会在某个代码块当中运行，但是却是运行在后台线程之上的。或者，它也可以是完全异步的，就像我们在 Android 中弹出 Activity 或者单击 UI 当中的某个组件，这些都可以被视为异步操作。

通常情况下，一个网络请求只会产生一条回应，但是只要您的 UI 还存在于那里，即便您只订阅了一个按钮，您会发现，按钮的单击事件流却可能是无限的。此外，它们所产生的回应也可能是空的，因此这种类型的数据源实际上只有两种功能：成功或者失败，它不会生成任何的数据。您可以将其视为类似写入数据库或者写入文件的操作。它实际上并不会返回相关的回应或者数据；它要么成功完成、要么失败。这种完成、失败的响应方面是由 RxJava 当中的数据源，结合那些所谓「终端事件 (terminal event)」而构建的。这类似于那些可以正常返回、也可以抛出异常的方法。

这种响应也有可能永远不会终止。让我们回到之前的按钮点击操作示例来，如果我们将按钮点击事件构建为数据源，那么只要您的这个 UI 还存在，那么这个响应永远都不会结束。只有当 UI 消失，并且您取消了该按钮点击事件的订阅之后，才可能会结束。在此之前，它将永远不会进入到完成状态。

这一切相当于实现了传统的观察者模式。我们有某些可以产生数据的组件，然后我们只想知道它们所产生的数据是什么，因此我们所要做的仅仅只是观察而已。所以我们可以添加一个监听器，当事件发生的时候就收到相应的通知。

### Flowable 对比 Observable [(14:50)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,50,true);)

RxJava 2 当中表示这种过程的类型有两种：分别是 Flowable 和 Observable。这两种类型最终都会构建出相同类型的数据，也就是可能会生成 0 到 n 个数据。它可以为空，也可能只有单独的一条数据，或者也可能会存在许多，这些过程可能会成功结束，也可能会发生错误。为什么我们需要两种类型来表示相同的数据结构呢？

这归结为所谓的「背压 (backpressure)」。这是一种允许您将某个数据源中的操作进行延时的东西。我们的系统资源是有限的，背压是一种告诉所有发送数据的组件减缓它们发送数据速度的方法，因为您很可能不能像发送数据一样，迅速地完成数据的处理。

在 RxJava 1 当中，系统当中的每种类型都存在背压，但是在 RxJava 2 当中，我们提炼出了两种类型。尽管所有的类型都存在背压，但是并不是所有的数据源实际上都要去实现它，否则的话您会很崩溃的。由于背压的存在，诸如继承之类的东西，都必须要进行良好的设计和预先计算。在 RxJava 2 当中，您可以在类型系统中指定是否支持背压。

举个例子，如果我们有一个触摸事件的数据源，这种数据源我们是无法真正让其减缓的：我们不能告诉用户，「您现在画了一半的曲线，现在您可以停下来了，等我处理完绘制，然后您再继续」。我们可以通过其他方式做到这一点，例如禁用按钮或者通过显示其他 UI 来尝试减缓用户的操作速度，但是数据源本身是不可能减速的。

您可以与数据库作比较，假设我们有一个很庞大的数据集，也许我们只想提取其中的某几行数据。这时候数据库可以对其很好地建模：它存在指针的概念。但是触摸事件流就无法模拟这一点，因为我们没有办法将用户的手指移开或者减缓他们手指的移动速度。在 RxJava 1 当中，您会看到这两种类型都使用 Observable 来实现，所以在运行时，您可能会试图尝试和使用背压，而这会产生一个异常，导致你的应用崩溃。

Observable**<**MotionEvent**>** events

**=** RxView**.**touches**(**paintView**);**

Observable**<**Row**>** rows

**=** db**.**createQuery**(**"SELECT \* …"**);**

*// MissingBackpressureException*

在 RxJava 2 当中，我们则使用了不同的类型实现，因为基本上，一个是支持背压，而另一个是不支持背压的。

Observable**<**MotionEvent**>** events

**=** RxView**.**touches**(**paintView**);**

Flowable**<**Row**>** rows

**=** db**.**createQuery**(**"SELECT \* …"**);**

由于这是两种不同的类型，因此必须用某种方式来将背压暴露出来，但是由于它们也可以构建出相同类型的数据，因此它们也必须用某种方式将数据推送到回调当中。用于监听来自这些数据源的事件的两个接口看起来非常相似。第一个方法称作 onNext，这是一个可以将数据交付给下一个步骤的方法。只要Observable 或者 Flowable 产生了一个或者多个数据，那么对于每个数据而言，都会被这个方法所调用，从而允许您对这些数据进行处理。

