# 1、观察者模式

观察者模式完美的将观察者和被观察者的对象分离开。举个例子，用户界面可以作为一个观察者，业务数据是被观察者，用户界面观察业务数据的变化，发现数据变化后，就显示在界面上。

面向对象设计的一个原则是：系统中的每个类将重点放在某一个功能上，而不是其他方面。一个对象只做一件事，并且将他做好。观察者模式在模块之间划定了清晰的界限，提高了应用程序的可维护性和重用性。

因为本文将从最基本的观察者模式讲起，让你从最基本最简单的角度入手RxJava。

简单介绍一下，A和B两个，A是被观察者，B是观察者，B对A进行观察，B并不是需要时刻盯着A，而是A如果发生了变化，会主动通知B，B会对应做一些变化。举个例子，假设A是连载小说，B是读者，读者订阅了连载小说，当小说出现了新的连载的时候，会推送给读者。读者不用时刻盯着小说连载，而小说有了新的连载会主动推送给读者。这就是观察者模式。而RxJava正是基于观察者模式开发的。

# 2、RxJava2.0的基本使用

理解好了观察者模式，我们开始RxJava2.0的学习。

首先引入RxJava2.0相关的类库。  
compile 'io.reactivex.rxjava2:rxjava:2.0.1'

compile 'io.reactivex.rxjava2:rxandroid:2.0.1'

## 2.1第一步：创建连载小说（被观察者）

//被观察者

Observable novel=Observable.create(new ObservableOnSubscribe<String>() {

@Override

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

emitter.onNext("连载1");

emitter.onNext("连载2");

emitter.onNext("连载3");

emitter.onComplete();

}

});

Observable中文意思就是被观察者，通过create方法生成对象，里面放的参数ObservableOnSubscribe<T>，可以理解为一个计划表，泛型T是要操作对象的类型，重写subscribe方法，里面写具体的计划，本文的例子就是推送连载1、连载2和连载3，

在subscribe中的ObservableEmitter<String>对象的Emitter是发射器的意思。ObservableEmitter有三种发射的方法，分别是void onNext(T value)、void onError(Throwable error)、void onComplete()，onNext方法可以无限调用，Observer（观察者）所有的都能接收到，onError和onComplete是互斥的，Observer（观察者）只能接收到一个，OnComplete可以重复调用，但是Observer（观察者）只会接收一次，而onError不可以重复调用，第二次调用就会报异常。

## 2.2 第二步：创建读者（观察者）

//观察者

Observer<String> reader=new Observer<String>() {

@Override

public void onSubscribe(Disposable d) {

mDisposable=d;

Log.e(TAG,"onSubscribe");

}

@Override

public void onNext(String value) {

if ("2".equals(value)){

mDisposable.dispose();

return;

}

Log.e(TAG,"onNext:"+value);

}

@Override

public void onError(Throwable e) {

Log.e(TAG,"onError="+e.getMessage());

}

@Override

public void onComplete() {

Log.e(TAG,"onComplete()");

}

};

通过new创建接口，并实现其内部的方法，看方法其实就应该差不多知道干嘛的，onNext、onError、onComplete都是跟被观察者发射的方法一一对应的，这里就相当于接收了。onSubscribe（Disposable d）里面的Disposable对象要说一下，Disposable英文意思是可随意使用的，这里就相当于读者和连载小说的订阅关系，如果读者不想再订阅该小说了，可以调用 mDisposable.dispose()取消订阅，此时连载小说更新的时候就不会再推送给读者了。

在 Subscriber 实现的三个方法中，顾名思义，对应三种不同状态：

1. onComplete(): 事件全部处理完成后回调

2. onError(Throwable t): 事件处理异常回调

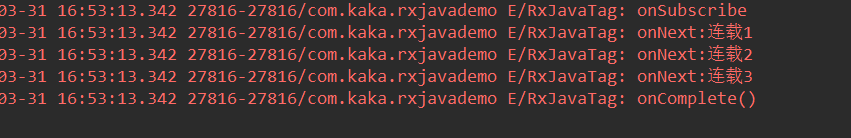
3. onNext(T t): 每接收到一个事件，回调一次

## 2.3 第三步：读者和连载小说建立订阅关系

novel.subscribe(reader);//一行代码搞定

在这里细心的你，可能已经发现了怎么是小说订阅了读者，之所以这样，是因为RxJava主要是想保持自己的链式编程，不得不把Observable(被观察者)放在前面,这里大家可以理解为小说被读者订阅了。

