Airbnb：我们的安卓客户端是如何使用 RxJava 的

realm.io

介绍 (0:00)

在这篇文章里，我会讨论 RxJava 和 Airbnb 的客户端里运用它的方法，我在 Airbnb 工作了一年多一点的时间。给你一个关于规模的感觉，我们一共有15个人。公司现在在快速扩张中，而且对于一个小组来说，集成新的技术都是非常具有挑战性的。

为什么用 RxJava? (0:47)

我们都知道 **移动开发是困难的**。移动用户期望即时响应，而且还有在不同的线程间来回切换的需求。除了主线程，你还要做网络连接，同时你还需要在后台处理其他的各种不同的事情。最重要的是，你不能阻塞 UI 线程。

RxJava 是解决这类问题的好方法，因为他能够使得线程间的切换比较容易。这已经集成在框架里面了。异步操作非常笨重而且容易出错，RxJava 使得你不用再这样做了，这也是你能把不同的线程组合在一起的原因。

我们需要 RxJava 的真正原因其实是 **我们的软件很烂**。为什么我们有如此多的 bug？为什么我们需要 crash 报告工具来跟踪我们成千上万的 crash，或者多少用户已经对我们生气了？这里可能有些什么事情不对劲。

我们需要改变；我觉得 imperative 编程是我们不应该采用的方法。当然，面向对象编程已经流行很多年了。它已经深入到了现代程序员的骨髓里了。每个人都盲目的使用它，但是它不是我们开发软件的必需品。

Functional 编程是 RxJava 里面的概念，而且我觉得用这种方法，代码更加健壮，而且永远不需要维护状态了。代码更加可靠而且你知道它一定工作。

底线：我们的问题是我们写了很多糟糕的代码，移动开发也是很困难的，而 RxJava 是解决这个问题的一个方案。

流 (3:48)

RxJava 是 ReactiveX 的一部分，一组开源库。它们有许多不同的库，包括 JavaScript，Groovy， Ruby， Java， C#，以及其他。然而，它们都有着同样的概念，这就是 functional 编程。

流是这个概念的核心部分。你的代码里面的所有东西都是“流”，而且你需要重新构建对它的认知。今天，我们认为代码是顺序执行的，因为它也是这样被编写的。你写一个指令，然后另外一个，然后你有一个循环，然后你调用一个方法，然后你返回。你把它们加入到不同的线程里面去，然后你有了并行处理。你总是需要考虑如果一个线程返回了怎么办，而此时你已经在代码的别的什么地方了。这非常难，特别是针对移动开发。

学习流的概念对我来说确实意味着些不同。我六个月前开始学习它，那时我刚开始在 Airbnb 里面使用 RxJava，它实在是太复杂了。我第一次看到它的时候，我刚看了一页，就迷失在不同的概念里面了。Observables， observers，太多了。然后，慢慢地熟悉起来，你才开始理解它的一些概念了，然后它变得有意义了很多。核心思想是 **一切皆是流**。

太多的概念 (5:54)

我们都同意 reactive 编程是困难的，但是它却越来越流行。我可以看到这样的趋势：React Native， React，还有其他的不同的库都涌现出来。还有些新的工具，例如 Cycle， Elm，和专注 Reactive 的其他语言的工具。

然后，的确是有太多的概念需要理解了！就我看来，有两个主要的概念：observable 和 observer，这也是你知道的两个主要类。然后，还有 subscriber， subscription， producer，hot、cold observables， backpressure， scheduler， subject，和更多。这只是 10%。它是一个庞然大物！

如果你感觉有些混淆，没关系，我很理解你。保持学习，保持实验，然后所有的事情最终都会有意义的。（我不是吓唬你，但这就是事实。）

拥抱挑战 (7:15)

我想我可以介绍下在 Airbnb 我们是怎么做的：我们的过程，我们学到的东西，好的经验和坏的教训。

团队规模 - 我们团队有 15 个人，这是个很大的团队了，也是我工作过的最大的团队。每个人都在不断的给同一个 repo（我们的安卓应用）提交代码。 我们只有一个代码库，而且同事们之间互相审核代码。每个人都了解发生了些什么而且每个人都知道你写的代码是什么意思，这点很重要。