Observable**<**MotionEvent**>** Flowable**<**Row**>interface** **Subscriber<**T**>** **{**

**void** **onNext(**T t**);**

**void** **onComplete();**

**void** **onError(**Throwable t**);**

**void** **onSubscribe(**Subscription s**);}**

**interface** **Disposable** **{**

**void** **dispose();}**

**interface** **Observer<**T**>** **{**

**void** **onNext(**T t**);**

**void** **onComplete();**

**void** **onError(**Throwable t**);**

**void** **onSubscribe(**Disposable d**);}**

**interface** **Subscription** **{**

**void** **cancel();**

**void** **request(long** r**);}**

这个操作很可能会无穷无尽。如果您对按钮点击事件进行了侦听，那么这个 onNext 方法会在每次用户点击了按钮之后被调用。对于非无尽源而言，我们存在两种终端事件。一个是 onComplete，表示操作已完成、已成功，另一个是onError，表示操作发生了错误，此外对 onNext 回调进行处理而导致的异常抛出也可以在这里看到。这两个就是所谓的终端事件，这意味着当您得到其中的某一个之后，将永远不会得到另一个回调。RxJava 1 和 RxJava 2 的不同之处在于最后那个被称为 onSubscribe 的方法。

如果您对 RxJava 1 有所了解的话，那么您会知道这是一个新的东西：当您订阅了 Observable 或者 Flowable 的时候，实际上您会创建出一个占用资源的对象，在您完成了相关的操作之后，您可能会经常需要对这个对象进行清理。这个onSubscribe 回调会在您开始监听 Observable 或 Flowable 的时候立即被调用，然后它便会开始处理这两种类型中的某一个。

对于 Observable 而言，将允许您对其调用 dispose 方法，这本质上是说：「我已经不再需要它了，因此我不再需要处理之后的调用了」。这对于网络请求而言，就是将取消这个网络请求。如果您正在对按钮点击事件监听，这个操作意味着您不再想去接收按钮点击事件，这会在视图上对监听器执行 onSet 操作。对于 onFlowable 类型也是如此。即便这个回调有着不同的名称，但是它们的使用是相同的。它可以像 Disposable 的 dispose 方法一样，可以立即取消这个方法调用。唯一的区别是：它使用的是这个被称为 request 的方法，也就是背压在 API 中展示自己的地方。

这个 request 方法告诉 Flowable 您需要更多的数据。我将在这里构建一个小图表，以便更好地说明这些东西是如何相互关联的。我们可以表示任何类型的发送对象 (emission)。它可以为零，也可以只有一个，也可以有多个，它可能会成功完成，也可能会发生错误。两者之间的唯一区别是：一个具有背压，另一个没有背压。

### 响应流 [(22:10)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,65,true);)

我想要解释一下为什么 Disposable 和订阅类型的命名为何如此不同，为什么它们的方法，一个是 dispose，而另一个是cancel。为什么不是对另一个进行扩展，然后添加 request 方法即可。原因是有一种名为响应流 (reactive stream) 的标准规范。这是一种主动让多个响应聚合在一起的东西，并且，我们可以为 Java 当中的响应式库制定一套标准的接口，最终我们可能会有这四种接口。

在中间部分，您会看到订阅者类型和订阅类型。这实际上是规范的一部分，这就是为什么一次性类型和观察者类型的名字会如此不同。

**interface** **Publisher<**T**>** **{**

**void** **subscribe(**Subscriber**<?** **super** T**>** s**);}**

**interface** **Subscriber<**T**>** **{**

**void** **onNext(**T t**);**

**void** **onComplete();**

**void** **onError(**Throwable t**);**

**void** **onSubscribe(**Subscription s**);}**

**interface** **Subscription** **{**

**void** **request(long** n**);**

**void** **cancel();}**

**interface** **Processor<**T**,** R**>** **extends** Subscriber**<**T**>,** Publisher**<**R**>** **{}**

之所以不同，是因为它们是标准的一部分，这一点我们是无法改变的。这样做的好处是，由于这是一种标准，如果您必须要在事件流当中使用两种不同的第三方库，只要它们都实现了这个标准，那么您就可以在它们之间无缝切换，尽管这不会频繁地在 Android 上发生。我现在对左列这边进行些许修改，改为能够实现响应流规范的类型，这意味着它们是支持背压的，而右边的这些类型是没有背压的。

