反应性代码的最基本单元是被观察对象和订阅者。一个被观察对象放出数据，一个订阅者消费这些数据。关于数据的释放有一种模式。一个被观察对象可能释放任意个数的数据（包括零个），然后它会在成功完成后终止，或者由于遇到错误而终止。对每个订阅者，被观察对象会调用Subscriber.onNext()很多次，然后调用Subscriber.onComplete()或Subscriber.onError()。
这看上去非常像标准的观察者模式，但它有一个关键的不同--被观察对象在没有被显式订阅之前是不会开始释放数据的。换句话说：如果没有人在收听，它是不会说话的。

一 被观察者和订阅者

现在来看一个例子，创建了一个基本的被观察对象：

```
Observable<String> myObservable = Observable.create(
    new Observable.OnSubscribe<String>() {
        @Override
        public void call(Subscriber<? super String> sub) {
            sub.onNext("Hello, world!");
            sub.onCompleted();
        }
    }
);
```

这里myObservable释放了一个“Hello, world!”的字符串，接下来需要创建一个订阅者来消费这个字符串：

```
Subscriber<String> mySubscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) { System.out.println(s); }

    @Override
    public void onCompleted() { }

    @Override
    public void onError(Throwable e) { }
};
```

这样就可以打印出每个释放出来的字符串了。最后我们只需要将这两者连接起来：

```
myObservable.subscribe(mySubscriber);
// 输出结果 "Hello, world!"
```

以上就是标准的使用模板了。但这个例子中很多代码都可以更加简化。

首先，Observable可以得到简化：

```
Observable<String> myObservable = Observable.just("Hello, world!");
```

这里的Observable.just()只会释放一条数据，然后完成终止。
然后，在这个例子中我们并不关心onCompleted()和onError()，代替的作法就是使用一个简单的类来定义在onNext()里做什么：

```
Action1<String> onNextAction = new Action1<String>() {
    @Override
    public void call(String s) {
        System.out.println(s);
    }
};
```

Action可以定义Subscriber的每个回调。Observable.subscribe()的参数可以有一、二、三个，去代替onNext()、onError()和onComplete()，就像这样：

```
myObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);
```

而在这个例子中我们只关心onNext()，因此只需要：

```
myObservable.subscribe(onNextAction);
// 输出 "Hello, world!"
```

这些代码放在一起就变成了：

```
Observable.just("Hello, world!")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            System.out.println(s);
```

```
        }
    });
```

二 操作者
假如我们需要在字符串后面加上一个字符串作为个人标记，可以如下修改：

```
Observable.just("Hello, world! - Dan")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            System.out.println(s);
        }
    });
```

此处是修改的被观察者，但如果我在很多地方都用到了这个观察者，但只是有时才会加上这段字符串该怎么做呢？

```
Observable.just("Hello, world!")
    .subscribe(new Action1<String>() {
        @Override
        public void call(String s) {
            System.out.println(s + "-Dan");
        }
    });
```

此处是修改的订阅者，但有时候订阅者是运行在主线程里，不适合做类似的操作，从概念上说，订阅者应当对事件作出反应，而不是变化。如果我可以在某个中间步骤对这个字符串作处理呢？这里就轮到Operators操作符出场了，它是在观察者和最终订阅者之间被用来操

作被释放的数据。
这里使用map在原字符串结尾加上该字符串：

```
Observable.just("Hello, world!")
    .map(new Func1<String, String>() {
        @Override
        public String call(String s) {
            return s + " -Dan";
        }
    })
    .subscribe(s -> System.out.println(s));
```

这里需要注意的一点是，Func1<String, String>尖括号内的参数，第一个String代表传入该map的对象类型是字符类型，第二个String代表从map传出的，即回调call的返回值类型也是字符类型。从这里其实就可以看出来，我们可以对传入的数据进行类型的转换，比如

String到integer：

```
Observable.just("Hello, world!")
    .map(new Func1<String, Integer>() {
        @Override
        public Integer call(String s) {
            return s.hashCode();
        }
    })
    .subscribe(i -> System.out.println(Integer.toString(i)));
```

