RxJava最友好教程

RxJava到底是什么？让我们直接跳过官方那种晦涩的追求精确的定义，其实初学RxJava只要把握两点：**观察者模式**和**异步**,就基本可以熟练使用RxJava了。

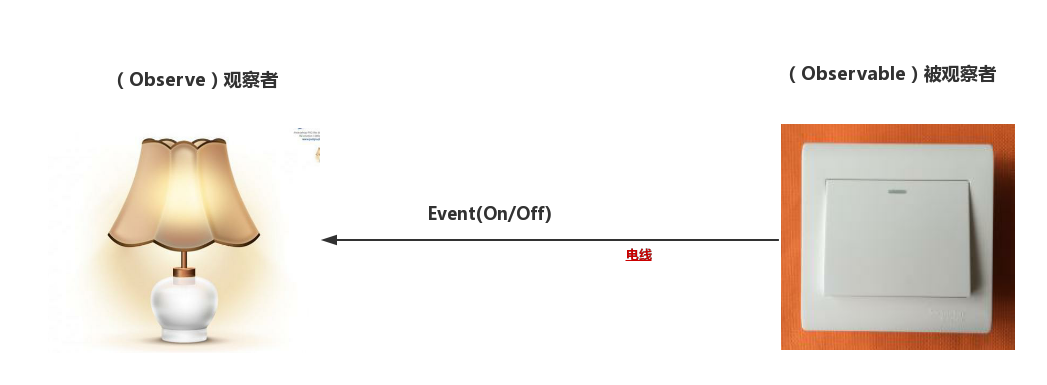
**异步**在这里并不需要做太多的解释，因为在概念和使用上，并没有太多高深的东西。大概就是你脑子里想能到的那些多线程，线程切换这些东西。我会在后面会讲解它的用法。

我们先把**观察者模式**说清楚



“按下开关，台灯灯亮”

在这个事件中，**台灯作为观察者，开关作为被观察者，台灯透过电线来观察开关的状态来并做出相应的处理**



观察上图，其实已经很明了了，不过需要指出一下几点（**对于下面理解RxJava很重要**）：

* 开关（被观察者）作为事件的**产生方**（生产“开”和“关”这两个事件），是**主动**的，是整个开灯事理流程的**起点**。
* 台灯（观察者）作为事件的**处理方**（处理“灯亮”和“灯灭”这两个事件），是**被动**的，是整个开灯事件流程的**终点**。
* 在起点和终点之间，即事件传递的过程中是可以被**加工，过滤，转换，合并等等方式**处理的（上图没有体现，后面对会讲到）。

我必须苦口婆心的告诉你：我们总结的这三点对于我们理解RxJava非常重要。因为上述三条分别对应了RxJava中被观察者(Observable),观察者(Observer)和操作符的职能。而**观察者模式又是RxJava程序运行的骨架**。

好了，我假设你已经完全理解了我上面讲述的东西。我们正式进入RxJava！

RxJava也是基于观察者模式来组建自己的程序逻辑的，就是构建**被观察者（Observable），观察者（Observer/Subscriber）**，然后建立二者的订阅关系（就像那根电线，连接起台灯和开关）实现**观察**，在事件传递过程中还可以**对事件做各种处理**。

Tips: Observer是观察者的接口， Subscriber是实现这个接口的抽象类,因此两个类都可以被当做观察者，由于Subscriber在Observe的基础上做了一些拓展，加入了新的方法，一般会更加倾向于使用Subscriber。

## 创建被观察者

* 正常模式：

Observable switcher=Observable.create(new Observable.OnSubscribe<String>(){

@Override

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

subscriber.onNext("On");

subscriber.onNext("Off");

subscriber.onNext("On");

subscriber.onNext("On");

subscriber.onCompleted();

}

});

这是最正宗的写法，创建了一个开关类，产生了五个事件，分别是：开，关，开，开，结束。

* 偷懒模式1

Observable switcher=Observable.just("On","Off","On","On");

* 偷懒模式2

String [] kk={"On","Off","On","On"};

Observable switcher=Observable.from(kk);

偷懒模式是一种简便的写法，实际上也都是**被观察者**把那些信息"On","Off","On","On"，包装成onNext（"On"）这样的事件依次发给**观察者**，当然，它自己补上了onComplete()事件。