我们使用 Phabricator 来审核代码，这个和 GitHub 上的 pull request 很类似。你撰写意见，提

出建议，给出反馈，还有其他的流程。在你决定使用 RxJava 之前，让每个人都能认同是很重要的。如果你开始使用它了，团队里的其他人还不知道要发生些什么，而且不知道背后的原因，那么进程将是十分困难的。如果你只有两个人，这没关系，但是当你有更多的人加入团队的时候，让每个人都全速前进将会是非常有挑战的事情。

学习曲线 - 你需要理解你常常会犯一些很低级的错误。你可能会写出没有任何意义的代码，你可能会导致产品 crash，但是所有的事情最终都会好起来的。在我的经验里，每个人大概花了两个月的时间来理解 RxJava。我建议团队一起讨论它，然后尽量给团队里面的其他成员解释这些概念，如果你计划采纳这项技术的话。你自己尽量先能有个好的理解，然后当你感觉到成熟了，把所有人叫到屋里讨论它。实战：打开 Android Studio 然后演示一些代码。

调试 - 这会是个大问题。每个社区里的人都知道这个问题，他们也知道这是需要改进的地方。我最近在我们的 bug 系统里面收到一个有着很多异常的堆栈日志。这是很复杂的事情，而且有很多干扰信息。我不知道是不是有人已经在积极地解决这个问题了，所以，如果有初创公司想找个可以练手的东西的话，这是个你可以开始的地方。

常见的陷阱 (11:34)

我想指出一些我们常见的拦路虎。在使用 RxJava 的过程中它们都是大麻烦。

observeOn() (11:49)

如果你想使用 RxJava，你需要知道以下的重要核心概念。

```
return observableFactory.<T>toObservable(this)
        .compose(this.<T>transform(observableRequest))
        .observeOn(Schedulers.io())
        .map(new ResponseMetadataOperator<>(this))
        .flatMap(this::mapResponse)
        .observeOn(AndroidSchedulers.mainThread())
        .<AirResponse<T>>compose(group.transform(tag))
        .doOnError(new ErrorLoggingAction(request))
        .doOnError(NetworkUtil::checkForExpiredToken)
        .subscribe(request.observer());
```

这是我们的应用中创建 RxJava observable 流的时候的一段代码。我们调用 observeOn 两次，看起来好像无意义。实际上，你每次调用 observeOn，后面的代码都会运行在那个 scheduler 上，然后你之后又调用一遍，它就又切换一次。

当我们使用 RxJava 的时候，你创建了一个流。一个关于 RxJava 的误解就是它是异步的，但

是事实上每件事情都是默认同步的。当你创建一个流的时候，你仅仅是创建了一个点，这里我们会向它订阅。当你订阅的时候，你把所有的东西都才创建在一起了，然后才能执行它。在你调用 subscribe 之前，你仅仅是创建了一个流。这比较类似声明的流程。当你说 observeOn，你切换到另外一个线程。如果你不调用 observeOn，每件事都还是在原来那个需要订阅给 observable 的线程里。这里我们有 subscribe，所以如果这是从主线程中调用的话，主线程里的所有事情都会发生而不论你做了些什么。所以 observeOn 是一个有效的调用其他线程工作的方法，而且它会使过程异步化。

我们第一次调用 observeOn 的时候，我们传入了一个 scheduler。RxJava 有一些内嵌的 scheduler，其中一个就是 I/O scheduler，这当然是和 I/O 线程工作在一起的，I/O 线程是一个和你的 I/O 绑定的线程池。map 和 flatMap 操作符在那个线程里面执行，然后当它结束的时候，我们把它发送回到主线程。所以，你正在主线程里面工作，假设这是从主线程里面调用的，然后加载到后台线程， 最后把它移回到主线程。