这里我们先看一下输出效果



小结一下：这就是RxJava2.0最最简单的用法，创建小说，创建读者，建立订阅关系，记住这三步，你就能实现一个最简单的RxJava2.0的用法。

# 3、RxJava2.0的异步和链式编程

前言里面有提到，RxJava是支持异步的，但是RxJava是如何做到的呢？这里就需要Scheduler。Scheduler，英文名调度器，它是RxJava用来控制线程。当我们没有设置的时候，RxJava遵循哪个线程产生就在哪个线程消费的原则，也就是说线程不会产生变化，始终在同一个。然后我们一般使用RxJava都是后台执行，前台调用，本着这个原则，我们需要调用observeOn(AndroidSchedulers.mainThread())，observeOn是事件回调的线程，AndroidSchedulers.mainThread()一看就知道是主线程，subscribeOn(Schedulers.io())，subscribeOn是事件执行的线程，Schedulers.io()是子线程，这里也可以用Schedulers.newThread()，只不过io线程可以重用空闲的线程，因此多数情况下 io() 比 newThread() 更有效率。前面的代码根据异步和链式编程的原则，我们可以写成

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

@Override

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

emitter.onNext("连载1");

emitter.onNext("连载2");

emitter.onNext("连载3");

emitter.onComplete();

}

})

.observeOn(AndroidSchedulers.mainThread())//回调在主线程

.subscribeOn(Schedulers.io())//执行在io线程

.subscribe(new Observer<String>() {

@Override

public void onSubscribe(Disposable d) {

Log.e(TAG,"onSubscribe");

}

@Override

public void onNext(String value) {

Log.e(TAG,"onNext:"+value);

}

@Override

public void onError(Throwable e) {

Log.e(TAG,"onError="+e.getMessage());

}

@Override

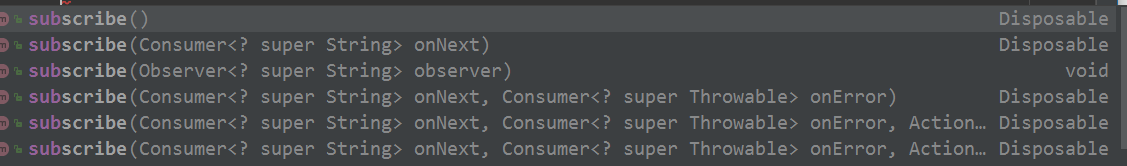
public void onComplete() {

Log.e(TAG,"onComplete()");

}

});

这里就是RxJava最常用的写法，异步+链式编程，还要再说一下，subscribe的方法重载，subscribe（）方法里什么参数也不放是空实现，也就是说连载小说无论出什么连载，读者都不关心，推送过来了也不读，如果读者只关心onNext方法里的内容，可以直接重载subscribe(Consumer<? spuer T> onNext)这个方法，会减少代码，当然如果是初学者还是建议创建Observer对象。



# 4、应用场景

说了这么多RxJava2.0的用法，你一定要问了到底在什么情况下使用？下面先给大家介绍一下典型的场景。

一、与Retrofit联用

Retrofit+RxJava的上网模式已经非常火了，如果有不了解的可以看笔者的这篇文章https://www.jianshu.com/writer#/notebooks/5118090/notes/25405151

二、Rxpermissions等类库的使用

基于RxJava的开源类库Rxpermissions、RxBinding以及RxBus在很多项目中已经非常常见，并且被证明了是极其好用的。

三、所有用到异步的地方

因为RxJava就是一个支持异步的链式编程，所以所有的用到异步的地方，我们都可以用RxJava来完成。

# 0、前言

Rxjava由于其基于事件流的链式调用、逻辑简洁 & 使用简单的特点，深受各大 Android开发者的欢迎。

# 1. 定义

RxJava 在 GitHub 的介绍：

RxJava：a library for composing asynchronous and event-based programs using observable sequences for the Java VM

// 翻译：RxJava 是一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库。

总结：RxJava 是一个 基于事件流、实现异步操作的库。

# 2. 作用

实现异步操作。

类似于 Android中的 AsyncTask 、Handler作用。

# 3. 特点

由于 RxJava的使用方式是：基于事件流的链式调用，所以使得 RxJava：

逻辑简洁

实现优雅

使用简单

更重要的是，随着程序逻辑的复杂性提高，它依然能够保持简洁 & 优雅。

# 4. 原理

## 4.1 生活例子引入

我用一个生活例子引入 & 讲解 Rxjava原理： 顾客到饭店吃饭

## 4.2 Rxjava原理介绍

Rxjava原理 基于 一种扩展的观察者模式。

Rxjava的扩展观察者模式中有4个角色：



具体原理

即RxJava原理可总结为：被观察者 （Observable） 通过 订阅（Subscribe） 按顺序发送事件 给观察者 （Observer）， 观察者（Observer） 按顺序接收事件 & 作出对应的响应动作。