以上是最常用到的创建方式，好了，我们就创建了一个开关类。

## 创建观察者

* 正常模式

Subscriber light=new Subscriber<String>() {

@Override

public void onCompleted() {

//被观察者的onCompleted()事件会走到这里;

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(String s) {

//处理传过来的onNext事件

Log.d("DDDDD","handle this---"+s)

}

这也是比较常见的写法，创建了一个台灯类。

* 偷懒模式(非正式写法)

Action1 light=new Action1<String>() {

@Override

public void call(String s) {

Log.d("DDDDD","handle this---"+s)

}

}

之所以说它是非正式写法，是因为Action1是一个单纯的人畜无害的接口，和Observer没有啥关系，只不过它可以当做观察者来使，专门处理onNext 事件，这是一种为了简便偷懒的写法。当然还有Action0，Action2,Action3...,0,1,2,3分别表示call()这个方法能接受几个参数。如果你还不懂，可以暂时跳过。后面我也会尽量使用new Subscriber方式，创建正统的观察者，便于你们理解。

## 订阅

现在已经创建了观察者和被观察者，但是两者还没有联系起来

switcher.subscribe(light);

我猜你看到这里应该有疑问了，为什么是开关订阅了台灯？应该是台灯订阅了开关才对啊。卧槽，到底谁观察谁啊！！

大家冷静，把刀放下，有话慢慢说，

是这样的，台灯观察开关，逻辑是没错的，而且正常来看就应该是light.subscribe(switcher)才对，之所以“开关订阅台灯”，是为了保证**流式API调用风格**

啥是**流式API调用风格**？

//这就是RxJava的流式API调用

Observable.just("On","Off","On","On")

//在传递过程中对事件进行过滤操作

.filter(new Func1<String, Boolean>() {

@Override

public Boolean call(String s) {

return s！=null;

}

})

.subscribe(mSubscriber);

上面就是一个非常简易的RxJava流式API的调用：同一个调用主体一路调用下来，一气呵成。

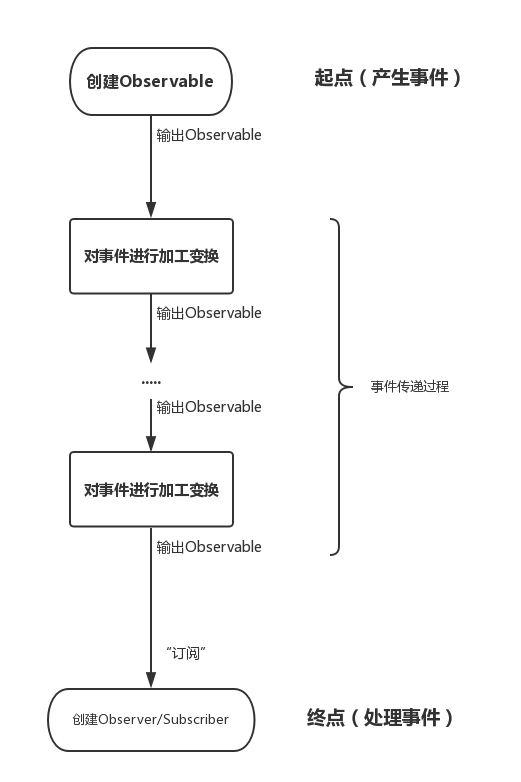
由于被观察者产生事件，是事件的起点，那么开头就是用Observable这个主体调用来创建被观察者，产生事件，为了保证流式API调用规则，就直接让Observable作为唯一的调用主体，一路调用下去。

一句话，**背后的真实的逻辑依然是台灯订阅了开关，但是在表面上，我们让开关“假装”订阅了台灯，以便于保持流式API调用风格不变。**

好了，现在分解动作都完成了，已经架构了一个基本的RxJava事件处理流程。

我们再来按照观察者模式的运作流程和流式Api的写法复习一遍：

流程图如下：



结合流程图的相应代码实例如下：

//创建被观察者，是事件传递的起点

Observable.just("On","Off","On","On")

//这就是在传递过程中对事件进行过滤操作

.filter(new Func1<String, Boolean>() {

@Override

public Boolean call(String s) {

return s！=null;

}

})

//实现订阅

.subscribe(

//创建观察者，作为事件传递的终点处理事件

new Subscriber<String>() {

@Override

public void onCompleted() {

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(String s) {

//处理事件

Log.d("DDDDD","handle this---"+s)

}

);

嗯，基本上我们就把RxJava的骨架就讲完了，总结一下：

* 创建被观察者，产生事件
* 设置事件传递过程中的过滤，合并，变换等加工操作。
* 订阅一个观察者对象，实现事件最终的处理。

Tips: 当调用订阅操作（即调用Observable.subscribe()方法）的时候，被观察者才真正开始发出事件。

现在开始讲异步操作？别着急，事件的产生起点和处理的终点我们都比较详细的讲解了，接下来我们好好讲讲事件传递过程中发生的那些事儿...

