目录

给 Android 开发者的 RxJava 详解

GitHub托管

https://github.com/JackChan1999/RxJava_Docs_For_Android_Developer

GitBook在线阅读

在线阅读，PDF、ePub、Mobi电子书下载

https://www.gitbook.com/book/alleniverson/rxjava-docs-for-android-developer/details

ReactiveX/RxJava文档中文版

https://www.gitbook.com/book/alleniverson/rxdocs/welcome

目录

关注我

- Email：619888095@qq.com
- CSDN博客：Allen Iverson
- 新浪微博：AndroidDeveloper
- GitHub：JackChan1999
- GitBook：alleniverson
- 个人博客：JackChan

前言

我从去年开始使用 RxJava，到现在一年多了。今年加入了 Flipboard 后，看到 Flipboard 的 Android 项目也在使用 RxJava，并且使用的场景越来越多。而最近这几个月，我也发现国内越来越多的人开始提及 RxJava。有人说『RxJava 真是太好用了』，有人说『RxJava 真是太难用了』，另外更多的人表示：我真的百度了也谷歌了，但我还是想问：RxJava 到底是什么？

鉴于 RxJava 目前这种既火爆又神秘的现状，而我又在一年的使用过程中对 RxJava 有了一些理解，我决定写下这篇文章来对 RxJava 做一个相对详细的、针对 Android 开发者的介绍。

这篇文章的目的有两个：1. 给对 RxJava 感兴趣的人一些入门的指引 2. 给正在使用 RxJava 但仍然心存疑惑的人一些更深入的解析

在正文开始之前的最后，放上 GitHub 链接和引入依赖的 gradle 代码：Github：

- https://github.com/ReactiveX/RxJava
- https://github.com/ReactiveX/RxAndroid

引入依赖：

```
// 版本号是文章发布时的最新稳定版
compile 'io.reactivex:rxjava:1.0.14'
compile 'io.reactivex:rxandroid:1.0.1'
```

另外，感谢 RxJava 核心成员流火枫林的技术支持和内测读者代码家、鲍永章、drakeet、马琳、有时放纵、程序亦非猿、大头鬼、XZoomEye、席德雨、TCahead、Tiiime、Ailurus、宅学长、妖孽、大大大大大臣哥、NicodeLee的帮助，以及周伯通招聘的赞助。

1. RxJava 到底是什么

一个词：异步。

RxJava 在 GitHub 主页上的自我介绍是 "a library for composing asynchronous and event-based programs using observable sequences for the Java VM"（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）。这就是 RxJava，概括得非常精准。

然而，对于初学者来说，这太难看懂了。因为它是一个『总结』，而初学者更需要一个『引言』。

其实，RxJava 的本质可以压缩为异步这一个词。说到根上，它就是一个实现异步操作的库，而别的定语都是基于这之上的。

2. RxJava 好在哪

换句话说，『同样是做异步，为什么人们用它，而不用现成的 AsyncTask / Handler / XXX / ... ？』

一个词：简洁。

异步操作很关键的一点是程序的简洁性，因为在调度过程比较复杂的情况下，异步代码经常会既难写也难被读懂。 Android 创造的 AsyncTask 和Handler ， 其实都是为了让异步代码更加简洁。 RxJava 的优势也是简洁，但它的简洁的与众不同之处在于，随着程序逻辑变得越来越复杂，它依然能够保持简洁。

假设有这样一个需求：界面上有一个自定义的视图 imageCollectorView ， 它的作用是显示多张图片，并能使用 addImage(Bitmap) 方法来任意增加显示的图片。现在需要程序将一个给出的目录数组 File[] folders 中每个目录下的 png 图片都加载出来并显示在 imageCollectorView 中。需要注意的是，由于读取图片的这一过程较为耗时，需要放在后台执行，而图片的显示则必须在 UI 线程执行。常用的实现方式有多种，我这里贴出其中一种：

```
new Thread() {
    @Override
    public void run() {
        super.run();
        for (File folder : folders) {
            File[] files = folder.listFiles();
            for (File file : files) {
                if (file.getName().endsWith(".png")) {
                    final Bitmap bitmap = getBitmapFromFile(file);
                    getActivity().runOnUiThread(new Runnable() {
                        @Override
                        public void run() {
                            imageCollectorView.addImage(bitmap);
                        }
                    });
                }
            }
        }
    }
}.start();
```

而如果使用 RxJava ， 实现方式是这样的：

```
Observable.from(folders)
    .flatMap(new Func1<File, Observable<File>>() {
```

```
        @Override
        public Observable<File> call(File file) {
            return Observable.from(file.listFiles());
        }
    })
    .filter(new Func1<File, Boolean>() {
        @Override
        public Boolean call(File file) {
            return file.getName().endsWith(".png");
        }
    })
    .map(new Func1<File, Bitmap>() {
        @Override
        public Bitmap call(File file) {
            return getBitmapFromFile(file);
        }
    })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Action1<Bitmap>() {
        @Override
        public void call(Bitmap bitmap) {
            imageCollectorView.addImage(bitmap);
        }
    });
```

那位说话了：『你这代码明明变多了啊！简洁个毛啊！』大兄弟你消消气，我说的是逻辑的简洁，不是单纯的代码量少（逻辑简洁才是提升读写代码速度的必杀技对不？）。观察一下你会发现，RxJava 的这个实现，是一条从上到下的链式调用，没有任何嵌套，这在逻辑的简洁性上是具有优势的。当需求变得复杂时，这种优势将更加明显（试想如果还要求只选取前 10 张图片，常规方式要怎么办？如果有更多这样那样的要求呢？再试想，在这一大堆需求实现完两个月之后需要改功能，当你翻回这里看到自己当初写下的那一片迷之缩进，你能保证自己将迅速看懂，而不是对着代码重新捋一遍思路？）。

另外，如果你的 IDE 是 Android Studio，其实每次打开某个 Java 文件的时候，你会看到被自动 Lambda 化的预览，这将让你更加清晰地看到程序逻辑：

```
Observable.from(folders)
    .flatMap((Func1) (folder) -> { Observable.from(file.listFiles()) })
    .filter((Func1) (file) -> { file.getName().endsWith(".png") })
    .map((Func1) (file) -> { getBitmapFromFile(file) })
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe((Action1) (bitmap) -> { imageCollectorView.addImage(bitmap) });
```

> 如果你习惯使用 Retrolambda，你也可以直接把代码写成上面这种简洁的形式。而如果你看到这里还不知道什么是 Retrolambda，我不建议你现在就去学习它。原因有两点：1. Lambda 是把双刃剑，它让你的代码简洁的同时，降低了代码的可读性，因此同时学习 RxJava 和 Retrolambda 可能会让你忽略 RxJava 的一些技术细节；2. Retrolambda 是 Java 6/7 对 Lambda 表达式的非官方兼容方案，它的向后兼容性和稳定性是无法保障的，因此对于企业项目，使用 Retrolambda 是有风险的。所以，与很多 RxJava 的推广者不同，我并不推荐在学习 RxJava 的同时一起学习 Retrolambda。事实上，我个人虽然很欣赏 Retrolambda，但我从来不用它。

在Flipboard 的 Android 代码中，有一段逻辑非常复杂，包含了多次内存操作、本地文件操作和网络操作，对象分分合合，线程间相互配合相互等待，一会儿排成人字，一会儿排成一字。如果使用常规的方法来实现，肯定是要写得欲仙欲死，然而在使用 RxJava 的情况下，依然只是一条链式调用就完成了。它很长，但很清晰。

所以， RxJava 好在哪？就好在简洁，好在那把什么复杂逻辑都能穿成一条线的简洁。

3. API 介绍和原理简析

这个我就做不到一个词说明了……因为这一节的主要内容就是一步步地说明 RxJava 到底怎样做到了异步，怎样做到了简洁。

3.1 概念：扩展的观察者模式

RxJava 的异步实现，是通过一种扩展的观察者模式来实现的。

观察者模式

先简述一下观察者模式，已经熟悉的可以跳过这一段。

观察者模式面向的需求是：A 对象（观察者）对 B 对象（被观察者）的某种变化高度敏感，需要在 B 变化的一瞬间做出反应。举个例子，新闻里喜闻乐见的警察抓小偷，警察需要在小偷伸手作案的时候实施抓捕。在这个例子里，警察是观察者，小偷是被观察者，警察需要时刻盯着小偷的一举一动，才能保证不会漏过任何瞬间。程序的观察者模式和这种真正的『观察』略有不同，观察者不需要时刻盯着被观察者（例如 A 不需要每过 2ms 就检查一次 B 的状态），而是采用注册(Register)或者称为订阅(Subscribe)的方式，告诉被观察者：我需要你的某某状态，你要在它变化的时候通知我。 Android 开发中一个比较典型的例子是点击监听器 OnClickListener 。对设置 OnClickListener 来说， View 是被观察者， OnClickListener 是观察者，二者通过 setOnClickListener() 方法达成订阅关系。订阅之后用户点击按钮的瞬间，Android Framework 就会将点击事件发送给已经注册的 OnClickListener 。采取这样被动的观察方式，既省去了反复检索状态的资源消耗，也能够得到最高的反馈速度。当然，这也得益于我们可以随意定制自己程序中的观察者和被观察者，而警察叔叔明显无法要求小偷『你在作案的时候务必通知我』。