**interface** **UserManager** **{**

User **getUser();**

**void** **setName(**String name**);**

**void** **setAge(int** age**);}**

**interface** **UserManager** **{**

Observable**<**User**>** **getUser();**

**void** **setName(**String name**);**

**void** **setAge(int** age**);}**

回到 UserManager 当中来，之前我们将用户从这个类当中单独提取了出来，然后并进行展示，而这是我们认为最合适的方法。现在我们可以对其重新进行构建，构建为一个 Observable<User>。它是用户对象的数据源，每当用户发生变更的时候，我们都将收到通知，从而可以通过展示这个用户从而对变更做出回应，这样我们便不用试图去通过系统中发生的其他事件，去猜测何时才是最适当的时间来显示这个用户。

RxJava 当中有几个专门的异步源类型，也是 Observable 的子集，因此存在三个类型。第一个被称为 Single。它当中只会有一个单独的数据，或者一个错误，因此它不与事件流不太像，而更像只包含一个数据的异步源。它也不存在背压。它的运作方式类似于 scaler。调用一个方法，然后要么得到一个返回值，要么这个方法抛出了异常。Single 本质上是用相同的理念构建的。当您订阅了一个 Single，那么您要么就取得数据，要么就收到错误。

区别在于它是响应式的。Completable 类似于返回值为 void 的方法。它要么成功完成但是没有数据返回，要么就抛出异常，要么就返回一个错误。您可以将这个方法视为是以响应式的方式运行的。它有一组您可以运行的代码，它们要么全部成功，要么就失败。

同 RxJava 1 相比，RxJava 2 新增了一个新的类型，名为 Maybe。说明这里可能会有一个数据、错误，也可能什么都没有。可以将其视为一种返回可空值的方法。这种方法如果不抛出异常的话，将总是会返回一些东西，但是返回值可能为空，也可能不为空。我们将立马来看一下该如何使用它，它与可空值的概念相同，只是它是以响应式的方式运作的。在 RxJava 2 当中，没有类型能够实际构建符合响应流的东西，它们只能够以 Observable 或者无背压的方式构建。

**interface** **UserManager** **{**

Observable**<**User**>** **getUser();**

**void** **setName(**String name**);**

**void** **setAge(int** age**);}**

**interface** **UserManager** **{**

Observable**<**User**>** **getUser();**

Completable **setName(**String name**);**

Completable **setAge(int** age**);}**

如果我们的 setName 和 setAge 是以异步方式调用的，那么它们可能成功，也可能失败，这并不会返回任何数据，这就是为什么我们会将它们构建为 Completable 类型。

### 数据源的创建 [(26:23)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,85,true);)

这个例子将展示如何创建异步源，以及如何将已在响应源当中使用的东西封装起来：

Flowable**.**just**(**"Hello"**);**

Flowable**.**just**(**"Hello"**,** "World"**);**

Observable**.**just**(**"Hello"**);**

Observable**.**just**(**"Hello"**,** "World"**);**

Maybe**.**just**(**"Hello"**);**

Single**.**just**(**"Hello"**);**

所有的类型都包含有静态方法，从而允许您使用类似标量的东西来创建它们。您也可以通过诸如数组之类可迭代的东西来创建它们，但是只有两个是我认为最有用、最适合的。

OkHttpClient client **=** *// …*

Request request **=** *// …*

Observable**.**fromCallable**(new** Callable**<**String**>()** **{**

**@Override** **public** String **call()** **throws** Exception **{**Y

**return** client**.**newCall**(**request**).**execute**();**Z

**}});**

第一个方法是 fromCallable。fromCallable 本质上会构建一些可以返回单一值的同步操作。在这种情况下，我委托给某些被称为 getName 的虚构方法。fromCallable 的好处在于，允许从可调用对象（标准的 Java 接口）抛出异常，这意味着我们可以使用已被检查过的异常来构造可能会失败的对象。如果我们有一个相关的 HTTP 请求，我们想要它可能会抛出 I/O 异常，那么我们可以把它放到 fromCallable 当中。返回的 Observable 将在订阅的时候执行该请求，如果这个请求抛出了异常，那么我们就可以在 onError 那里进行处理。如果请求成功完成，那么我们将在 onNext 得到回应。