## RxJava的操作符

即使你已经看了我上面那段讲解，Rxjava可能还打动不了你，没关系，事件产生的起点和消费的终点其实没那么吸引人，真正有意思的是事件传递过程中的那些鬼斧神工的操作。

由于篇幅的限制，我只讲两三个操作，其他的操作请看我的[RxJava操作的Demo](https://github.com/ladingwu/ApplicationDemo)（**记得在github给我点star或者follow一下,不然我就坐在地上不起来，哼**）

### 变换

#### Map操作

比如被观察者产生的事件中只有图片文件路径；,但是在观察者这里只想要bitmap,那么就需要**类型变换**。

Observable.create(new Observable.just(getFilePath())

//使用map操作来完成类型转换

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

@Override

public Bitmap call(String s) {

//显然自定义的createBitmapFromPath(s)方法，是一个极其耗时的操作

return createBitmapFromPath(s);

}

})

.subscribe(

//创建观察者，作为事件传递的终点处理事件

new Subscriber<Bitmap>() {

@Override

public void onCompleted() {

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(Bitmap s) {

//处理事件

showBitmap(s)

}

);

* 实际上在使用map操作时，new Func1<String, Bitmap>() 就对应了类型的转你方向，String是原类型，Bitmap是转换后的类型。在call()方法中，输入的是原类型，返回转换后的类型

你认真看完上面的代码就会觉得，何必在过程中变换类型呢？我直接在事件传递的终点,在观察者中变换就行咯。老实说，你这个想法没毛病，但实际上，上面写的代码是不合理的。

我在代码中也提到，读取文件，创建bitmap可能是一个耗时操作，那么就应该在子线程中执行，主线程应该仅仅做展示。那么线程切换一般就会是比较复杂的事情了。但是在Rxjava中，是非常方便的。

Observable.create(new Observable.just(getFilePath())

//指定了被观察者执行的线程环境

.subscribeOn(Schedulers.newThread())

//将接下来执行的线程环境指定为io线程

.observeOn(Schedulers.io())

//使用map操作来完成类型转换

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

@Override

public Bitmap call(String s) {

//显然自定义的createBitmapFromPath(s)方法，是一个极其耗时的操作

return createBitmapFromPath(s);

}

})

//将后面执行的线程环境切换为主线程

.observeOn(AndroidSchedulers.mainThread())

.subscribe(

//创建观察者，作为事件传递的终点处理事件

new Subscriber<Bitmap>() {

@Override

public void onCompleted() {

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(Bitmap s) {

//处理事件

showBitmap(s)

}

);

由上面的代码可以看到，使用操作符将事件处理逐步分解，通过线程调度为每一步设置不同的线程环境，完全解决了你线程切换的烦恼。可以说线程调度+操作符，才真正展现了RxJava无与伦比的魅力。

### flatmap操作

先提出一个需求，查找一个学校每个班级的每个学生，并打印出来。

如果用老办法：先读出所有班级的数据，循环每个班级。再循环中再读取每个班级中每个学生，然后循环打印出来。

还是得说，这种想法，没毛病，就是嵌套得有点多。

Rxjava说：我不是针对谁...

//创建被观察者，获取所有班级

Observable.from(getSchoolClasses())

.flatMap(new Func1<SingleClass, Observable<Student>>() {

@Override

public Observable<Student> call(SingleClass singleClass) {

//将每个班级的所有学生作为一列表包装成一列Observable<Student>，将学生一个一个传递出去

return Observable.from(singleClass.getStudents());

}

})

.subscribe(

//创建观察者，作为事件传递的终点处理事件

new Subscriber<Student>() {

@Override

public void onCompleted() {

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(Student student) {

//接受到每个学生类

Log.d("DDDDDD",student.getName())

}

);