OnClickListener 的模式大致如下图：

如图所示，通过 setOnClickListener() 方法，Button 持有 OnClickListener 的引用（这一过程没有在图上画出）；当用户点击时，Button 自动调用 OnClickListener 的 onClick() 方法。另外，如果把这张图中的概念抽象出来（Button -> 被观察者、OnClickListener -> 观察者、setOnClickListener() -> 订阅，onClick() -> 事件），就由专用的观察者模式（例如只用于监听控件点击）转变成了通用的观察者模式。如下图：

而 RxJava 作为一个工具库，使用的就是通用形式的观察者模式。

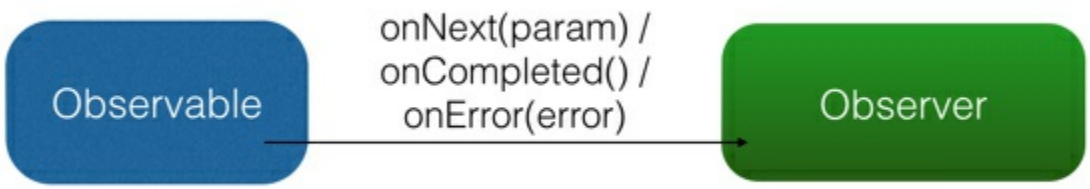
RxJava 的观察者模式

RxJava 有四个基本概念：Observable (可观察者，即被观察者)、 Observer (观察者)、 subscribe (订阅)、事件。Observable 和 Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer。

与传统观察者模式不同， RxJava 的事件回调方法除了普通事件 onNext() （相当于 onClick() / onEvent()）之外，还定义了两个特殊的事件：onCompleted() 和 onError()。

onCompleted(): 事件队列完结。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的 onNext() 发出时，需要触发 onCompleted() 方法作为标志。

onError(): 事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。 在一个正确运行的事件序列中, onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。需要注意的是，onCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。 RxJava 的观察者模式大致如下图：



3.2 基本实现

基于以上的概念， RxJava 的基本实现主要有三点：

1) 创建 Observer

Observer 即观察者，它决定事件触发的时候将有怎样的行为。 RxJava 中的 Observer 接口的实现方式：

```
Observer<String> observer = new Observer<String>() {
    @Override
    public void onNext(String s) {
        Log.d(tag, "Item: " + s);
    }

    @Override
    public void onCompleted() {
        Log.d(tag, "Completed!");
    }

    @Override
    public void onError(Throwable e) {
        Log.d(tag, "Error!");
    }
};
```

除了 Observer 接口之外，RxJava 还内置了一个实现了 Observer 的抽象类：Subscriber。 Subscriber 对 Observer 接口进行了一些扩展，但他们的基本使用方式是完全一样的：

```
Subscriber<String> subscriber = new Subscriber<String>() {
    @Override
    public void onNext(String s) {
        Log.d(tag, "Item: " + s);
    }

    @Override
```

```
    public void onCompleted() {
        Log.d(tag, "Completed!");
    }

    @Override
    public void onError(Throwable e) {
        Log.d(tag, "Error!");
    }
};
```

不仅基本使用方式一样，实质上，在 RxJava 的 subscribe 过程中，Observer 也总是会先被转换成一个 Subscriber 再使用。所以如果你只想使用基本功能，选择 Observer 和 Subscriber 是完全一样的。它们的区别对于使用者来说主要有两点：

onStart(): 这是 Subscriber 增加的方法。它会在 subscribe 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作，例如数据的清零或重置。这是一个可选方法，默认情况下它的实现为空。需要注意的是，如果对准备工作的线程有要求（例如弹出一个显示进度的对话框，这必须在主线程执行），onStart() 就不适用了，因为它总是在 subscribe 所发生的线程被调用，而不能指定线程。要在指定的线程来做准备工作，可以使用 doOnSubscribe() 方法，具体可以在后面的文中看到。

unsubscribe(): 这是 Subscriber 所实现的另一个接口 Subscription 的方法，用于取消订阅。在这个方法被调用后，Subscriber 将不再接收事件。一般在这个方法调用前，可以使用 isUnsubscribed() 先判断一下状态。unsubscribe() 这个方法很重要，因为在 subscribe() 之后，Observable 会持有 Subscriber 的引用，这个引用如果不能及时被释放，将有内存泄露的风险。所以最好保持一个原则：要在不再使用的时候尽快在合适的地方（例如 onPause() onStop() 等方法中）调用 unsubscribe() 来解除引用关系，以避免内存泄露的发生。

2) 创建 Observable

Observable 即被观察者，它决定什么时候触发事件以及触发怎样的事件。RxJava 使用 create() 方法来创建一个 Observable，并为它定义事件触发规则：

```
Observable observable = Observable.create(new Observable.OnSubscribe<String>() {
    @Override
    public void call(Subscriber<? super String> subscriber) {
        subscriber.onNext("Hello");
        subscriber.onNext("Hi");
        subscriber.onNext("Aloha");
        subscriber.onCompleted();
    }
});
```

可以看到，这里传入了一个 OnSubscribe 对象作为参数。OnSubscribe 会被存储在返回的 Observable 对象中，它的作用相当于一个计划表，当 Observable 被订阅的时候，OnSubscribe 的 call() 方法会自动被调用，事件序列就会依照设定依次触发（对于上面的代码，就是观察者

Subscriber 将会被调用三次 onNext() 和一次 onCompleted()) 。这样，由被观察者调用了观察者的回调方法，就实现了由被观察者向观察者的事件传递，即观察者模式。

> 这个例子很简单：事件的内容是字符串，而不是一些复杂的对象；事件的内容是已经定好了的，而不像有的观察者模式一样是待确定的（例如网络请求的结果在请求返回之前是未知的）；所有事件在一瞬间被全部发送出去，而不是夹杂一些确定或不确定的时间间隔或者经过某种触发器来触发的。总之，这个例子看起来毫无实用价值。但这是为了便于说明，实质上只要你想，各种各样的事件发送规则你都可以自己来写。至于具体怎么做，后面都会讲到，但现在不行。只有把基础原理先说明白了，上层的运用才能更容易说清楚。

create() 方法是 RxJava 最基本的创造事件序列的方法。基于这个方法， RxJava 还提供了一些方法用来快捷创建事件队列，例如：

- just(T...): 将传入的参数依次发送出来。

```
Observable observable = Observable.just("Hello", "Hi", "Aloha");
// 将会依次调用：
// onNext("Hello");
// onNext("Hi");
// onNext("Aloha");
// onCompleted();
```

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

```
String[] words = {"Hello", "Hi", "Aloha"};
Observable observable = Observable.from(words);
// 将会依次调用：
// onNext("Hello");
// onNext("Hi");
// onNext("Aloha");
// onCompleted();
```

上面 just(T...) 的例子和 from(T[]) 的例子，都和之前的 create(OnSubscribe) 的例子是等价的。

3) Subscribe (订阅)

创建了 Observable 和 Observer 之后，再用 subscribe() 方法将它们联结起来，整条链子就可以工作了。代码形式很简单：

```
observable.subscribe(observer);
// 或者：
observable.subscribe(subscriber);
```

> 有人可能会注意到， subscribe() 这个方法有点怪：它看起来是『observalbe 订阅了 observer / subscriber』而不是『observer / subscriber 订阅了 observalbe』，这看起来就像『杂志订阅了读者』一样颠倒了对象关系。这让人读起来有点别扭，不过如果把 API 设计成 observer.subscribe(observable) / subscriber.subscribe(observable)，虽然更加符合思维逻辑，但对流式 API 的设计就造成影响了，比较起来明显是得不偿失的。

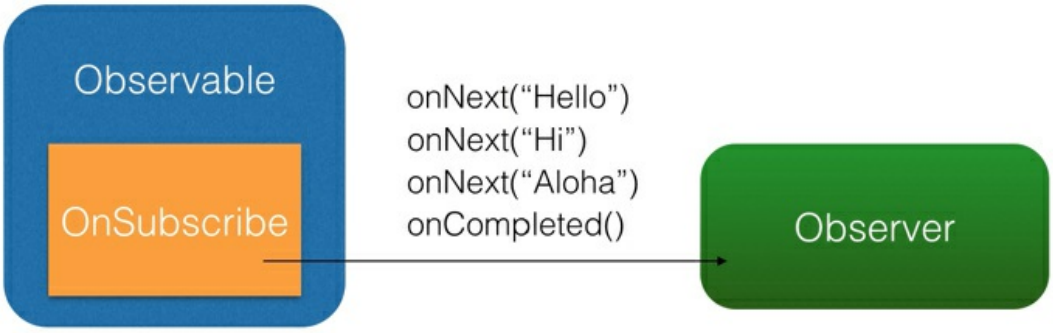
`Observable.subscribe(Subscriber)` 的内部实现是这样的（仅核心代码）：

```
// 注意：这不是 subscribe() 的源码，而是将源码中与性能、兼容性、扩展性有关的代码剔除后的核心代码。
// 如果需要看源码，可以去 RxJava 的 GitHub 仓库下载。
public Subscription subscribe(Subscriber subscriber) {
    subscriber.onStart();
    onSubscribe.call(subscriber);
    return subscriber;
}
```

可以看到，subscriber() 做了3件事：

1. 调用 Subscriber.onStart() 。这个方法在前面已经介绍过，是一个可选的准备方法。
2. 调用 Observable 中的 OnSubscribe.call(Subscriber) 。在这里，事件发送的逻辑开始运行。从这也可以看出，在 RxJava 中， Observable 并不是在创建的时候就立即开始发送事件，而是在它被订阅的时候，即当 subscribe() 方法执行的时候。
3. 将传入的 Subscriber 作为 Subscription 返回。这是为了方便 unsubscribe().

