[深入浅出RxJava（一：基础篇）](http://blog.csdn.net/lzyzsd/article/details/41833541)

## 基础

RxJava最核心的两个东西是Observables（被观察者，事件源）和Subscribers（观察者）。Observables发出一系列事件，Subscribers处理这些事件。这里的事件可以是任何你感兴趣的东西（触摸事件，web接口调用返回的数据。。。）  
  
一个Observable可以发出零个或者多个事件，知道结束或者出错。每发出一个事件，就会调用它的Subscriber的onNext方法，最后调用Subscriber.onNext()或者Subscriber.onError()结束。

Rxjava的看起来很想设计模式中的观察者模式，但是有一点明显不同，那就是如果一个Observerble没有任何的的Subscriber，那么这个Observable是不会发出任何事件的。

# Hello World

创建一个Observable对象很简单，直接调用Observable.create即可

Observable<String> myObservable = Observable.create(

**new** Observable.OnSubscribe<String>() {

        @Override

**public** **void** call(Subscriber<? **super** String> sub) {

            sub.onNext("Hello, world!");

            sub.onCompleted();

        }

    }

);

这里定义的Observable对象仅仅发出一个Hello World字符串，然后就结束了。接着我们创建一个Subscriber来处理Observable对象发出的字符串。

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) { }

};

这里subscriber仅仅就是打印observable发出的字符串。通过subscribe函数就可以将我们定义的myObservable对象和mySubscriber对象关联起来，这样就完成了subscriber对observable的订阅。

myObservable.subscribe(mySubscriber);

一旦mySubscriber订阅了myObservable，myObservable就是调用mySubscriber对象的onNext和onComplete方法，mySubscriber就会打印出Hello World！

## 更简洁的代码

是不是觉得仅仅为了打印一个hello world要写这么多代码太啰嗦？我这里主要是为了展示RxJava背后的原理而采用了这种比较啰嗦的写法，RxJava其实提供了很多便捷的函数来帮助我们减少代码。  
  
首先来看看如何简化Observable对象的创建过程。RxJava内置了很多简化创建Observable对象的函数，比如Observable.just就是用来创建只发出一个事件就结束的Observable对象，上面创建Observable对象的代码可以简化为一行

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

接下来看看如何简化Subscriber，上面的例子中，我们其实并不关心OnComplete和OnError，我们只需要在onNext的时候做一些处理，这时候就可以使用Action1类。

Action1<String> onNextAction = **new** Action1<String>() {

    @Override

**public** **void** call(String s) {

        System.out.println(s);

    }

};

subscribe方法有一个重载版本，接受三个Action1类型的参数，分别对应OnNext，OnComplete， OnError函数。

myObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);

这里我们并不关心onError和onComplete，所以只需要第一个参数就可以

myObservable.subscribe(onNextAction);

// Outputs "Hello, world!"

上面的代码最终可以写成这样

Observable.just("Hello, world!")

    .subscribe(**new** Action1<String>() {

        @Override

**public** **void** call(String s) {

              System.out.println(s);

        }

    });

使用java8的lambda可以使代码更简洁

Observable.just("Hello, world!")

    .subscribe(s -> System.out.println(s));

Android开发中，强烈推荐使用[retrolambda](https://github.com/evant/gradle-retrolambda" \t "http://blog.csdn.net/lzyzsd/article/details/41833541/_blank)这个gradle插件，这样你就可以在你的代码中使用lambda了。

## 变换

让我们做一些更有趣的事情吧！  
比如我想在hello world中加上我的签名，你可能会想到去修改Observable对象：

Observable.just("Hello, world! -Dan")

    .subscribe(s -> System.out.println(s));

如果你能够改变Observable对象，这当然是可以的，但是如果你不能修改Observable对象呢？比如Observable对象是第三方库提供的？比如我的Observable对象被多个Subscriber订阅，但是我只想在对某个订阅者做修改呢？  
那么在Subscriber中对事件进行修改怎么样呢？比如下面的代码：

Observable.just("Hello, world!")

    .subscribe(s -> System.out.println(s + " -Dan"));

这种方式仍然不能让人满意，因为我希望我的Subscribers越轻量越好，因为我有可能会在mainThread中运行subscriber。另外，根据响应式函数编程的概念，Subscribers更应该做的事情是“响应”，响应Observable发出的事件，而不是去修改。如果我能在某些中间步骤中对“Hello World！”进行变换是不是很酷？

## 操作符（Operators）

操作符就是为了解决对Observable对象的变换的问题，操作符用于在Observable和最终的Subscriber之间修改Observable发出的事件。RxJava提供了很多很有用的操作符。  
比如map操作符，就是用来把把一个事件转换为另一个事件的。

Observable.just("Hello, world!")

  .map(**new** Func1<String, String>() {

      @Override

**public** String call(String s) {

**return** s + " -Dan";

      }

  })

  .subscribe(s -> System.out.println(s));

使用lambda可以简化为

Observable.just("Hello, world!")

    .map(s -> s + " -Dan")

    .subscribe(s -> System.out.println(s));

是不是很酷？map()操作符就是用于变换Observable对象的，map操作符返回一个Observable对象，这样就可以实现链式调用，在一个Observable对象上多次使用map操作符，最终将最简洁的数据传递给Subscriber对象。

## map操作符进阶

map操作符更有趣的一点是它不必返回Observable对象返回的类型，你可以使用map操作符返回一个发出新的数据类型的observable对象。  
比如上面的例子中，subscriber并不关心返回的字符串，而是想要字符串的hash值

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)));

很有趣吧？我们初始的Observable返回的是字符串，最终的Subscriber收到的却是Integer，当然使用lambda可以进一步简化代码：

Observable.just("Hello, world!")

    .map(s -> s.hashCode())

    .subscribe(i -> System.out.println(Integer.toString(i)));

前面说过，Subscriber做的事情越少越好，我们再增加一个map操作符

Observable.just("Hello, world!")

    .map(s -> s.hashCode())

    .map(i -> Integer.toString(i))

    .subscribe(s -> System.out.println(s));

## 不服？

是不是觉得我们的例子太简单，不足以说服你？你需要明白下面的两点:  
  
1.Observable和Subscriber可以做任何事情  
Observable可以是一个数据库查询，Subscriber用来显示查询结果；

Observable可以是屏幕上的点击事件，Subscriber用来响应点击事件；

Observable可以是一个网络请求，Subscriber用来显示请求结果。  
  
2.Observable和Subscriber是独立于中间的变换过程的。  
在Observable和Subscriber中间可以增减任何数量的map。整个系统是高度可组合的，操作数据是一个很简单的过程。