好了，基本上按照RxJava的骨架搭起来就能完成需求。你说棒不棒？？

其实FlatMap是比较难懂的一个操作符，作为初学者其实会用就好，所以我推荐的对于FlatMap的解释是：将每个Observable产生的事件里的信息再包装成新的Observable传递出来，

那么为什么FlatMap可以破除嵌套难题呢？

就是因为FlatMap可以再次包装新的Observable,而每个Observable都可以使用from(T[])方法来创建自己，这个方法接受一个列表，然后将列表中的数据包装成一系列事件。

## 异步（线程调度）

异步是相对于主线程来讲的子线程操作，在这里我们不妨使用线程调度这个概念更加贴切。

首先介绍一下RxJava的线程环境有哪些选项：



在讲解Map操作符时，已经提到了线程调度，在这里我用更加简介的代码代替：

//new Observable.just()执行在新线程

Observable.create(new Observable.just(getFilePath())

//指定在新线程中创建被观察者

.subscribeOn(Schedulers.newThread())

//将接下来执行的线程环境指定为io线程

.observeOn(Schedulers.io())

//map就处在io线程

.map(mMapOperater)

//将后面执行的线程环境切换为主线程，

//但是这一句依然执行在io线程

.observeOn(AndroidSchedulers.mainThread())

//指定线程无效，但这句代码本身执行在主线程

.subscribeOn(Schedulers.io())

//执行在主线程

.subscribe(mSubscriber);

实际上线程调度只有subscribeOn（）和observeOn（）两个方法。对于初学者，只需要掌握两点：

* subscribeOn（）它指示Observable在一个指定的调度器上**创建**（只作用于被观察者创建阶段）。只能指定一次，如果指定多次则以第一次为准
* observeOn（）指定在事件传递（加工变换）和最终被处理（观察者）的发生在哪一个调度器。可指定多次，每次指定完都在下一步生效。

线程调度掌握到这个程度，在入门阶段时绝对够用的了。

## 结尾

好了，对于RxJava整个入门文章到这里就完全结束了，现在再来回看RxJava，你会发现，**它就是在观察者模式的骨架下，通过丰富的操作符和便捷的异步操作来完成对于复杂业务的处理**。

我相信你对于整个RxJava的骨架，以及执行流程应该有了相当的了解，现在就只需要多练习一下操作符的用法了。

本文没有介绍太多的操作符，很多没来得及介绍的操作符的用法实例都放在github上的[RxJavaDemo](https://github.com/ladingwu/ApplicationDemo" \t "_blank)项目上了，后期还会继续加上更多操作符的使用，欢迎大家上去看看，对照代码，手机运行一下。

## 进阶

## 前言

之前就写过一篇《关于Rxjava最友好的文章》,反响很不错，由于那篇文章的定位就是**简单友好**，因此尽可能的摒弃复杂的概念，只抓住关键的东西来讲，以保证大家都能看懂。

不过那篇文章写完之后，我就觉得应该还得有一篇文章给RxJava做一个深入的讲解才算完美，于是就有了今天的进阶篇。因为一个团队里可能大家都会用RxJava，但是必须要有一个人很懂这个，不然碰到问题可就麻烦了。

在前一篇文章中的最后，我们得出结论：**RxJava就是在观察者模式的骨架下，通过丰富的操作符和便捷的异步操作来完成对于复杂业务的处理**。今天我们还是就结论中的**观察者模式**和**操作符**来做深入的拓展。

在进入正题之前，还是希望大家先去我的主页上看看《关于Rxjava最友好的文章》。

## 关于观察者模式

前一篇文章首先就重点谈到了观察者模式，我们认为**观察者模式RxJava的骨架\***。在这里不是要推翻之前的结论，而是希望从深入它的内部的去了解它的实现。

依然使用之前文章中关于开关和台灯的代码

//创建一个被观察者（开关）

Observable switcher=Observable.create(new Observable.OnSubscribe<String>(){

@Override

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

subscriber.onNext("On");

subscriber.onNext("Off");

subscriber.onNext("On");

subscriber.onNext("On");

subscriber.onCompleted();

}

});

//创建一个观察者（台灯）

Subscriber light=new Subscriber<String>() {

@Override

public void onCompleted() {

//被观察者的onCompleted()事件会走到这里;

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(String s) {

//处理传过来的onNext事件

Log.d("DDDDD","handle this---"+s)