整个过程中对象间的关系如下图：



或者可以看动图：

除了 subscribe(Observer) 和 subscribe(Subscriber)，subscribe() 还支持不完整定义的回调，RxJava 会自动根据定义创建出 Subscriber 。形式如下：

```
Action1<String> onNextAction = new Action1<String>() {
    // onNext()
    @Override
    public void call(String s) {
        Log.d(tag, s);
    }
};
Action1<Throwable> onErrorAction = new Action1<Throwable>() {
    // onError()
    @Override
    public void call(Throwable throwable) {
        // Error handling
    }
};
Action0 onCompletedAction = new Action0() {
    // onCompleted()
    @Override
    public void call() {
        Log.d(tag, "completed");
    }
};

// 自动创建 Subscriber ，并使用 onNextAction 来定义 onNext()
observable.subscribe(onNextAction);
// 自动创建 Subscriber ，并使用 onNextAction 和 onErrorAction 来定义 onNext() 和 onError()

observable.subscribe(onNextAction, onErrorAction);
// 自动创建 Subscriber ，并使用 onNextAction、 onErrorAction 和 onCompletedAction 来定义
onNext()、 onError() 和 onCompleted()
```

```
observable.subscribe(onNextAction, onErrorAction, onCompletedAction);
```

简单解释一下这段代码中出现的 Action1 和 Action0。 Action0 是 RxJava 的一个接口，它只有一个方法 call()，这个方法是无参无返回值的；由于 onCompleted() 方法也是无参无返回值的，因此 Action0 可以被当成一个包装对象，将 onCompleted() 的内容打包起来将自己作为一个参数传入 subscribe() 以实现不完整定义的回调。这样其实也可以看做将 onCompleted() 方法作为参数传进了 subscribe()，相当于其他某些语言中的『闭包』。 Action1 也是一个接口，它同样只有一个方法 call(T param)，这个方法也无返回值，但有一个参数；与 Action0 同理，由于 onNext(T obj) 和 onError(Throwable error) 也是单参数无返回值的，因此 Action1 可以将 onNext(obj) 和 onError(error) 打包起来传入 subscribe() 以实现不完整定义的回调。事实上，虽然 Action0 和 Action1 在 API 中使用最广泛，但 RxJava 是提供了多个 ActionX 形式的接口 (例如 Action2, Action3) 的，它们可以被用以包装不同的无返回值的方法。

> 注：正如前面所提到的，Observer 和 Subscriber 具有相同的角色，而且 Observer 在 subscribe() 过程中最终会被转换成 Subscriber 对象，因此，从这里开始，后面的描述我将用 Subscriber 来代替 Observer，这样更加严谨。

4) 场景示例

下面举两个例子：

> 为了把原理用更清晰的方式表述出来，本文中挑选的都是功能尽可能简单的例子，以至于有些示例代码看起来会有『画蛇添足』『明明不用 RxJava 可以更简便地解决问题』的感觉。当你看到这种情况，不要觉得是因为 RxJava 太啰嗦，而是因为在过早的时候举出真实场景的例子并不利于原理的解析，因此我刻意挑选了简单的情景。

a. 打印字符串数组

将字符串数组 names 中的所有字符串依次打印出来：

```
String[] names = ...;
Observable.from(names)
    .subscribe(new Action1<String>() {
        @Override
        public void call(String name) {
            Log.d(tag, name);
        }
    });
```

b. 由 id 取得图片并显示

由指定的一个 drawable 文件 id drawableRes 取得图片，并显示在 ImageView 中，并在出现异常的时候打印 Toast 报错：

```
int drawableRes = ...;
```

```
ImageView imageView = ...;
Observable.create(new OnSubscribe<Drawable>() {
    @Override
    public void call(Subscriber<? super Drawable> subscriber) {
        Drawable drawable = getTheme().getDrawable(drawableRes));
        subscriber.onNext(drawable);
        subscriber.onCompleted();
    }
}).subscribe(new Observer<Drawable>() {
    @Override
    public void onNext(Drawable drawable) {
        imageView.setImageDrawable(drawable);
    }

    @Override
    public void onCompleted() {
    }

    @Override
    public void onError(Throwable e) {
        Toast.makeText(activity, "Error!", Toast.LENGTH_SHORT).show();
    }
});
```

正如上面两个例子这样，创建出 Observable 和 Subscriber ，再用 subscribe() 将它们串起来，一次 RxJava 的基本使用就完成了。非常简单。

然而，

在 RxJava 的默认规则中，事件的发出和消费都是在同一个线程的。也就是说，如果只用上面的方法，实现出来的只是一个同步的观察者模式。观察者模式本身的目的就是『后台处理，前台回调』的异步机制，因此异步对于 RxJava 是至关重要的。而要实现异步，则需要用到 RxJava 的另一个概念： Scheduler 。

3.3 线程控制 —— Scheduler (一)

在不指定线程的情况下， RxJava 遵循的是线程不变的原则，即：在哪个线程调用 subscribe()，就在哪个线程生产事件；在哪个线程生产事件，就在哪个线程消费事件。如果需要切换线程，就需要用到 Scheduler （调度器）。

1) Scheduler 的 API (一)

在RxJava 中，Scheduler ——调度器，相当于线程控制器，RxJava 通过它来指定每一段代码应该运行在什么样的线程。RxJava 已经内置了几个 Scheduler ， 它们已经适合大多数的使用场景：

- Schedulers.immediate(): 直接在当前线程运行，相当于不指定线程。这是默认的 Scheduler。
- Schedulers.newThread(): 总是启用新线程，并在新线程执行操作。
- Schedulers.io(): I/O 操作（读写文件、读写数据库、网络信息交互等）所使用的 Scheduler。行为模式和 newThread() 差不多，区别在于 io() 的内部实现是是用一个无数量上限的线程池，可以重用空闲的线程，因此多数情况下 io() 比 newThread() 更有效率。不要把计算工作放在 io() 中，可以避免创建不必要的线程。
- Schedulers.computation(): 计算所使用的 Scheduler。这个计算指的是 CPU 密集型计算，即不会被 I/O 等操作限制性能的操作，例如图形的计算。这个 Scheduler 使用的固定的线程池，大小为 CPU 核数。不要把 I/O 操作放在 computation() 中，否则 I/O 操作的等待时间会浪费 CPU。
- 另外， Android 还有一个专用的 AndroidSchedulers.mainThread()，它指定的操作将在 Android 主线程运行。

有了这几个 Scheduler ，就可以使用 subscribeOn() 和 observeOn() 两个方法来对线程进行控制了。subscribeOn(): 指定 subscribe() 所发生的线程，即 Observable.OnSubscribe 被激活时所处的线程。或者叫做事件产生的线程。observeOn(): 指定 Subscriber 所运行在的线程。或者叫做事件消费的线程。

文字叙述总归难理解，上代码：

```
Observable.just(1, 2, 3, 4)
    .subscribeOn(Schedulers.io()) // 指定 subscribe() 发生在 IO 线程
    .observeOn(AndroidSchedulers.mainThread()) // 指定 Subscriber 的回调发生在主线程
    .subscribe(new Action1<Integer>() {
        @Override
        public void call(Integer number) {
            Log.d(tag, "number:" + number);
        }
    });
```

上面这段代码中，由于 subscribeOn(Schedulers.io()) 的指定，被创建的事件的内容 1、2、3、4 将会在 IO 线程发出；而由于 observeOn(AndroidScheculers.mainThread()) 的指定，因此 subscriber 数字的打印将发生在主线程 。事实上，这种在 subscribe() 之前写上两句 subscribeOn(Scheduler.io()) 和 observeOn(AndroidSchedulers.mainThread()) 的使用方式非常常见，它适用于多数的 『后台线程取数据，主线程显示』的程序策略。