Flowable**.**fromCallable**(()** **->** "Hello"**);**

Observable**.**fromCallable**(()** **->** "Hello"**);**

Maybe**.**fromCallable**(()** **->** "Hello"**);**

Maybe**.**fromAction**(()** **->** System**.**out**.**println**(**"Hello"**));**

Maybe**.**fromRunnable**(()** **->** System**.**out**.**println**(**"Hello"**))**

Single**.**fromCallable**(()** **->** "Hello"**);**

Completable**.**fromCallable**(()** **->** "Ignored!"**);**

Completable**.**fromAction**(()** **->** System**.**out**.**println**(**"Hello"**));**

Completable**.**fromRunnable**(()** **->** System**.**out**.**println**(**"Hello"**));**

fromCallable 可用于五种类型。这可以用于构造包含单数据块的同步源。这将是您在命令式编程当中所使用的一个方法，也就是一个能够返回值的方法。在响应式编程当中，fromCallable 将是您用于构造此类型的方法。

还有两个额外的方法，对了可能还有 completable，它们允许您构造不返回值的方法。正如我所说，只是一个可执行的方法而已，只是它们现在已经变成响应式的了。

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

**@Override**

**public** **void** **subscribe(**ObservableEmitter**<**String**>** e**)** **throws** Exception **{**

e**.**onNext**(**"Hello"**);**

e**.**onComplete**();**

**}});**

创建 Observable 最强大的方法就是 create 了。每当有一个新的订阅者的时候，它所带的那个回调都会被调用。我们将这个东西称之为发射器 (emitter)，发射器就是正在侦听的对象。我们可以获取数据，然后将其发送给发射器。在这个例子当中，我所做的就是同步发送一个数据，之后就可以完成事件流的运转，因为它已经成功完成了。

我将把它转换为一个 lambda 表达式，然后清理掉一些样板代码。这样我就可以发送多个数据。

Observable**.**create**(**e **->** **{**

e**.**onNext**(**"Hello"**);**

e**.**onNext**(**"World"**);**

e**.**onComplete**();});**

与 fromCallable 不同的是，我可以多次调用 onNext。这样做的另一个优点在于，我们现在可以构建异步数据了。如果我有一个 HTTP 请求，我需要异步运行它而不是同步运行，那么我就可以在 HTTP 回调当中调用发射器当中的 onNext 方法。

这种创建方法的另一个好处在于，它在人们取消订阅的时候执行一些操作。

OkHttpClient client **=** *// …*

Request request **=** *// …*

Observable**.**create**(**e **->** **{**

Call call **=** client**.**newCall**(**request**);**

e**.**setCancelation**(()** **->** call**.**cancel**());**

call**.**enqueue**(new** Callback**()** **{**

**@Override** **public** **void** **onResponse(**Response r**)** **throws** IOException **{**

e**.**onNext**(**r**.**body**().**string**());**

e**.**onComplete**();**

**}**A

**@Override** **public** **void** **onFailure(**IOException e**)** **{**

e**.**onError**(**e**);**

**}**

**});});**

如果有人停止了对这个 HTTP 请求的侦听，那么就没有理由再继续执行这个请求。我们现在可以添加取消操作，取消 HTTP 请求并清除相关的资源。

这对 Android 而言也是非常有用的，因为这也是我们如何构建与 UI 交互的方式。当您订阅了一个 Observable 之后，我们开始侦听按钮的点击，然后您取消了订阅操作，那么就删除这个侦听器，这样我们就不会泄露该视图的应用。