}

//订阅

switcher.subscribe(light);

以上就是一个RxJava观察者架构，  
看到这样的代码不知道你会不会有一些疑惑：

* 被观察者中的Observable.**OnSubscribe**是什么，有什么用？
* call(**subscriber**)方法中，**subscriber**哪里来的?
* 为什么只有在订阅之后，被观察者才会开始发送消息？

其实，这些问题都可以通过了解OnSubscribe来解决。

那我们先来看看关于OnSubscribe的定义

//上一篇文章也提到Acton1这个接口，内部只有一个待实现call()方法

//没啥特别，人畜无害

public interface Action1<T> extends Action {

void call(T t);

}

//OnSubscribe继承了这个Action1接口

public interface OnSubscribe<T> extends Action1<Subscriber<? super T>> {

// OnSubscribe仍然是个接口

}

那么也就是说，OnSubscribe本质上也是和 Action1一样的接口，只不过它专门用于Observable内部。

而在Observable观察者的类中，OnSubscribe是它**唯一的属性**,同时也是Observable构造函数中**唯一必须**传入的参数，也就是说，只要创建了Observable，那么内部也一定有一个OnSubscribe对象。

当然，**Observable是没有办法直接new的**，我们只能通过create()，just()等等方法创建，当然，这些方法背后去调用了new Observable(onSubscribe)

public class Observable<T> {

//唯一的属性

final OnSubscribe<T> onSubscribe;

//构造函数，因为protected，我们只能使用create函数

protected Observable(OnSubscribe<T> f) {

this.onSubscribe = f;

}

//create(onSubscribe) 内部调用构造函数。

public static <T> Observable<T> create(OnSubscribe<T> f) {

return new Observable<T>(RxJavaHooks.onCreate(f));

}

....

....

}

当创建了Observable和Subscribe之后，调用subscribe(subscriber)方法时，发生了什么呢？

//传入了观察者对象

public final Subscription subscribe(final Observer<? super T> observer) {

....

//往下调用

return subscribe(new ObserverSubscriber<T>(observer));

}

public final Subscription subscribe(Subscriber<? super T> subscriber) {

//往下调用

return Observable.subscribe(subscriber, this);

}

//调用到这个函数

static <T> Subscription subscribe(Subscriber<? super T> subscriber, Observable<T> observable) {

// new Subscriber so onStart it

subscriber.onStart();

// add a significant depth to already huge call stacks.

try {

// 在这里简单讲，对onSubscribe进行封装，不必紧张。

OnSubscribe onSubscribe=RxJavaHooks.onObservableStart(observable, observable.onSubscribe);

//这个才是重点！！！

//这个调用的具体实现方法就是我们创建观察者时

//写在Observable.create()中的call()呀

//而调用了那个call(),就意味着事件开始发送了

onSubscribe.call(subscriber);

//不信你往回看

return RxJavaHooks.onObservableReturn(subscriber);

} catch (Throwable e) {

....

....

}

return Subscriptions.unsubscribed();