而前面提到的由图片 id 取得图片并显示的例子，如果也加上这两句：

```
int drawableRes = ...;
ImageView imageView = ...;
Observable.create(new OnSubscribe<Drawable>() {
    @Override
    public void call(Subscriber<? super Drawable> subscriber) {
        Drawable drawable = getTheme().getDrawable(drawableRes));
        subscriber.onNext(drawable);
        subscriber.onCompleted();
    }
})
.subscribeOn(Schedulers.io()) // 指定 subscribe() 发生在 IO 线程
.observeOn(AndroidSchedulers.mainThread()) // 指定 Subscriber 的回调发生在主线程
.subscribe(new Observer<Drawable>() {
    @Override
    public void onNext(Drawable drawable) {
        imageView.setImageDrawable(drawable);
    }

    @Override
    public void onCompleted() {
    }

    @Override
    public void onError(Throwable e) {
        Toast.makeText(activity, "Error!", Toast.LENGTH_SHORT).show();
    }
});
```

那么，加载图片将会发生在 IO 线程，而设置图片则被设定在了主线程。这就意味着，即使加载图片耗费了几十甚至几百毫秒的时间，也不会造成丝毫界面的卡顿。

2) Scheduler 的原理 (一)

RxJava 的 Scheduler API 很方便，也很神奇（加了一句话就把线程切换了，怎么做到的？而且 subscribe() 不是最外层直接调用的方法吗，它竟然也能被指定线程？）。然而 Scheduler 的原理需要放在后面讲，因为它的原理是以下一节《变换》的原理作为基础的。

好吧这一节其实我屁也没说，只是为了让你安心，让你知道我不是忘了讲原理，而是把它放在了更合适的地方。

3.4 变换

终于要到牛逼的地方了，不管你激动不激动，反正我是激动了。

RxJava 提供了对事件序列进行变换的支持，这是它的核心功能之一，也是大多数人说『RxJava 真是太好用了』的最大原因。所谓变换，就是将事件序列中的对象或整个序列进行加工处理，转换成不同的事件或事件序列。概念说着总是模糊难懂的，来看 API。

1) API

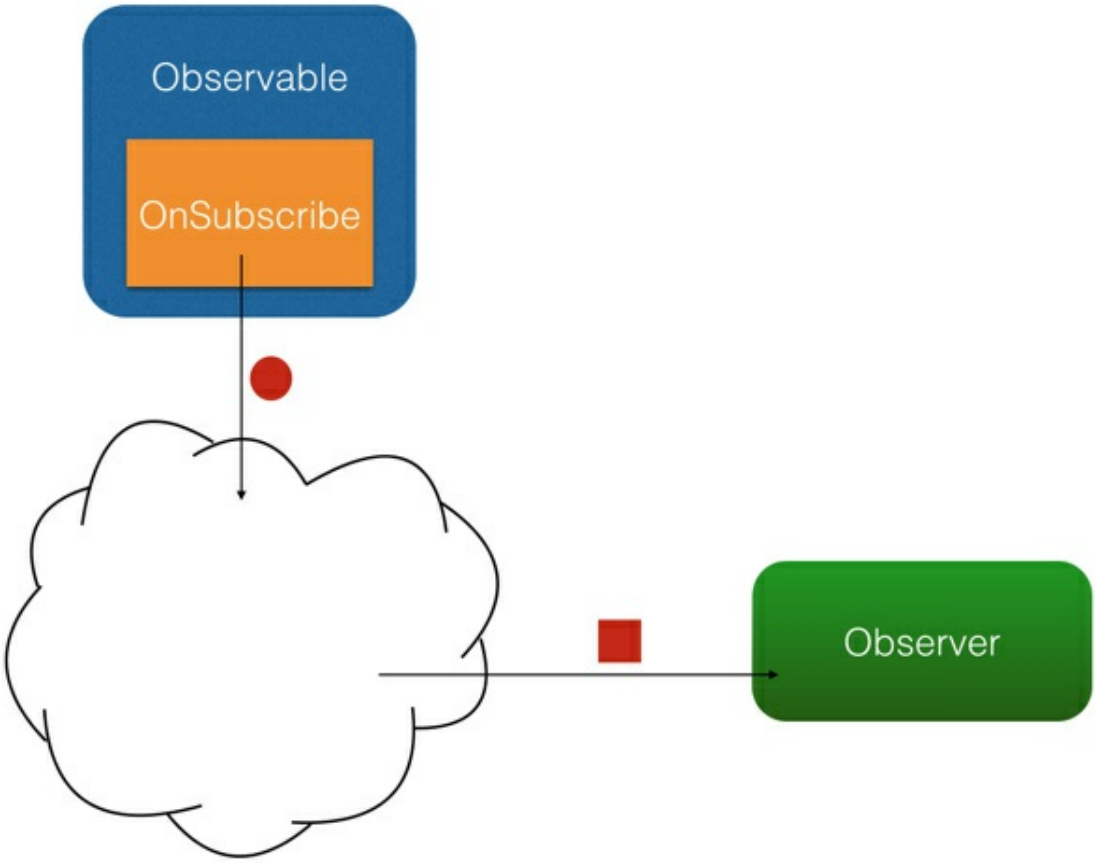
首先看一个 map() 的例子：

```
Observable.just("images/logo.png") // 输入类型 String
    .map(new Func1<String, Bitmap>() {
        @Override
        public Bitmap call(String filePath) { // 参数类型 String
            return getBitmapFromPath(filePath); // 返回类型 Bitmap
        }
    })
    .subscribe(new Action1<Bitmap>() {
        @Override
        public void call(Bitmap bitmap) { // 参数类型 Bitmap
            showBitmap(bitmap);
        }
    });
```

这里出现了一个叫做 Func1 的类。它和 Action1 非常相似，也是 RxJava 的一个接口，用于包装含有一个参数的方法。 Func1 和 Action 的区别在于， Func1 包装的是有返回值的方法。另外，和 ActionX 一样， FuncX 也有多个，用于不同参数个数的方法。FuncX 和 ActionX 的区别在 FuncX 包装的是有返回值的方法。

可以看到，map() 方法将参数中的 String 对象转换成一个 Bitmap 对象后返回，而在经过 map() 方法后，事件的参数类型也由 String 转为了 Bitmap。这种直接变换对象并返回的，是最常见的也最容易理解的变换。不过 RxJava 的变换远不止这样，它不仅可以针对事件对象，还可以针对整个事件队列，这使得 RxJava 变得非常灵活。我列举几个常用的变换：

- map(): 事件对象的直接变换，具体功能上面已经介绍过。它是 RxJava 最常用的变换。 map() 的示意图：



- flatMap(): 这是一个很有用但非常难理解的变换，因此我决定花多些篇幅来介绍它。 首先假设这么一种需求：假设有一个数据结构『学生』，现在需要打印出一组学生的名字。实现方式很简单：

```
Student[] students = ...;
Subscriber<String> subscriber = new Subscriber<String>() {
    @Override
    public void onNext(String name) {
        Log.d(tag, name);
    }
    ...
};
Observable.from(students)
    .map(new Func1<Student, String>() {
        @Override
        public String call(Student student) {
            return student.getName();
        }
    })
    .subscribe(subscriber);
```

很简单。那么再假设：如果要打印出每个学生所需要修的所有课程的名称呢？（需求的区别在于，每个学生只有一个名字，但却有多个课程。）首先可以这样实现：

```
Student[] students = ...;
Subscriber<Student> subscriber = new Subscriber<Student>() {
    @Override
    public void onNext(Student student) {
        List<Course> courses = student.getCourses();
        for (int i = 0; i < courses.size(); i++) {
            Course course = courses.get(i);
            Log.d(tag, course.getName());
        }
    }
    ...
};

Observable.from(students).subscribe(subscriber);
```

依然很简单。那么如果我不想在 Subscriber 中使用 for 循环，而是希望 Subscriber 中直接传入单个的 Course 对象呢（这对于代码复用很重要）？用 map() 显然是不行的，因为 map() 是一对一的转化，而我现在的要求是一对多的转化。那怎么才能把一个 Student 转化成多个 Course 呢？