Flowable**.**create**(**e **->** **{** … **});**

Observable**.**create**(**e **->** **{** … **});**

Maybe**.**create**(**e **->** **{** … **});**

Single**.**create**(**e **->** **{** … **});**

Completable**.**create**(**e **->** **{** … **});**

可以对这五种类型使用这个发射器来创建 Observable。

### 数据源的观察 [(30:44)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,105,true);)

Flowable**<**String**>**

**interface** **Subscriber<**T**>** **{**

**void** **onNext(**T t**);**

**void** **onComplete();**

**void** **onError(**Throwable t**);**

**void** **onSubscribe(**Subscription s**);}**

**interface** **Subscription** **{**

**void** **cancel();**

**void** **request(long** r**);}**

Observable**<**String**>**

**interface** **Observer<**T**>** **{**

**void** **onNext(**T t**);**

**void** **onComplete();**

**void** **onError(**Throwable t**);**

**void** **onSubscribe(**Disposable d**);}**

**interface** **Disposable** **{**

**void** **dispose();}**

当订阅 Observable 的时候，您不会直接使用这些接口，而是使用 subscribe 方法来直接开始侦听。

由于上面这四种方法的存在，您可能会陷入到一个尴尬的境地。我该如何处理这个对象，该如何取消订阅呢？

Observable**<**String**>** o **=** Observable**.**just**(**"Hello"**);**

o**.**subscribe**(new** DisposableObserver**<**String**>()** **{**

**@Override** **public** **void** **onNext(**String s**)** **{** … **}**

**@Override** **public** **void** **onComplete()** **{** … **}**

**@Override** **public** **void** **onError(**Throwable t**)** **{** … **}});**

相反，我们有一个名为 DisposableObserver 的类型，它将自动为您处理第四个方法，并允许您只用关心来自 Observable本身的通知。但是，我们这里该如何调用 dispose 方法呢？

Observable**<**String**>** o **=** Observable**.**just**(**"Hello"**);**

o**.**subscribe**(new** DisposableObserver**<**String**>()** **{**

**@Override** **public** **void** **onNext(**String s**)** **{** … **}**

**@Override** **public** **void** **onComplete()** **{** … **}**

**@Override** **public** **void** **onError(**Throwable t**)** **{** … **}});**

您可以做的一件事就是持有这个观察者对象。它实现了 Disposable，因此您可以调用 dispose 方法，它会将其转发到过程链当中。但是，在 RxJava2 当中有一个名为 subscribeWith 的新方法。

这允许您以类似 RxJava1 当中的方式来使用它，现在它将会返回一个可以直接调用 dispose 方法的对象。在 RxJava1 当中，这个操作被称为订阅。在所有的 Observable 当中，它属于是 Disposable 的。

如果大家知道复合订阅 (composite subscription) 的话，那么我们也有复合订阅的操作，这允许您订阅多个事件流，接收那些返回的 Disposable 对象，然后将它们添加到 Disposable 列表当中，这样您就可以取消订阅多个事件流。在 Android 上您会看到很多这样的用法，也就是您有一个针对 Activity 或者 Fragment 的复合 Disposable，这样您就可以在onDestroy 或者其他合适的生命周期当中取消订阅。

Observable**<**String**>** o **=** Observable**.**just**(**"Hello"**);**

o**.**subscribeWith**(new** DisposableObserver**<**String**>()** **{** … **});**

Maybe**<**String**>** m **=** Maybe**.**just**(**"Hello"**);**

m**.**subscribeWith**(new** DisposableMaybeObserver**<**String**>()** **{** … **});**

Single**<**String**>** s **=** Single**.**just**(**"Hello"**);**

s**.**subscribeWith**(new** DisposableSingleObserver**<**String**>()** **{** … **});**

Completable c **=** Completable**.**completed**();**

c**.**subscribeWith**(new** DisposableCompletableObserver**<**String**>()** **{** … **});**

这里有四种非背压类型，其中一个使用了 Observable，实际上这里可能用 Flowable 会更合适，即便 Flowable 使用了订阅回调，而不是 Disposable 的。我们在 RxJava 2 当中提供的类型允许您以相同的方式来构造它。

Flowable**<**String**>** f **=** Flowable**.**just**(**"Hello"**);**

Disposable d1 **=** f**.**subscribeWith**(new** DisposableSubscriber**<**String**>()** **{** … **});**

Observable**<**String**>** o **=** Observable**.**just**(**"Hello"**);**

Disposable d2 **=** o**.**subscribeWith**(new** DisposableObserver**<**String**>()** **{** … **});**