# 5. 基本使用

Rxjava的使用方式有两种：

分步骤实现：该方法主要为了深入说明Rxjava的原理 & 使用，主要用于演示说明

基于事件流的链式调用：主要用于实际使用

## 5.1 方式1：分步骤实现

5.1.1 使用步骤



5.1.2 步骤详解

步骤1：创建被观察者 （Observable ）& 生产事件

即 顾客入饭店 - 坐下餐桌 - 点菜

具体实现

// 1. 创建被观察者 Observable 对象

Observable<Integer> observable = Observable.create(new ObservableOnSubscribe<Integer>() {

// create() 是 RxJava 最基本的创造事件序列的方法

// 此处传入了一个 OnSubscribe 对象参数

// 当 Observable 被订阅时，OnSubscribe 的 call() 方法会自动被调用，即事件序列就会依照设定依次被触发

// 即观察者会依次调用对应事件的复写方法从而响应事件

// 从而实现被观察者调用了观察者的回调方法 & 由被观察者向观察者的事件传递，即观察者模式

// 2. 在复写的subscribe（）里定义需要发送的事件

@Override

public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {

// 通过 ObservableEmitter类对象产生事件并通知观察者

// ObservableEmitter类介绍

// a. 定义：事件发射器

// b. 作用：定义需要发送的事件 & 向观察者发送事件

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

emitter.onComplete();

}

});

<--扩展：RxJava 提供了其他方法用于 创建被观察者对象Observable -->

// 方法1：just(T...)：直接将传入的参数依次发送出来

Observable observable = Observable.just("A", "B", "C");

// 将会依次调用：

// onNext("A");

// onNext("B");

// onNext("C");

// onCompleted();

// 方法2：from(T[]) / from(Iterable<? extends T>) : 将传入的数组 / Iterable 拆分成具体对象后，依次发送出来

String[] words = {"A", "B", "C"};

Observable observable = Observable.from(words);

// 将会依次调用：

// onNext("A");

// onNext("B");

// onNext("C");

// onCompleted();

步骤2：创建观察者 （Observer ）并 定义响应事件的行为

即 开厨房 - 确定对应菜式

发生的事件类型包括：Next事件、Complete事件 & Error事件。

具体实现

<--方式1：采用Observer 接口 -->

// 1. 创建观察者 （Observer ）对象

Observer<Integer> observer = new Observer<Integer>() {

// 2. 创建对象时通过对应复写对应事件方法 从而 响应对应事件

// 观察者接收事件前，默认最先调用复写 onSubscribe（）

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

}

// 当被观察者生产Next事件 & 观察者接收到时，会调用该复写方法 进行响应

@Override

public void onNext(Integer value) {

Log.d(TAG, "对Next事件作出响应" + value);

}

// 当被观察者生产Error事件& 观察者接收到时，会调用该复写方法 进行响应

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

// 当被观察者生产Complete事件& 观察者接收到时，会调用该复写方法 进行响应

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

};

<--方式2：采用Subscriber 抽象类 -->

// 说明：Subscriber类 = RxJava 内置的一个实现了 Observer 的抽象类，对 Observer 接口进行了扩展

// 1. 创建观察者 （Observer ）对象

Subscriber<String> subscriber = new Subscriber<Integer>() {

// 2. 创建对象时通过对应复写对应事件方法 从而 响应对应事件

// 观察者接收事件前，默认最先调用复写 onSubscribe（）

@Override

public void onSubscribe(Subscription s) {

Log.d(TAG, "开始采用subscribe连接");

}

// 当被观察者生产Next事件 & 观察者接收到时，会调用该复写方法 进行响应

@Override

public void onNext(Integer value) {

Log.d(TAG, "对Next事件作出响应" + value);

}

// 当被观察者生产Error事件& 观察者接收到时，会调用该复写方法 进行响应

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

// 当被观察者生产Complete事件& 观察者接收到时，会调用该复写方法 进行响应

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

};

<--特别注意：2种方法的区别，即Subscriber 抽象类与Observer 接口的区别 -->

// 相同点：二者基本使用方式完全一致（实质上，在RxJava的 subscribe 过程中，Observer总是会先被转换成Subscriber再使用）

// 不同点：Subscriber抽象类对 Observer 接口进行了扩展，新增了两个方法：

// 1. onStart()：在还未响应事件前调用，用于做一些初始化工作

// 2. unsubscribe()：用于取消订阅。在该方法被调用后，观察者将不再接收 & 响应事件

// 调用该方法前，先使用 isUnsubscribed() 判断状态，确定被观察者Observable是否还持有观察者Subscriber的引用，如果引用不能及时释放，就会出现内存泄露

步骤3：通过订阅（Subscribe）连接观察者和被观察者

即 顾客找到服务员 - 点菜 - 服务员下单到厨房 - 厨房烹调

具体实现

observable.subscribe(observer);