这个时候，就需要用 flatMap() 了：

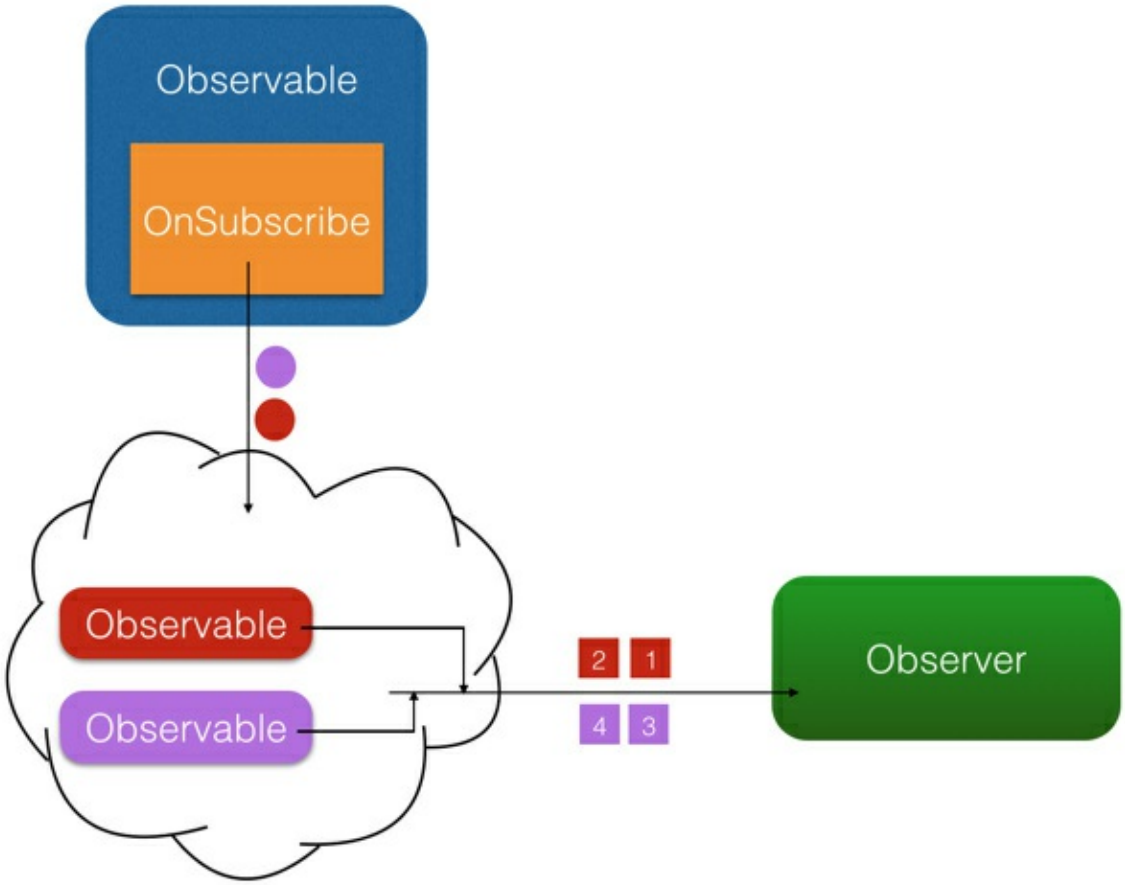
```
Student[] students = ...;
Subscriber<Course> subscriber = new Subscriber<Course>() {
    @Override
    public void onNext(Course course) {
        Log.d(tag, course.getName());
    }
    ...
};
Observable.from(students)
    .flatMap(new Func1<Student, Observable<Course>>() {
        @Override
        public Observable<Course> call(Student student) {
            return Observable.from(student.getCourses());
        }
    })
    .subscribe(subscriber);
```

从上面的代码可以看出， flatMap() 和 map() 有一个相同点：它也是把传入的参数转化之后返回另一个对象。但需要注意，和 map() 不同的是， flatMap() 中返回的是个 Observable 对象，并且这个 Observable 对象并不是被直接发送到了 Subscriber 的回调方法中。 flatMap() 的原理是这样的：

1. 使用传入的事件对象创建一个 Observable 对象；

2. 并不发送这个 Observable, 而是将它激活，于是它开始发送事件；
3. 每一个创建出来的 Observable 发送的事件，都被汇入同一个 Observable，而这个 Observable 负责将这些事件统一交给 Subscriber 的回调方法。这三个步骤，把事件拆成了两级，通过一组新创建的 Observable 将初始的对象『铺平』之后通过统一路径分发了下去。而这个『铺平』就是 flatMap() 所谓的 flat。

flatMap() 示意图：



扩展：由于可以在嵌套的 Observable 中添加异步代码， flatMap() 也常用于嵌套的异步操作，例如嵌套的网络请求。示例代码（Retrofit + RxJava）：

```
networkClient.token() // 返回 Observable<String>，在订阅时请求 token，并在响应后发送 token
    .flatMap(new Func1<String, Observable<Messages>>() {
        @Override
        public Observable<Messages> call(String token) {
            // 返回 Observable<Messages>，在订阅时请求消息列表，并在响应后发送请求到的消息列表

            return networkClient.messages();
        }
    })
    .subscribe(new Action1<Messages>() {
        @Override
```

```
        public void call(Messages messages) {
            // 处理显示消息列表
            showMessages(messages);
        }
    });
```

传统的嵌套请求需要使用嵌套的 Callback 来实现。而通过 flatMap()，可以把嵌套的请求写在一条链中，从而保持程序逻辑的清晰。

- throttleFirst(): 在每次事件触发后的一定时间间隔内丢弃新的事件。常用作去抖动过滤，例如按钮的点击监听器： RxView.clickEvents(button) // RxBinding 代码，后面的文章有解释 .throttleFirst(500, TimeUnit.MILLISECONDS) // 设置防抖间隔为 500ms .subscribe(subscriber); 妈妈再也不怕我的用户手抖点开两个重复的界面啦。

此外， RxJava 还提供很多便捷的方法来实现事件序列的变换，这里就不一一举例了。

2) 变换的原理：lift()

这些变换虽然功能各有不同，但实质上都是针对事件序列的处理和再发送。而在 RxJava 的内部，它们是基于同一个基础的变换方法： lift(Operator)。首先看一下 lift() 的内部实现（仅核心代码）：

```
// 注意：这不是 lift() 的源码，而是将源码中与性能、兼容性、扩展性有关的代码剔除后的核心代码。
// 如果需要看源码，可以去 RxJava 的 GitHub 仓库下载。
public <R> Observable<R> lift(Operator<? extends R, ? super T> operator) {
    return Observable.create(new OnSubscribe<R>() {
        @Override
        public void call(Subscriber subscriber) {
            Subscriber newSubscriber = operator.call(subscriber);
            newSubscriber.onStart();
            onSubscribe.call(newSubscriber);
        }
    });
}
```

这段代码很有意思：它生成了一个新的 Observable 并返回，而且创建新 Observable 所用的参数 OnSubscribe 的回调方法 call() 中的实现竟然看起来和前面讲过的 Observable.subscribe() 一样！然而它们并不一样哟~不一样的地方关键就在于第二行 onSubscribe.call(subscriber) 中的 onSubscribe 所指代的对象不同（高能预警：接下来的几句话可能会导致身体的严重不适）

- subscribe() 中这句话的 onSubscribe 指的是 Observable 中的 onSubscribe 对象，这个没有问题，但是 lift() 之后的情况就复杂了点。