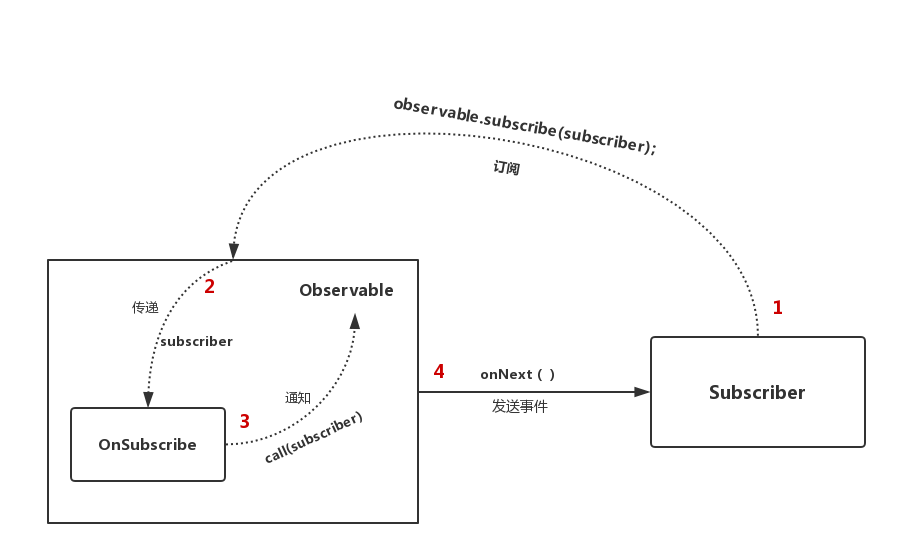
}

}

代码看到这里，我们就可以对上面三个问题做统一的回答了：

* onSubscribe是Observable内部唯一属性，是**连接Observable和subscriber的关键，相当于连接台灯和开关的那根电线**
* call(Subscriber<? super String> subscriber)中的subscriber，就是我们自己创建的那个观察者
* 只有在订阅的时候，才会发生onSubscribe.call(subscriber)，进而才会开始调用onNext(),onComplete()等。

到这里，你是不是对于RxJava的观察者模式了解更加清晰了呢？我们用流程图复习一下刚才的过程。



了解了上面这些，我们就可以更进一步做以下总结：

* **订阅这个动作，实际上是观察者(subscriber)对象把自己传递给被观察者(observable)内部的onSubscribe**。
* **onSubscribe的工作就是调用call(subscriber)来通知被观察者发送消息给这个subscriber**。

以上的结论对于下面我们理解操作符的原理十分有帮助，因此一定要看明白。

观察者模式介绍到这里，才敢说讲完了。

## 关于操作符

上一篇文章讲了一些操作符，并且在github上放了很多其他的操作符使用范例给大家，因此在这里不会介绍更多操作符的用法，而是讲解操作符的实现原理。他是如何拦截事件，然后变换处理之后，最后传递到观察者手中的呢？

相信了解相关内容的人可能会想到lift()操作符，它本来是其他操作符做变换的基础，不过那已经是好几个版本以前的事情了。但是目前版本中RxJava已经不一样了了，直接把lift()的工作下放到每个操作符中，把lift的弱化了(但是依然保留了lift()操作符)。

因此，我们在这里不必讲解lift，直接拿一个操作符做例子，来了解它的原理即可，因为基本上操作符的实现原理都是一样的。

以map()为例，依然拿之前文章里面的例子：

Observable.create(new Observable.just(getFilePath())

//使用map操作来完成类型转换

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

@Override

public Bitmap call(String s) {

//显然自定义的createBitmapFromPath(s)方法，是一个极其耗时的操作

return createBitmapFromPath(s);

}

})

.subscribe(

//创建观察者，作为事件传递的终点处理事件

new Subscriber<Bitmap>() {

@Override

public void onCompleted() {

Log.d("DDDDDD","结束观察...\n");

}

@Override

public void onError(Throwable e) {

//出现错误会调用这个方法

}

@Override

public void onNext(Bitmap s) {

//处理事件

showBitmap(s)

}

);

看看map背后到底做了什么

public final <R> Observable<R> map(Func1<? super T, ? extends R> func) {

//创建了全新代理的的Observable，构造函数传入的参数是OnSubscribe

//OnSubscribeMap显然是OnSubscribe的一个实现类，

//也就是说，OnSubscribeMap需要实现call()方法

//构造函数传入了真实的Observable对象

//和一个开发者自己实现的Func1的实例

return create(new OnSubscribeMap<T, R>(this, func));

}

看OnSubscribeMap的具体实现：

public final class OnSubscribeMap<T, R> implements OnSubscribe<R> {

//用于保存真实的Observable对象

final Observable<T> source;

//还有我们传入的那个Func1的实例

final Func1<? super T, ? extends R> transformer;

public OnSubscribeMap(Observable<T> source, Func1<? super T, ? extends R> transformer) {

this.source = source;

this.transformer = transformer;

}

//实现了call方法，我们知道call方法传入的Subscriber

//就是订阅之后，外部传入真实的的观察者

@Override

public void call(final Subscriber<? super R> o) {

//把外部传入的真实观察者传入到MapSubscriber，构造一个代理的观察者

MapSubscriber<T, R> parent = new MapSubscriber<T, R>(o, transformer);

o.add(parent);

//让外部的Observable去订阅这个代理的观察者

source.unsafeSubscribe(parent);

}

//Subscriber的子类，用于构造一个代理的观察者

static final class MapSubscriber<T, R> extends Subscriber<T> {

//这个Subscriber保存了真实的观察者

final Subscriber<? super R> actual;