可以看到数据到订阅者那里时已经变成了Integer类型了，更需要说明的一点是自定义model类型的对象也是可以通过它来进行转换的。

需要注意的点：
1 最小的单元实际上是Oberver观察者，但实际使用当中使用得最多的是Subscriber订阅者，因为需要通过它连接被观察者对象。
2 虽然这一点的理解影响并不大但还是需要了解，即热被观察者和冷被观察者，热被观察者指无论是否有订阅者都会释放数据，相对的冷被观察者如先前所说那样，如果没有订阅者它是不会开始释放数据的。
首先需要介绍Observable.from()，它的作用是传入一个数据集合执行多次直到集合全部释放出来，例如：

```
Observable.from(urls).subscribe(url -> System.out.println(url));
```

然后，介绍Observable.flatMap()，

```
// Returns a List of website URLs based on a text search
Observable<List<String>> query(String text);
```

```
query("Hello, world!")
    .flatMap(new Func1<List<String>, Observable<String>>() {
        @Override
        public Observable<String> call(List<String> urls) {
            return Observable.from(urls);
        }
    })
    .subscribe(url -> System.out.println(url));
```

这里需要说明的是，传入flatMap的是第一个参数List<String>，而返回的值是Observable<String>，利用Observable.from(urls)的

意思就是将url集合分别分次的返回，相当于会返回多次Observable<String>直到url集合全部返回。flatMap()可以返回任意

Observable。
假如有这样一个方法：

```
// 返回网站标题，如果404就返回空
Observable<String> getTitle(String URL);
```

现在不打印url了，而是打印每个接受到的网站标题。

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(new Func1<String, Observable<String>>() {
        @Override
        public Observable<String> call(String url) {
            return getTitle(url);
        }
    })
    .subscribe(title -> System.out.println(title));
```

现在需要把404的情况过滤掉，即不能显示为空：

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .filter(title -> title != null)
    .subscribe(title -> System.out.println(title));
```

filter()只会在通过它内部的布尔值检查才会释放接受到的数据，否则不会释放出来。

现在我们最多只显示5个结果：

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .filter(title -> title != null)
.take(5)
    .subscribe(title -> System.out.println(title));
```

take()最多释放指定的数量。（如果少于5个标题，它很早就会结束）

现在我们希望将标题保存在磁盘上：

```
query("Hello, world!")
    .flatMap(urls -> Observable.from(urls))
    .flatMap(url -> getTitle(url))
    .filter(title -> title != null)
    .take(5)
    .doOnNext(title -> saveTitle(title))
    .subscribe(title -> System.out.println(title));
```

doOnNext()允许我们在每次释放一项数据的时候增添额外的行为。

在第一章中，我们了解了RxJava的基本框架，第二章中我们知道操作符可以变得如何强大。但也许你可能仍然有些迟疑，要让你信服还远远不够。那么本章将会介绍一

些有关RxJava的其他优点。

Error Handling

在此之前我们已经大大忽略了onComplete()和IonError()。它们在Observable停止释放数据时被调用，无论是成功完成还是失败出错。而Subscriber可以监听到这两个方法：

```
Observable.just("Hello, world!")
    .map(s -> potentialException(s))
    .map(s -> anotherPotentialException(s))
    .subscribe(new Subscriber<String>() {
        @Override
        public void onNext(String s) { System.out.println(s); }

        @Override
        public void onCompleted() { System.out.println("Completed!"); }