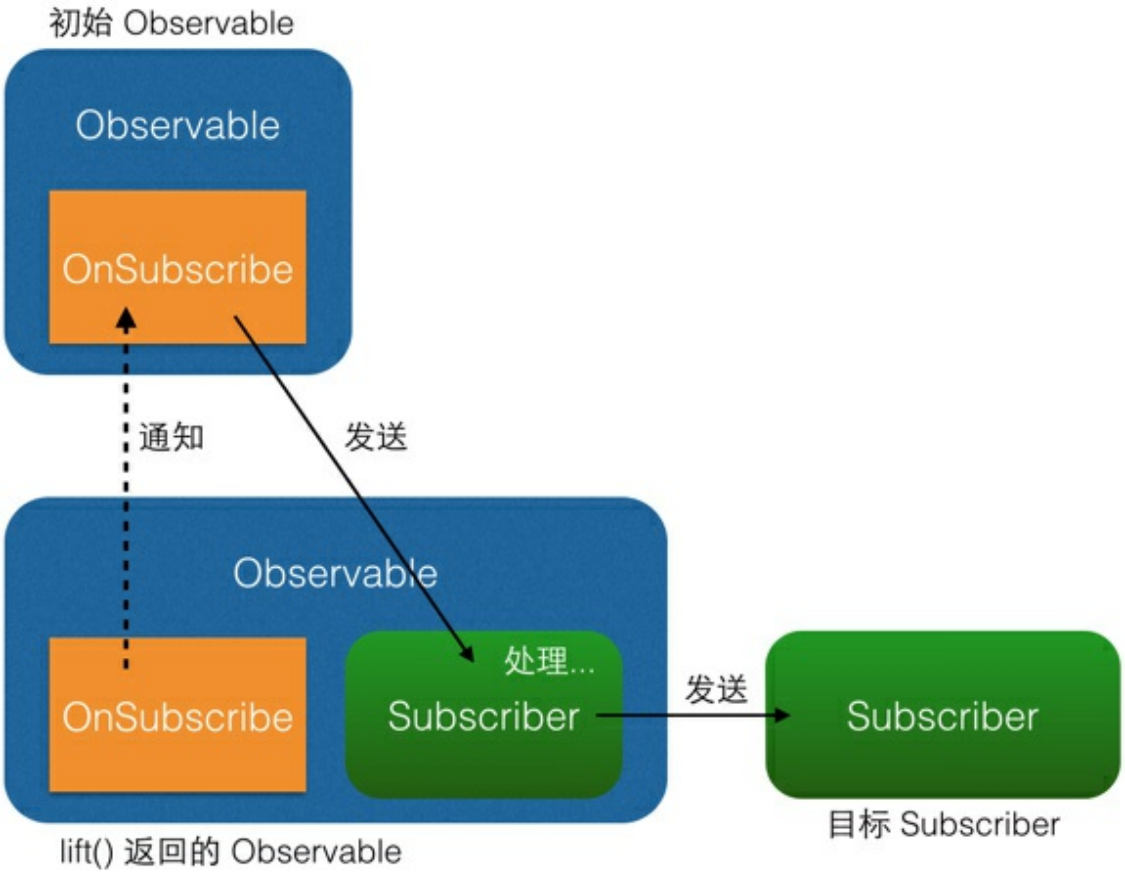
当含有 lift() 时：

1. lift() 创建了一个 Observable 后，加上之前的原始 Observable，已经有两个 Observable 了；
2. 而同样地，新 Observable 里的新 OnSubscribe 加上之前的原始 Observable 中的原始 OnSubscribe，也就有了两个 OnSubscribe；
3. 当用户调用经过 lift() 后的 Observable 的 subscribe() 的时候，使用的是 lift() 所返回的新的 Observable，于是它所触发的 onSubscribe.call(subscriber)，也是用的新 Observable 中的新 OnSubscribe，即在 lift() 中生成的那个 OnSubscribe；
4. 而这个新 OnSubscribe 的 call() 方法中的 onSubscribe，就是指的原始 Observable 中的原始 OnSubscribe，在这个 call() 方法里，新 OnSubscribe 利用 operator.call(subscriber) 生成了一个新的 Subscriber（Operator 就是在这里，通过自己的 call() 方法将新 Subscriber 和原始 Subscriber 进行关联，并插入自己的『变换』代码以实现变换），然后利用这个新 Subscriber 向原始 Observable 进行订阅。

   这样就实现了 lift() 过程，有点像一种代理机制，通过事件拦截和处理实现事件序列的变换。

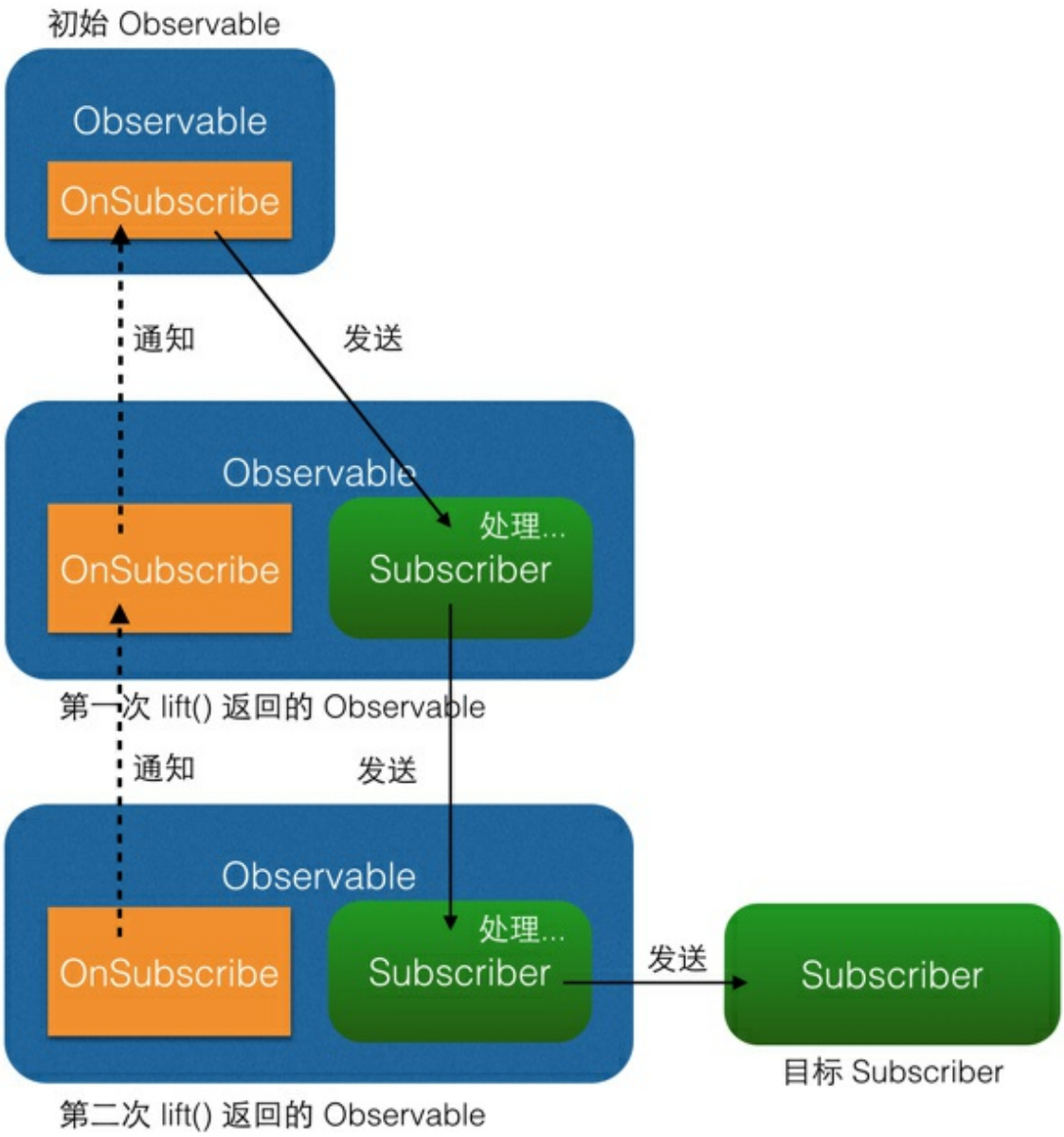
精简掉细节的话，也可以这么说：在 Observable 执行了 lift(Operator) 方法之后，会返回一个新的 Observable，这个新的 Observable 会像一个代理一样，负责接收原始的 Observable 发出的事件，并在处理后发送给 Subscriber。

如果你更喜欢具象思维，可以看图：



或者可以看动图：

两次和多次的 lift() 同理，如下图：



举一个具体的 Operator 的实现。下面这是一个将事件中的 Integer 对象转换成 String 的例子，仅供参考：

```
observable.lift(new Observable.Operator<String, Integer>() {
    @Override
    public Subscriber<? super Integer> call(final Subscriber<? super String> subscriber
) {
        // 将事件序列中的 Integer 对象转换为 String 对象
        return new Subscriber<Integer>() {
            @Override
            public void onNext(Integer integer) {
                subscriber.onNext("" + integer);
            }

            @Override
```

```
            public void onCompleted() {
                subscriber.onCompleted();
            }

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

> 讲述 lift() 的原理只是为了让你更好地了解 RxJava，从而可以更好地使用它。然而不管你是否理解了 lift() 的原理，RxJava 都不建议开发者自定义 Operator 来直接使用 lift()，而是建议尽量使用已有的 lift() 包装方法（如 map() flatMap() 等）进行组合来实现需求，因为直接使用 lift() 非常容易发生一些难以发现的错误。

3) compose: 对 Observable 整体的变换

除了 lift() 之外， Observable 还有一个变换方法叫做 compose(Transformer)。它和 lift() 的区别在于， lift() 是针对事件项和事件序列的，而 compose() 是针对 Observable 自身进行变换。举个例子，假设在程序中有多个 Observable，并且他们都需要应用一组相同的 lift() 变换。你可以这么写：

```
observable1
    .lift1()
    .lift2()
    .lift3()
    .lift4()
    .subscribe(subscriber1);
observable2
    .lift1()
    .lift2()
    .lift3()
    .lift4()
    .subscribe(subscriber2);
observable3
    .lift1()
    .lift2()
    .lift3()
    .lift4()
    .subscribe(subscriber3);
observable4
    .lift1()
    .lift2()
    .lift3()
```

```
    .lift4()
    .subscribe(subscriber1);
```

你觉得这样太不软件工程了，于是你改成了这样：

```
private Observable liftAll(Observable observable) {
    return observable
        .lift1()
        .lift2()
        .lift3()
        .lift4();
}
...
liftAll(observable1).subscribe(subscriber1);
liftAll(observable2).subscribe(subscriber2);
liftAll(observable3).subscribe(subscriber3);
liftAll(observable4).subscribe(subscriber4);
```

可读性、可维护性都提高了。可是 Observable 被一个方法包起来，这种方式对于 Observale 的灵活性似乎还是增添了那么点限制。怎么办？这个时候，就应该用 compose() 来解决了：