//我们自己在外部自己定义的Func1

final Func1<? super T, ? extends R> mapper;

boolean done;

public MapSubscriber(Subscriber<? super R> actual, Func1<? super T, ? extends R> mapper) {

this.actual = actual;

this.mapper = mapper;

}

//外部的Observable发送的onNext()等事件

//都会首先传递到代理观察者这里

@Override

public void onNext(T t) {

R result;

try {

//mapper其实就是开发者自己创建的Func1，

//call()开始变换数据

result = mapper.call(t);

} catch (Throwable ex) {

Exceptions.throwIfFatal(ex);

unsubscribe();

onError(OnErrorThrowable.addValueAsLastCause(ex, t));

return;

}

//调用真实的观察者的onNext()

//从而在变换数据之后，把数据送到真实的观察者手中

actual.onNext(result);

}

//onError()方法也是一样

@Override

public void onError(Throwable e) {

if (done) {

RxJavaHooks.onError(e);

return;

}

done = true;

actual.onError(e);

}

@Override

public void onCompleted() {

if (done) {

return;

}

actual.onCompleted();

}

@Override

public void setProducer(Producer p) {

actual.setProducer(p);

}

}

}

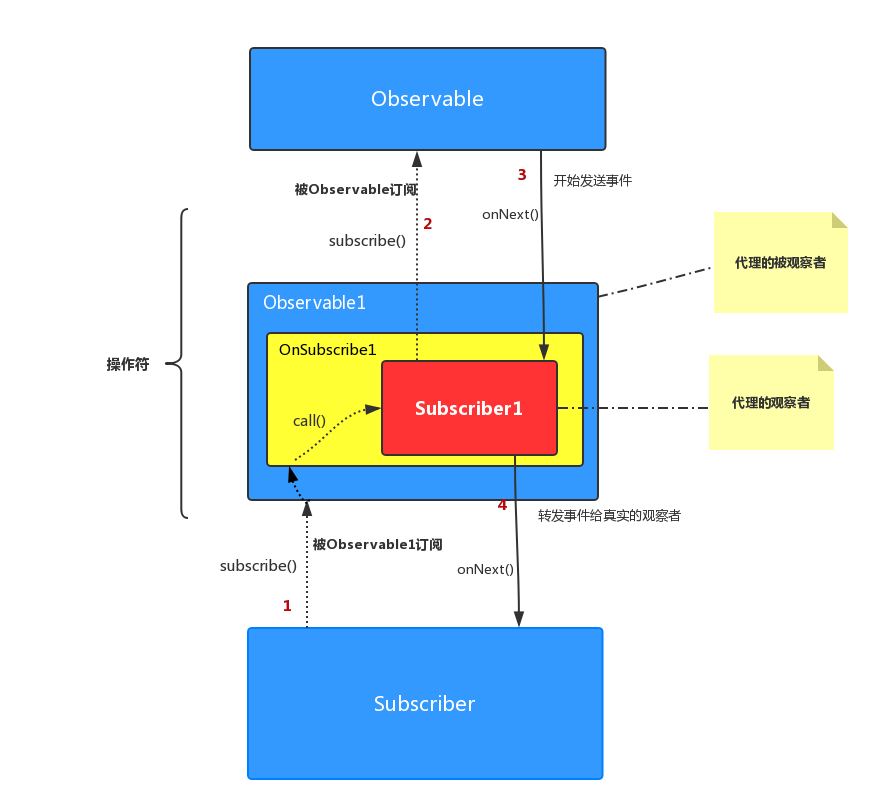
map操作符的原理基本上就讲完了，其他的操作符和map在原理上是一致的。

假如你想创建自定义的操作符(**这其实是不建议的**)，你应该按照上面的步骤

* 创建一个代理的被观察者
* 实现被观察者中onSubscribe的call方法
* 在call方法中创建一个代理的观察者，让真实的被观察者订阅它。

我知道你会有点晕，没关系，我后面会写一个自定义操作符放在我的github上,可以关注下。

下面，我们先通过一个流程图巩固一下前面学习的成果。



下次你使用操作符时，心里应该清楚，每使用一个操作符，都会创建一个代理观察者和一个代理被观察者，以及他们之间是如何相互调用的。相信有了这一层的理解，今后使用RxJava应该会更加得心应手。

不过最后，我想给大家留一个思考题：使用一个操作符时，流程图是这样的，那么使用多个操作符呢？