        @Override
        public void onError(Throwable e) { System.out.println("Ouch!"); }
    });
```

假设potentialException()和anotherPotentialException()都由可能抛出异常，而Observable将会在onComplete()或onError()结束释放。那么这一段代码要么会输出Completed，要么会输出Ouch。这里有一些需要注意的地方：

1 在整个数据流的任何地方一旦抛出异常，onError()方法就会被调用。这会令异常处理变得轻松多了，我完全可以在每个方法后面添加异常处理方法。

2 操作符可以不用处理异常。你可以将异常交给Subscriber，让它决定如何处理。

3 你明确知道Subscriber会在何时结束接收数据项。

了解这些会使错误处理比以前更加轻松，因为如果用回调你不得不在每个回调里都加上错误处理方法，这不仅仅会带来重复代码也意味着每个回调必须明确怎样处理错误，也就表示你的回调代码必须紧紧依附于调用者。

用RxJava，你的Observable甚至都不用知道遇到异常错误该如何处理。你的其他任何操作符都不用处理错误，他们会在出现严重错误时直接跳过。你可以把你所有的错误处理都交给Subscriber。

Schedulers

如果你的Android应用有网络请求的话，那么你不得不将这个耗时的操作放在其他线程上，但在Android上多线程处理是很困难的，因为你必须保证在正确的线程上运行正确的代码，一旦有误程序就会崩溃。一个经典的异常就会在你尝试修改主线程上的View的时候抛出来。

使用RxJava，subscribeOn()用来决定你的observer代码在哪个线程运行，observeOn()用来决定你的Subscriber会在哪个线程运行(下面的代码为例，意思是根据url请求图片会在其他线程执行，而bitmap获取下来之后会在主线程上执行设置图片，简而言之，请求操作在subscribeOn内定义在哪条线程执行，当结果来了之后会根据observeOn内的设置来决定subscribe内的操作会在哪条线程上执行)。

```
myObservableServices.retrieveImage(url)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(bitmap -> myImageView.setImageBitmap(bitmap));
```

这里需要注意的地方是**subscribeOn**内的设置，可以设置很多种线程，但常用的有三种，源码如下：

```java
private final Scheduler computationScheduler;
private final Scheduler ioScheduler;
private final Scheduler newThreadScheduler;
```

那么该这三种的不同在于，第一种会有一个固定大小的线程池，线程池的大小等于有效processor个数，然后再使用最近最少使用原则来选择worker。第二种和第三种每次请求都会新起一个线程，而两者区别仅仅是前者有缓存机制后者没有。

Subscriptions

有一件事情我一直没有提到，当调用Observable.subscribe()它会返回一个Subscription。它代表了你的Observable和subscriber的连接：

```
Subscription subscription = Observable.just("Hello, World!")
    .subscribe(s -> System.out.println(s));
```

换句话说，你可以在适当的时候得到这个连接进行绑定或解绑：

```
subscription.unsubscribe();
System.out.println("Unsubscribed=" + subscription.isUnsubscribed());
// Outputs "Unsubscribed=true"
```

关于RxJava很好的一个地方就是解绑意味着停止整条链，无论什么时候正在做什么，一旦解绑就会立即终止，仅此而已。这里需要解释的是，这个数据链的理解，你可以把整个数据链从开头到结尾完全连接起来去思考，有入口也有出口，数据从某个操作符进入后，中间经过了多个操作符的处理方法最终返回出口，接下来就分别发送给订阅者，即subscribe()的内容。一般的操作符都会返回Observable<T>类型，而这里的Subscription是最终出口subscribe的返回类型，因此一旦拿到它便可以随时随地进行进行数据链的控制。

Conclusion

Keep in mind that these articles are an introduction to RxJava. There's a lot more to learn than what I presented and it's not all sunshine and daisies (for example, read up on backpressure). Nor would I use reactive code for everything - I reserve it for the more complex parts of the code that I want to break into simpler logic.

Originally, I had planned for this post to be the conclusion of the series, but a common request has been for some practical RxJava examples in Android, so you can now continue onwards to part 4. I hope that this introduction is enough to get you started on a fun framework. If you want to learn more, I suggest reading the official RxJava wiki. And remember: the infinite is possible.

Many thanks to all the people who took the time to proofread these articles: Matthias Käppler, Matthew Wear, Ulysses Popple, Hamid Palo and Joel Drotos (worth the click for the beard alone).

---

[1] This is one reason why I try to keep my `Subscriber` as lightweight as possible; I want to minimize how much I block the main thread.

[2] Deferring calls to `observeOn()` and `subscribeOn()` is good practice because it gives the `Subscriber` more flexibility to

handle processing as it wants. For instance, an `Observable` might take a while, but if the `Subscriber` is already in an I/O thread you wouldn't need to observe it on a new thread.

[3] In part 1 I noted that `Observable.just()` is a little more complex than just calling `onNext()` and `onComplete()`. The reason is subscriptions; it actually checks if the `Subscriber` is still subscribed before calling `onNext()`.

前面三章主要讲解了RxJava的部分内容，而作为Android开发者就不得不提到将RxJava和Android结合起来应用的框架---RxAndroid。

RxAndroid是RxJava的扩展，包含了一些针对Android的特殊绑定。

1 AndroidSchedulers 它为Android线程机制提供了现成的schedulers。如果希望在UI线程上执行一些代码只管用AndroidSchedulers.mainThread()。