Maybe**<**String**>** m **=** Maybe**.**just**(**"Hello"**);**

Disposable d3 **=** m**.**subscribeWith**(new** DisposableMaybeObserver**<**String**>()** **{** … **});**

Single**<**String**>** s **=** Single**.**just**(**"Hello"**);**

Disposable d4 **=** s**.**subscribeWith**(new** DisposableSingleObserver**<**String**>()** **{** … **});**

Completable c **=** Completable**.**completed**();**

Disposable d5 **=** c**.**subscribeWith**(new** DisposableCompletableObserver**<**String**>()** **{** … **});**

您会从这五种类型当中得到一个返回的 Disposable，即便 Flowable 返回的有些许的不同。您可以将这些东西视为一种资源、一个文件或者数据库上一个指针。您不可能打开文件之后却没有办法关闭它。千万不要不对 Disposable 进行管理就对 Observable 进行订阅，因为最终您需要取消订阅操作。

### 操作符 [(33:52)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,122,true);)

操作符 (operator) 可以完成三种操作：

* 以某种方式操作或组合数据
* 以某种方式操作线程
* 以某种方式操作发射对象

就像我们此前必须实现的某些操作，比如说将一个同步方法调用转变为响应式，操作符就是做类似的事情。这里我们要对字符串进行操作，通过操作符让其返回一个新的字符串。

Observable**<**String**>** greeting **=** Observable**.**just**(**"Hello"**);**

Observable**<**String**>** yelling **=** greeting**.**map**(**s **->** s**.**toUppercase**());**

在响应式编程当中，我们可以创建一个 Observable，然后我们通过操作符来执行相关的操作。在本例当中，map 就是一个操作符，它允许我们获取正在发射的数据，并对其执行一些操作，从而创建一个新类型的数据。

如果我们这时候回到用户对象当中，我们最初定义了回调会回到后台线程上运行，而我们必须显式地将其移动到主线程当中。实际上这里有一个内置的操作符允许您这样做，此外还允许您用一个更加清晰的方式来做到这一点。

Observable**<**User**>** user **=** um**.**getUser**();**

Observable**<**User**>** mainThreadUser **=**user**.**observeOn**(**AndroidSchedulers**.**mainThread**());**

我们可以这么说，我想要在不同的线程上来观察这个 Observable 当中发射的对象，所以从这个用户对象当中出来的东西都需要放在后台线程当中。但是从主线程用户当中出来的东西现在都会放到主线程当中。observeOn 这里就是所谓的操作符。

由于我们需要对线程顺序进行变更，因此应用这些操作符的顺序是非常重要的。与 observeOn 类似，我们可以变更Observable 运作的位置。

OkHttpClient client **=** *// …*

Request request **=** *// …*

Observable**<**Response**>** response **=** Observable**.**fromCallable**(()** **->** **{**

**return** client**.**newCall**(**request**).**execute**();**

**});**

Observable**<**Response**>** backgroundResponse **=** response**.**subscribeOn**(**Schedulers**.**io**());**

如果我们正在执行一个网络请求，这个网络请求仍需要同步完成，但是我们不希望它在主线程上发生。那么我们可以应用一个改变我们订阅的 Observable 位置的运算符，将其变更为操作最终发生的地方。当我们订阅到后台响应的时候，那么它将变更为后台线程。I/O 是您可以使用的线程池之一，它将在该线程池上工作，然后发送通知给侦听者。subscribeOn 在这里就是变更工作地点的操作符。

看到这些都会返回一个新的 Observable 的感觉很好，因为我们可以将它们组合并链接到一起。正常情况下您可以看到，我们没有为这些创建中间变量。我们只是以一定的顺序来使用操作符。我们需要让网络请求在后台线程上执行。然后在主线程上观察到网络请求的返回结果。然后将网络回应变更为字符串。最后读取字符串。顺序在这里是非常重要的。

OkHttpClient client **=** *// …*

Request request **=** *// …*

Observable**<**Response**>** response **=** Observable**.**fromCallable**(()** **->** **{**