如果你不使用 RxJava，这会是个非常复杂的事情。然而，现在我们有这么简单的描述性的方法来实现你想做的事情。这也是为什么 RxJava 会很复杂的原因：这么少的代码，但是却要花很长的时间来真正理解里面发生了些什么。

subscribeOn() (16:14)

另外一个和 observeOn 联系紧密的概念即使 subscribeOn。这会改变 observable 订阅了的那个线程，如果你对这些概念不熟的话，听起来会觉得很复杂。

```
return observableFactory.<T>toObservable(this)
        .compose(this.<T>transform(observableRequest))
        .observeOn(Schedulers.io())
        .map(new ResponseMetadataOperator<>(this))
        .flatMap(this::mapResponse)
        .observeOn(AndroidSchedulers.mainThread())
        .<AirResponse<T>>compose(group.transform(tag))
        .doOnError(new ErrorLoggingAction(request))
        .doOnError(NetworkUtil::checkForExpiredToken)
        .subscribeOn(Schedulers.io())
        .subscribe(request.observer());
```

第一个调用，observableFactory.<T>toObservable 是 observable 对象创建的地方，这也是直接受 subscribeOn 影响的代码。还有一些运行 subscription 的代码，当你需要向它订阅的时候，你也有些代码需要运行。然后你才会有那个流上的其他变化。当执行到 subscription 的代码的时候，这些才会变化，而不是其他的地方。你什么时候调用它没有关系，它只在 subscription 被执行的时候线程才会改变。

错误处理 (18:05)

```
return observableFactory.<T>toObservable(this)
        .compose(this.<T>transform(observableRequest))
        .observeOn(Schedulers.io())
        .map(new ResponseMetadataOperator<>(this))
        .flatMap(this::mapResponse)
        .observeOn(AndroidSchedulers.mainThread())
        .<AirResponse<T>>compose(group.transform(tag))
        .doOnError(new ErrorLoggingAction(request))
        .doOnError(NetworkUtil::checkForExpiredToken)
        .subscribeOn(Schedulers.io())
        .subscribe(request.observer());
```

我们使用 doOnError 作为错误日志的一个方法。你的网络出现异常了，然后你想给你的分析服务注入日志，你想知道这种情况发生了多少次。doOnError 是个每次你在流上出现错误都会被执行的动作，然后你会有多次调用，所以你有对于一个流的多次错误处理。当它看见一个错误事件的时候，它就会调用它的方法，但是这是个副作用。

```
return observableRequest
        .rawRequest()
        .<Observable<Response<T>>>newCall()
        .observeOn(Schedulers.io())
        .unsubscribeOn(Schedulers.io())
        .flatMap(responseMapper(airRequest))
        .onErrorResumeNext(errorMapper(airRequest));
```

另一个可以被使用的结构是 onErrorResumeNext，这个工作起来像是个 catch 块，这在 reactive 世界里就是个没有意义的事情。这就好像你再说，“Hey，当我看到一个错误的时候，我想运行这个动作来扑捉这个错误，然后继续执行，然后打包那个异常，写个日志，然后返回个空的数据集合或者其他什么东西。”如果你还是 imperative 思维，这就像个 catch 块。

单元测试 (20:04)

给同步的流数据做单元测试听起来很复杂，所以 RxJava 提供了这个漂亮的类叫做 TestSubscriber。

```
@Test public void testErrorResponseNonJSON() {
 server.enqueue(new MockResponse()
        .setBody("something bad happened")
```

```
        .setResponseCode(500));
 TestRequest request = new TestRequest.Builder<String>().build();
 TestSubscriber<AirResponse<String>> subscriber = new TestSubscriber<>();
 observableFactory.<String>toObservable(request).subscribe(subscriber);
 subscriber.awaitTerminalEvent(3L, TimeUnit.SECONDS);
 NetworkException exception = (NetworkException)
        subscriber.getOnErrorEvents().get(0);
 assertThat(exception.errorResponse(), equalTo(null));
 assertThat(exception.bodyString(), equalTo("something bad happened"));
}
```