```java
retrofitService.getImage(url)
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(bitmap -> myImageView.setImageBitmap(bitmap));
```

2 AndroidObservable 它提供了在Android生命周期内的一些功能。比如bindActivity() 和 bindFragment()，它们会自动让观察者在AndroidSchedulers.mainThread()主线程执行，并且会在离开activity或fragment时自动停止释放数据。

```java
AndroidObservable.bindActivity(this, retrofitService.getImage(url))
    .subscribeOn(Schedulers.io())
    .subscribe(bitmap -> myImageView.setImageBitmap(bitmap);
```

3 AndroidObservable.fromBroadcast() 它允许你创建一个像BroadcastReceiver一样工作的Observable。

```java
IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY_ACTION);
AndroidObservable.fromBroadcast(context, filter)
    .subscribe(intent -> handleConnectivityChange(intent));
```

4 ViewObservable 它针对于View，监听点击事件可以用ViewObservable.clicks()，监听TextView的内容变化可以用ViewObservable.text()。

```java
ViewObservable.clicks(mCardNameEditText, false)
    .subscribe(view -> handleClick(view));
```

Retrofit

回调使用：

```java
```

```java
@GET("/user/{id}/photo")
void getUserPhoto(@Path("id") int id, Callback<Photo> cb);
```

Rx使用：

```java
@GET("/user/{id}/photo")
Observable<Photo> getUserPhoto(@Path("id") int id);
```

如果我们第一个请求是获取图片，第二个请求是获取图片信息，那么如今我们可以将这两次请求进行合并：

```java
Observable.zip(
    service.getUserPhoto(id),
    service.getPhotoMetadata(id),
    (photo, metadata) -> createPhotoWithData(photo, metadata))
    .subscribe(photoWithData -> showPhoto(photoWithData));
```

生命周期

我把难点留到了最后，即如何处理activity的生命周期。有两个老生常谈的问题：

1 在configuration改变的时候持续Subscription（比如屏幕翻转）。

2 因Observable内持有context引用引发的内存泄漏问题。

尽管没有标准答案，但有些思路可以参考。

第一个问题，缓存一份请求，在activity重建后再次被订阅，而在此之前需要先解绑上一次被订阅的观察者：

```java
Observable<Photo> request = service.getUserPhoto(id).cache();
Subscription sub = request.subscribe(photo -> handleUserPhoto(photo));

// ...When the Activity is being recreated...
sub.unsubscribe();

// ...Once the Activity is recreated...
request.subscribe(photo -> handleUserPhoto(photo));
```

第二个问题，重点在于管理subscription，即合适的解绑订阅，那么通常的做法就是CompositeSubscription，添加进所有的subscription然后在适当的时候一次性解绑：

```java
private CompositeSubscription mCompositeSubscription
    = new CompositeSubscription();

private void doSomething() {
    mCompositeSubscription.add(
        AndroidObservable.bindActivity(this, Observable.just("Hello, World!"))
        .subscribe(s -> System.out.println(s)));
}

@Override
protected void onDestroy() {
    super.onDestroy();

```

```
14.         mCompositeSubscription.unsubscribe();
15.     }
```

当然这个操作可以放到基类去处理，子类只需要继承就可以了，但是要注意的是一旦解绑之后，要重用时就必须要新建一个CompositeSubscription。

基础教学到此结束，重点在于数据链和操作符的理解。