**return** client**.**newCall**(**request**).**execute**();**

**})**

**.**subscribeOn**(**Schedulers**.**io**())**

**.**map**(**response **->** response**.**body**().**string**())** *// Ok!*

**.**observeOn**(**AndroidSchedulers**.**mainThread**());***//*

由于我在 observeOn 之后使用了 map 操作符，因此这将在 Android 的主线程上运行。我们不想在主线程上读取 HTTP 网络回应，我们希望这个过程在我们切换到主线程之前就已经发生。先是网络请求进来，然后它在 Observable 链当中发射出回应。我们将这个回应 map 为字符串当中，最后我们将线程更改为主线程，在这里我们可以最终订阅并在 UI 当中显示这段数据。

### 操作符特殊化 [(37:28)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,150,true);)

还有其他操作符会接收 Observable 为参数，并将其返回成其他不同的类型。诸如 first() 之类的操作符会获取第一个元素，然后将其转换为字符串并返回给您。在 RxJava1 当中，我们得到的只是会发射第一个项目的一个 Observable 。这是非常奇怪的，因为如果您有一个项目列表，您调用它希望获取第一个项目，但是您得到的却是只包含一个项目的列表。您得到的是一个 scaler。在 RxJava 2 当中，当调用这个操作符时，它将保证只返回一个元素。

如果 Observable 是空的话，这将会产生一个错误，因为我们知道 single 既可以是返回一个项目，也可以是返回一个错误，所以类似 firstElement() 之类的操作符，将可能会返回一个 maybe 对象。当 Observable 是空的话，那么您就可以进入到 onComplete 阶段而不是跳转到错误处理阶段。还有一些同样也可能会返回 Completable，所以如果您不关心是否有元素返回，只关心是完成还是失败的话，那么就返回 Completable 即可，这也是 Completable 的意义。

所有的这些特性都存在于 Flowable 当中。它们都具备相同的操作符。它们同样也都会返回相同的特殊类型。这是一个显示部分操作符的图标。右上角显示的是类型阈值缩小，所以当您要调用某个东西，比如说想要计算响应流当中的元素数量，那么您就可以使用这些阈值小的类型，比如说 single。然后我们同样也有相反的操作符，也就是接受某个类型为参数并将其转换为某种更宽泛的东西。您可以把 single 转变为 Observable。

### 拥抱响应式 [(39:33)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,166,true);)

如果我们想要让我们的原始示例应用上响应式编程的话，那么我们可以订阅我们的用户，然后说「我想要在主线程上获取通知，然后将其推送给 UI，并展示该用户」。每当用户发生了变更，那么这段代码就会自动运行。您将会自动看到更新，就再也不需要担心去管理状态了。

*// onCreate*

disposables**.**add**(**um**.**getUser**()**

**.**observeOn**(**AndroidSchedulers**.**mainThread**())**

**.**subscribeWith**(new** DisposableObserver**<**User**>()** **{**

**@Override** **public** **void** **onNext(**User user**)** **{**

tv**.**setText**(**user**.**toString**());**

**}**

**@Override** **public** **void** **onComplete()** **{** */\* ignored \*/* **}**

**@Override** **public** **void** **onError(**Throwable t**)** **{** */\* crash or show \*/* **}**

**}));**

*// onDestroy*

disposables**.**dispose**();**

但是，我们必须记住要去管理所返回的 Disposable，因为我们是在 Android 上运行，当 Activity 消失之后，我们希望这段代码停止运行。在 onDestroy 中，我们会对其进行处理，将 Disposable 处置掉。

同样，当我们最终做出异步请求来更更改数据时，我们希望它在后台线程上进行。我们想要在主线程上观察结果，无论其是成功还是失败，并且在成功回调当中我们希望能够重新启用文本框输入。

disposables**.**add**(**um**.**setName**(**"Jane Doe"**)**

**.**subscribeOn**(**Schedulers**.**io**())**

**.**observeOn**(**AndroidSchedulers**.**mainThread**())**

**.**subscribeWith**(new** DisposableCompletableObserver**()** **{**

**@Override** **public** **void** **onComplete()** **{**

*// success! re-enable editing*

**}**

**@Override** **public** **void** **onError(**Throwable t**)** **{**

*// retry or show*

**}**

**}));**

再次强调一遍，因为我们不可能打开了一个文件后却不关闭它，因此如果我们订阅了某个事件后就需要管理Disposable，所以我们将其添加到 Disposable 列表当中。