// 或者 observable.subscribe(subscriber)；

扩展说明

<-- Observable.subscribe(Subscriber) 的内部实现 -->

public Subscription subscribe(Subscriber subscriber) {

subscriber.onStart();

// 步骤1中 观察者 subscriber抽象类复写的方法，用于初始化工作

onSubscribe.call(subscriber);

// 通过该调用，从而回调观察者中的对应方法从而响应被观察者生产的事件

// 从而实现被观察者调用了观察者的回调方法 & 由被观察者向观察者的事件传递，即观察者模式

// 同时也看出：Observable只是生产事件，真正的发送事件是在它被订阅的时候，即当 subscribe() 方法执行时

}

## 5.2 方式2：优雅的实现方法 - 基于事件流的链式调用

上述的实现方式是为了说明Rxjava的原理 & 使用

在实际应用中，会将上述步骤&代码连在一起，从而更加简洁、更加优雅，即所谓的 RxJava基于事件流的链式调用

// RxJava的链式操作

Observable.create(new ObservableOnSubscribe<Integer>() {

// 1. 创建被观察者 & 生产事件

@Override

public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

emitter.onComplete();

}

}).subscribe(new Observer<Integer>() {

// 2. 通过通过订阅（subscribe）连接观察者和被观察者

// 3. 创建观察者 & 定义响应事件的行为

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

}

// 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(Integer value) {

Log.d(TAG, "对Next事件"+ value +"作出响应" );

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

}

}

注：整体方法调用顺序：观察者.onSubscribe（）> 被观察者.subscribe（）> 观察者.onNext（）>观察者.onComplete()

这种 基于事件流的链式调用，使得RxJava：

逻辑简洁

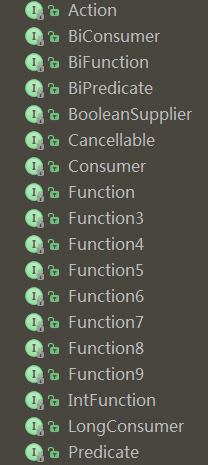
实现优雅

使用简单

更重要的是，随着程序逻辑的复杂性提高，它依然能够保持简洁 & 优雅。所以，一般建议使用这种基于事件流的链式调用方式实现RxJava。

特别注意

RxJava 2.x 提供了多个函数式接口 ，用于实现简便式的观察者模式。具体如下：



以 Consumer为例：实现简便式的观察者模式

Observable.just("hello").subscribe(new Consumer<String>() {

// 每次接收到Observable的事件都会调用Consumer.accept（）

@Override

public void accept(String s) throws Exception {

System.out.println(s);

}

});

# 7. 额外说明

## 7.1 被观察者 Observable的subscribe()具备多个重载的方法

public final Disposable subscribe() {}

// 表示观察者不对被观察者发送的事件作出任何响应（但被观察者还是可以继续发送事件）

public final Disposable subscribe(Consumer<? super T> onNext) {}

// 表示观察者只对被观察者发送的Next事件作出响应

public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError) {}

// 表示观察者只对被观察者发送的Next事件 & Error事件作出响应

public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete) {}

// 表示观察者只对被观察者发送的Next事件、Error事件 & Complete事件作出响应

public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwable> onError, Action onComplete, Consumer<? super Disposable> onSubscribe) {}

// 表示观察者只对被观察者发送的Next事件、Error事件 、Complete事件 & onSubscribe事件作出响应

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

// 表示观察者对被观察者发送的任何事件都作出响应

## 7.2 可采用 Disposable.dispose() 切断观察者 与 被观察者 之间的连接

即观察者 无法继续 接收 被观察者的事件，但被观察者还是可以继续发送事件

具体使用

// 主要在观察者 Observer中 实现

Observer<Integer> observer = new Observer<Integer>() {

// 1. 定义Disposable类变量

private Disposable mDisposable;

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

// 2. 对Disposable类变量赋值

mDisposable = d;

}

@Override

public void onNext(Integer value) {

Log.d(TAG, "对Next事件"+ value +"作出响应" );

if (value == 2) {

// 设置在接收到第二个事件后切断观察者和被观察者的连接

mDisposable.dispose();

Log.d(TAG, "已经切断了连接：" + mDisposable.isDisposed());

}

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

};

# 8. 总结

本文主要对 Rxjava 的入门知识进行讲解，包括基本介绍、原理 & 具体使用等

接下来，我将持续推出 Android中 Rxjava 2.0 的一系列文章，包括原理、操作符、应用场景、背压等等。

# 9.操作符：

创建操作符

变换操作符

组合/合并操作符

功能性操作符

过滤操作符

条件/布尔操作符

# 10.应用：

线程控制（切换/调度）

背压

网络请求轮询

网络请求嵌套回调

功能防抖

网络请求出错重连

从磁盘/内存缓存中获取缓存数据

合并数据源

联合判断

联想搜索优化