你可以使用 TestSubscriber 来订阅你的流，然后你可以阻塞它，直到它获得了一个事件。有一些简便的方法，例如 .awaitTerminalEvent，这也会阻塞你的线程直到一个终端事件（例如：onCompleted 或者 onError）。对于你的流中的每个事件，你可以得到 0 次到 n 次的 onNext 事件，然后当它结束的时候，你获得 onCompleted 或者它失败了，你获得 onError，之后你再也收不到任何时间了，流也结束了。

```
@Test public void testUnicodeHeader() {
 server.enqueue(new MockResponse().setBody("\"Hello World\""));
 TestRequest request = new TestRequest.Builder<String>()
 .header("Bogus", "中華電信")
 .build();
 observableFactory.toObservable(request)
 .toBlocking()
 .first();
 RecordedRequest recordedRequest = server.takeRequest();
 assertThat(recordedRequest.getHeader("Bogus"), equalTo("????"));
}
```

另一件事是你可以使用 toBlocking。这会立即阻塞线程，这在单元测试里面十分有用。当然，作为产品代码用处不大。如果你使用 RxJava，你不太可能会阻塞你自己线程，但是在测试的时候就非常方便了。这会比使用测试 subscriber 代码量少点。如果你知道不会失败，你可以直接阻塞然后获得第一个事件。

在这个例子里面，我们使用了 OkHttp 来 mock 响应，然后发送假的响应回去。我们增加了一个前缀，然后测试一些前缀的特殊字符，如果这是个 OkHttp 的 bug 的话。然后我们正确清理前缀，就可以正常测试了。

内存泄漏 (22:41)

如果你做移动开发，你知道内存异常的所有情况。

当你向一个流订阅的时候，你得到了一个 subscription。当你得到这个 subscription 之后，你可以注销它，所以你需要显示地释放资源。你不在需要引用那个流了。我们都知道发起请求的重要性，例如，从安卓的 activity 或者 fragment 中发起请求。你不要忘记了，你想在 acitivity 销毁的时候释放这些资源，这是一个常见的模式。

```
private final CompositeSubscription pendingSubscriptions =
        new CompositeSubscription();

@Override public void onCreate() {
  pendingSubscriptions.add(
          observable.subscribe(observer));
}

@Override public void onDestroy() {
  pendingSubscriptions.clear();
}
```

你可以使用 CompositeSubscription，这是个能够集合多个订阅的类，而且你给它增加一个订阅。然后，一旦你销毁 activity,你就能清除它了。

附加资源 (23:55)

问&答 (23:35)

问：当有错误的时候你们重试的机制是怎样的？例如向服务器发起的取数据的网络请求。

Felipe: 在我们的实际中，我们没有重试机制；我们就是失败。但是如果你想使用它，一个方法就是使用 onErrorResumeNext。你可以继续另一个请求，然后处理多次。如果你获得一个错误，继续两个不同的请求然后你能实现两次，这样你可以在你需要的时候重试两次。这是一个方法。我没有真正地用 RxJava 实现它，所以我没有一个特别的建议。

问：在 Java 8 中，lambdas 不需要拥有一个它被创建的上下文的引用，但是在安卓里面，我认为他们需要？安卓没有真正实现 Java 8。我很好奇你们在处理交还引用中的关于内存泄漏的经验。

Felipe: 据我所知，关于内存管理，我们没有遇到过 lambdas 的问题。这很令人吃惊。我们使用 Retrolambda 这是个 Java 8 的 lambdas 到 Java 7 字节流的移植，因为安卓不能支持 Java 8。把字节流改成和 Java 7 兼容是种 hack 的方法。我没有遇见任何问题，我的 lambdas 代码都运行正常。

问：我不知道你们有没有跟踪 RxJava repo，但是 Ben Christensen 十月份离开去为 Facebook 工作了，自从那以后，项目就进入了无人问津的状态。现在它被另外一个人维护着。你们有这方面的担心吗，把 Rxjava 框架作为开发平台的长期可行性？