我现在要跳过这个步骤了，因为我现在时间不太够了。但 RxJava 2 相比 RxJava 1 而言，好处之一便是可以进行基础的架构迁移。特别是在 Android 上，所创建的中间对象是非常少的。当我们创建响应流的时候，每个调用的操作符都必须返回一个新的 Observable 来实现该行为。当您调用了 map 之后，就会得到一个新的 Observable，也就是获取上一个Observable，然后执行一个函数，最后发射出新的数据。

这就需要对一堆中间对象进行内存分配，以便能够构建响应流。RxJava 2 改变了它的工作方式，我们所创建的中间对象变得非常少。在创建响应流的时候，内存分配也变小了不少，特别是在调用操作符的时候。每个结果所生成的对象也变少了，并且对响应流进行订阅所产生的开销也变低了。必须进行的方法调度也变少了，所以最终我们得到的是一个运行更快、GC 更少的版本，而这些没有让 API 功能得到任何的降低。

### 结论 [(42:02)](https://realm.io/cn/news/gotocph-jake-wharton-exploring-rxjava2-android/javascript:presentz.changeChapter(0,205,true);)

RxJava 2 的想法是，我们能够在 Android 中用它来处理所有异步操作，无论是网络、Android 自身的事件、数据库，还是 UI，使用它来编写代码，对这些数据源的变化做出响应，而不是试图去追逐变更和管理相关的状态。现在，RxJava 2 已经处于开发者预览版本，因此我们差不多快要完成 API 的开发了。大约一个月之后，它的最终版本将会推出，以此您就可以开始使用它，并用它来处理这些类型的操作。

**class** **RxJavaInterop** **{**

**static** **<**T**>** Flowable**<**T**>** **toV2Flowable(**rx**.**Observable**<**T**>** o**)** **{** … **}**

**static** **<**T**>** Observable**<**T**>** **toV2Observable(**rx**.**Observable**<**T**>** o**)** **{** … **}**

**static** **<**T**>** Maybe**<**T**>** **toV2Maybe(**rx**.**Single**<**T**>** s**)** **{** … **}**

**static** **<**T**>** Maybe**<**T**>** **toV2Maybe(**rx**.**Completable c**)** **{** … **}**

**static** **<**T**>** Single**<**T**>** **toV2Single(**rx**.**Single**<**T**>** s**)** **{** … **}**

**static** Completable **toV2Completable(**rx**.**Completable c**)** **{** … **}**

**static** **<**T**>** rx**.**Observable**<**T**>** **toV1Observable(**Publisher**<**T**>** p**)** **{** … **}**

**static** **<**T**>** rx**.**Observable**<**T**>** **toV1Observable(**Observable**<**T**>** o**,** …**)** **{** … **}**

**static** **<**T**>** rx**.**Single**<**T**>** **toV1Single(**Single**<**T**>** o**)** **{** … **}**

**static** **<**T**>** rx**.**Single**<**T**>** **toV1Single(**Maybe**<**T**>** m**)** **{** … **}**

**static** rx**.**Completable **toV1Completable(**Completable c**)** **{** … **}**

**static** rx**.**Completable **toV1Completable(**Maybe**<**T**>** m**)** **{** … **}}**

如果您还在使用 RxJava 1 的话，这个转换项目将允许您能够在不同的类型之间相互转换，此外它还允许您对应用进行增量更新。

github**.**com**/**akarnokd**/**RxJava2Interop

您的依赖文件最终看起来会像这样：

dependencies **{**

compile 'io**.**reactivex**.**rxjava2**:rxjava:**2.0**.**0**-**RC3'

compile 'io**.**reactivex**.**rxjava2**:rxandroid:**2.0**.**0**-**RC1'

*// Optionally...*

compile 'com**.**github**.**akarnokd**:**rxjava2**-interop:**0.3**.**0'**}**

RxJava 2 并不是一个新鲜玩意儿。响应式编程也是一样的，不过 Android 本身就是一个高度响应式的平台，我们此前已经习惯了强制的状态管理方式来构建应用了。

响应式编程允许我们用更合适的方式来构建应用，从而更适合异步操作进行。我们需要接纳数据源的异步特性，而不是试图去管理所有的状态，我们应该将异步源和响应式编程结合在一起，共同让我们的应用进入到响应式的世界当中。