```
public class LiftAllTransformer implements Observable.Transformer<Integer, String> {
    @Override
    public Observable<String> call(Observable<Integer> observable) {
        return observable
            .lift1()
            .lift2()
            .lift3()
            .lift4();
    }
}
...
Transformer liftAll = new LiftAllTransformer();
observable1.compose(liftAll).subscribe(subscriber1);
observable2.compose(liftAll).subscribe(subscriber2);
observable3.compose(liftAll).subscribe(subscriber3);
observable4.compose(liftAll).subscribe(subscriber4);
```

像上面这样，使用 compose() 方法，Observable 可以利用传入的 Transformer 对象的 call 方法直接对自身进行处理，也就不必被包在方法的里面了。

compose() 的原理比较简单，不附图喽。

3.5 线程控制：Scheduler (二)

除了灵活的变换，RxJava 另一个牛逼的地方，就是线程的自由控制。

1) Scheduler 的 API (二)

前面讲到了，可以利用 subscribeOn() 结合 observeOn() 来实现线程控制，让事件的产生和消费发生在不同的线程。可是在了解了 map() flatMap() 等变换方法后，有些好事的（其实就是当初刚接触 RxJava 时的我）就问了：能不能多切换几次线程？

答案是：能。因为 observeOn() 指定的是 Subscriber 的线程，而这个 Subscriber 并不是（严格说应该为『不一定是』，但这里不妨理解为『不是』）subscribe() 参数中的 Subscriber，而是 observeOn() 执行时的当前 Observable 所对应的 Subscriber，即它的直接下级 Subscriber 。换句话说，observeOn() 指定的是它之后的操作所在的线程。因此如果有多次切换线程的需求，只要在每个想要切换线程的位置调用一次 observeOn() 即可。上代码：

```
Observable.just(1, 2, 3, 4) // IO 线程，由 subscribeOn() 指定
    .subscribeOn(Schedulers.io())
    .observeOn(Schedulers.newThread())
    .map(mapOperator) // 新线程，由 observeOn() 指定
    .observeOn(Schedulers.io())
    .map(mapOperator2) // IO 线程，由 observeOn() 指定
    .observeOn(AndroidSchedulers.mainThread)
    .subscribe(subscriber);  // Android 主线程，由 observeOn() 指定
```

如上，通过 observeOn() 的多次调用，程序实现了线程的多次切换。

不过，不同于 observeOn()， subscribeOn() 的位置放在哪里都可以，但它是只能调用一次的。

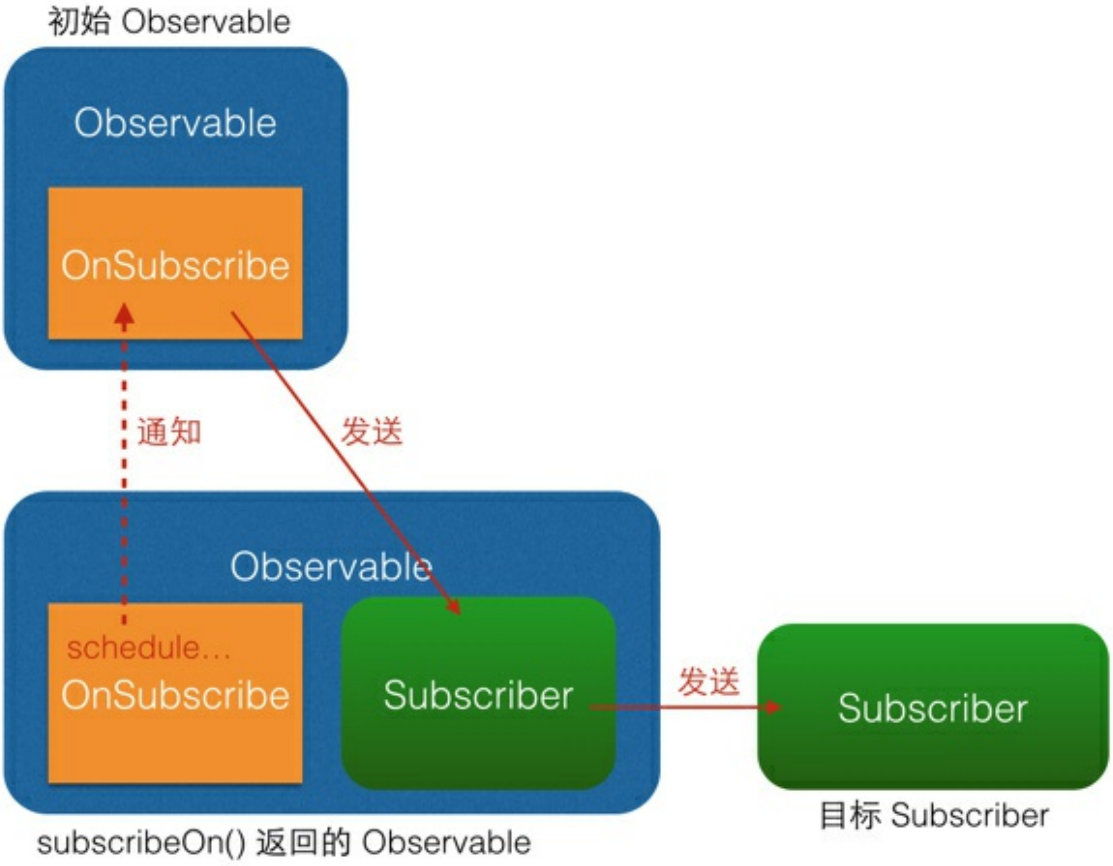
又有好事的（其实还是当初的我）问了：如果我非要调用多次 subscribeOn() 呢？会有什么效果？

这个问题先放着，我们还是从 RxJava 线程控制的原理说起吧。

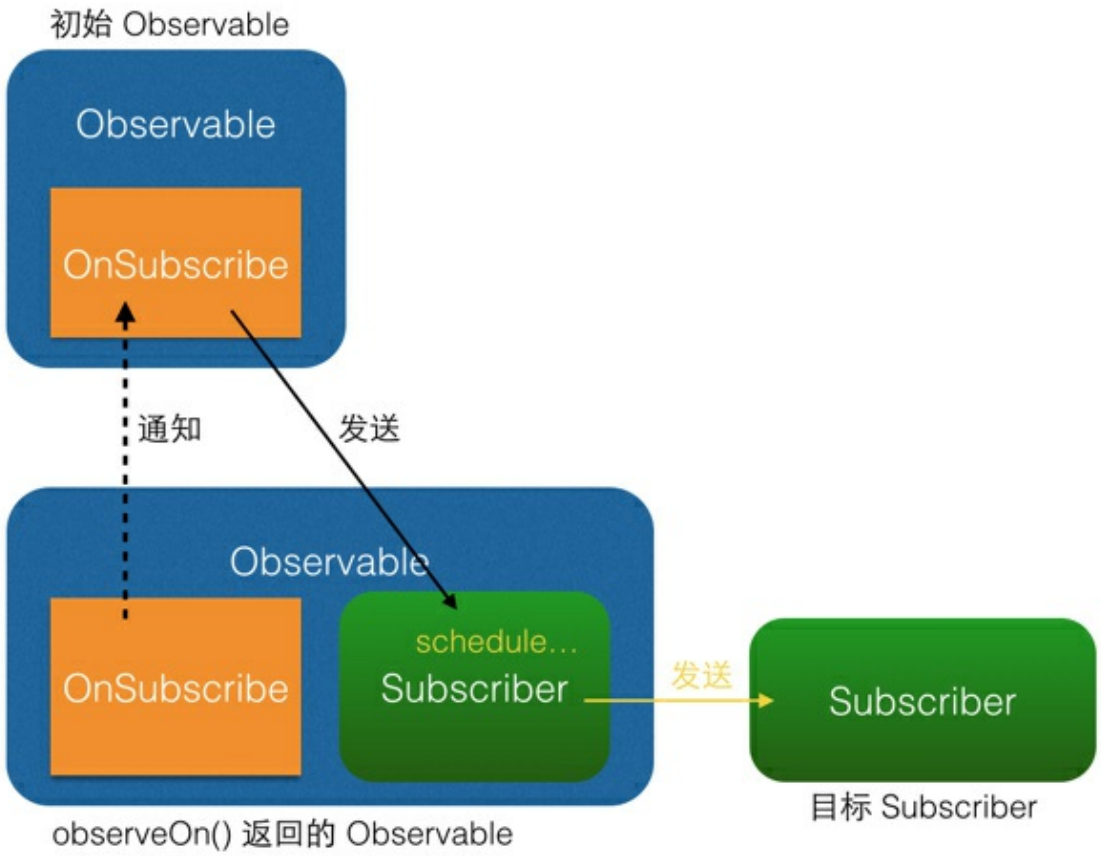
2) Scheduler 的原理（二）

其实， subscribeOn() 和 observeOn() 的内部实现，也是用的 lift()。具体看图（不同颜色的箭头表示不同的线程）：

subscribeOn() 原理图：

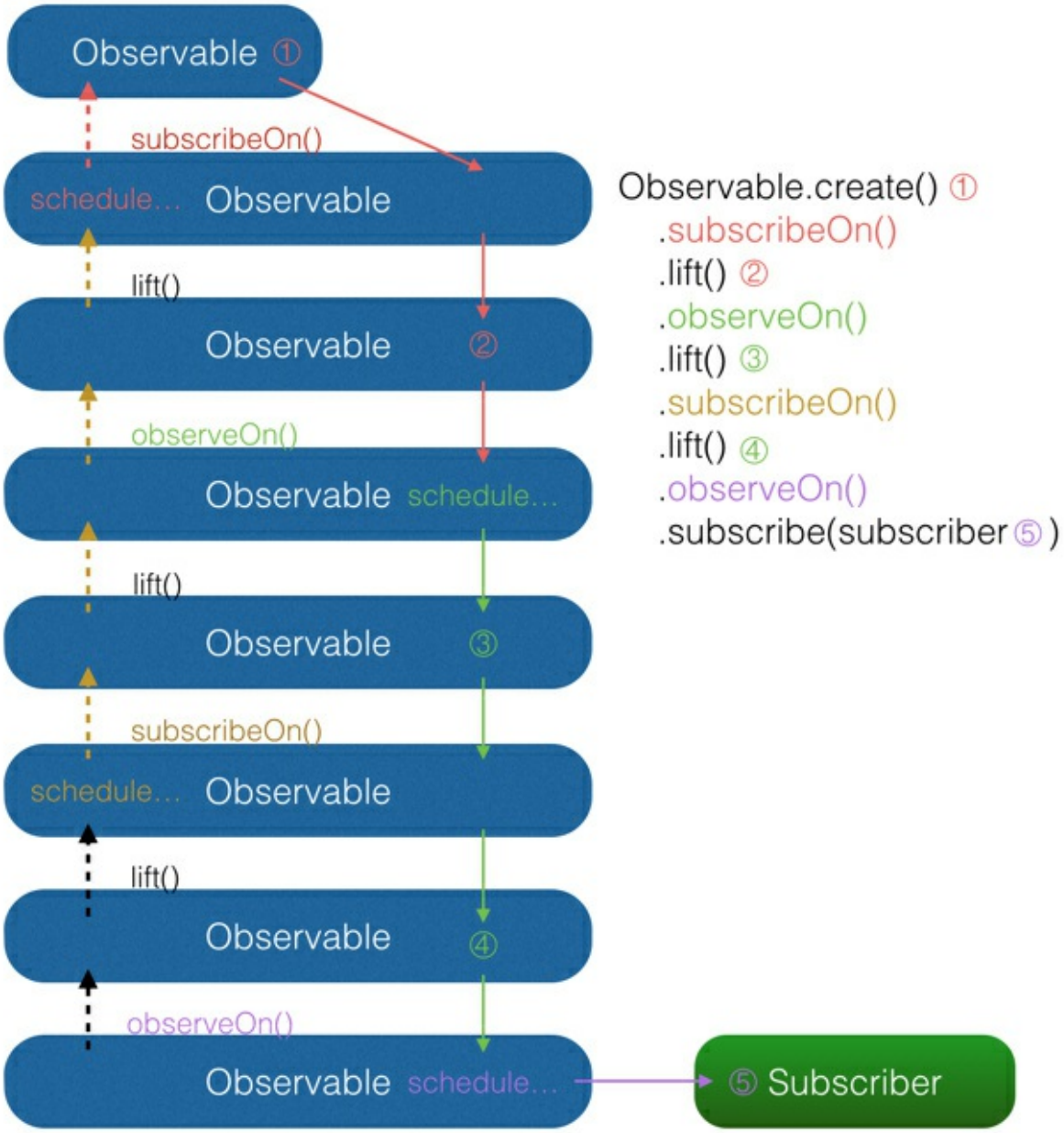


observeOn() 原理图：



从图中可以看出，subscribeOn() 和 observeOn() 都做了线程切换的工作（图中的 "schedule..." 部位）。不同的是， subscribeOn() 的线程切换发生在 OnSubscribe 中，即在它通知上一级 OnSubscribe 时，这时事件还没有开始发送，因此 subscribeOn() 的线程控制可以从事件发出的开端就造成影响；而 observeOn() 的线程切换则发生在它内建的 Subscriber 中，即发生在它即将给下一级 Subscriber 发送事件时，因此 observeOn() 控制的是它后面的线程。

最后，我用一张图来解释当多个 subscribeOn() 和 observeOn() 混合使用时，线程调度是怎么发生的（由于图中对象较多，相对于上面的图对结构做了一些简化调整）：



图中共有 5 处含有对事件的操作。由图中可以看出，①和②两处受第一个 subscribeOn() 影响，运行在红色线程；③和④处受第一个 observeOn() 的影响，运行在绿色线程；⑤处受第二个 onserveOn() 影响，运行在紫色线程；而第二个 subscribeOn()，由于在通知过程中线程就被第一个 subscribeOn() 截断，因此对整个流程并没有任何影响。这里也就回答了前面的问题：当使用了多个 subscribeOn() 的时候，只有第一个 subscribeOn() 起作用。

3) 延伸：doOnSubscribe()

然而，虽然超过一个的 subscribeOn() 对事件处理的流程没有影响，但在流程之前却是可以利用的。

在前面讲 Subscriber 的时候，提到过 Subscriber 的 onStart() 可以用作流程开始前的初始化。然而 onStart() 由于在 subscribe() 发生时就被调用了，因此不能指定线程，而是只能执行在 subscribe() 被调用时的线程。这就导致如果 onStart() 中含有对线程有要求的代码（例如在界面上显示一个

ProgressBar，这必须在主线程执行），将会有线程非法的风险，因为有时你无法预测 subscribe() 将会在什么线程执行。

而与 Subscriber.onStart() 相对应的，有一个方法 Observable.doOnSubscribe() 。它和 Subscriber.onStart() 同样是在 subscribe() 调用后而且在事件发送前执行，但区别在于它可以指定线程。默认情况下， doOnSubscribe() 执行在 subscribe() 发生的线程；而如果在 doOnSubscribe() 之后有 subscribeOn() 的话，它将执行在离它最近的 subscribeOn() 所指定的线程。

示例代码：