Felipe: 这是事实。据我所知，Netflix 承诺分配人来做这件事，但是看起来还没有开始。David Karnok，独自维护这个产品的人，真的非常非常聪明。事实是，RxJava 1.0 就很稳定了。我不认为我会担心未来的 bug 修正或者安全性或者其他类似的问题。未来几年都不会有问题。关于未来的开发工作，比如 2.0，这是未知的事情。我们不知道会发生什么。我对 1.0 很满意，因为它非常稳定。

问：你能说明下拆分流的技术吗？现在有许多合并流的资料，我能找到的唯一一个关于分拆的技术是 publish-connect 模型，而且我感觉这更适合增加多个订阅而不是拆分 observable 为多个可观察的流。

Felipe: 正确，所以以前我使用 share，这是使得一个 observable 被多人订阅的方法，正如我说的那样。我不记得 share 和 publish-connect 具体的差别了，它们本质上是同样的事情。我们用它来允许多人订阅同一个 observable，但是我真不知道我们能这样做的其他的方法。

问：你提到一个难点是 hot 和 cold observables。你有相关的材料或者在你的代码里面的命名来解释下谁是谁吗，就像，“哦，惊喜！你的订阅有许多副作用！”

Felipe: 这是你需要理解的又一个概念。我们不是真有这样的问题因为大部分我们使用 RxJava 的情况是围绕 Retrofit 的，这也是非常直接的。你订阅它，然后你发起一个网络请求。我不是真的有太多的关于它的问题，因为我也没有接触那么多。但是当 RxJava repo 有一个好的 Wiki，他们有许多很棒的文档，而且 ReactiveX 网站也有许多好东西。

问：当你决定转向 RxJava 的时候，你决定把整个代码库里的代码都迁移过去。你能谈些 Airbnb 在这期间的挑战吗？这仅仅是人的问题，或者是别的什么？

Felipe: 我会说大部分是人的问题，包括我，因为我也是在学习。当你还在学习的时候，你还要教会别人这个工具，这是个问题。解释有些概念实在太难了，因为它们是如此的复杂。在我看来最大的挑战是让每个人都能全速前进。我们没有许多的产品问题。我们有一些 crash 而且是本地使用的原因，我记不太清了。我们有些继承的网络部分的建立是使用的 Volley，然后我们转向了 Retrofit。我们有许多围绕着那周边的继承的代码，像请求类，而且它还不是真正地符合 Retrofit 工作的方式的，所以我们实际上是 fork 了 Retrofit 然后让它在我们当前的设置里工作的。然而，大部分挑战是让每一个人都理解它。这需要时间，你不能一撮而就。

问：你提到 RxJava 是困难的。现在回过头去看你们跨越的旅程，你认为，为了你提到的这些优点而做的妥协是值得的吗？

Felipe: 我会认为这是值得投资的。这当然是个大投资。我们需要花费大量的时间学习，然后

我认为这一定是值得的。一旦你使用的越来越多，你得到的好处也就越多。当你开始结合流而且重新赋予他们意义的时候，你会看着而且想，“嘿，我用一小段代码实现了许多的事情。”当你到达这个阶段的时候，这是个特别神奇的事情。所以我觉得认为这是值得的。

问：你在所有的事情中都用了 RxJava 吗，或者你们有一个分界线，只有在某种复杂的异步任务中你们才会采用 RxJavava?

Flipe: 我们呢，只在我们的网络层使用它。我们没有给 UI 暴露这些细节，因为那会是个更大的赌注。让每一个人都理解，然后知识真正被运用，因为如果你给你的 activity 层或者你的 view 暴露它，这就意味着所有人都需要理解它，运用它。在网络层，只有少数的同事接触它，所以你不需要所有人都理解它。我们只暴露 API，然后它就工作了。这就好比一座桥，我们在这端用 RxJava，然后另外一端是普通的大陆。这是有局限性的，虽然它是我们应用的核心。