```
Observable.create(onSubscribe)
    .subscribeOn(Schedulers.io())
    .doOnSubscribe(new Action0() {
        @Override
        public void call() {
            progressBar.setVisibility(View.VISIBLE); // 需要在主线程执行
        }
    })
    .subscribeOn(AndroidSchedulers.mainThread()) // 指定主线程
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(subscriber);
```

如上，在 doOnSubscribe()的后面跟一个 subscribeOn()，就能指定准备工作的线程了。

4. RxJava 的适用场景和使用方式

4.1 与 Retrofit 的结合

> Retrofit 是 Square 的一个著名的网络请求库。没有用过 Retrofit 的可以选择跳过这一小节也没关系，我举的每种场景都只是个例子，而且例子之间并无前后关联，只是个抛砖引玉的作用，所以你跳过这里看别的场景也可以的。

Retrofit 除了提供了传统的 Callback 形式的 API，还有 RxJava 版本的 Observable 形式 API。下面我用对比的方式来介绍 Retrofit 的 RxJava 版 API 和传统版本的区别。

以获取一个 User 对象的接口作为例子。使用Retrofit 的传统 API，你可以用这样的方式来定义请求：

```
@GET("/user")
public void getUser(@Query("userId") String userId, Callback<User> callback);
```

在程序的构建过程中， Retrofit 会把自动把方法实现并生成代码，然后开发者就可以利用下面的方法来获取特定用户并处理响应：

```
getUser(userId, new Callback<User>() {
    @Override
```

```
    public void success(User user) {
        userView.setUser(user);
    }

    @Override
    public void failure(RetrofitError error) {
        // Error handling
        ...
    }
};
```

而使用 RxJava 形式的 API，定义同样的请求是这样的：

```
@GET("/user")
public Observable<User> getUser(@Query("userId") String userId);
使用的时候是这样的：

getUser(userId)
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<User>() {
        @Override
        public void onNext(User user) {
            userView.setUser(user);
        }

        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable error) {
            // Error handling
            ...
        }
    });
```

看到区别了吗？

当 RxJava 形式的时候，Retrofit 把请求封装进 Observable，在请求结束后调用 onNext() 或在请求失败后调用 onError()。

对比来看，Callback 形式和 Observable 形式长得不太一样，但本质都差不多，而且在细节上 Observable 形式似乎还比 Callback 形式要差点。那 Retrofit 为什么还要提供 RxJava 的支持呢？

因为它好用啊！从这个例子看不出来是因为这只是最简单的情况。而一旦情景复杂起来，Callback 形式马上就会开始让人头疼。比如：

假设这么一种情况：你的程序取到的 User 并不应该直接显示，而是需要先与数据库中的数据进行比对和修正后再显示。使用 Callback 方式大概可以这么写：

```
getUser(userId, new Callback<User>() {
    @Override
    public void success(User user) {
        processUser(user); // 尝试修正 User 数据
        userView.setUser(user);
    }

    @Override
    public void failure(RetrofitError error) {
        // Error handling
        ...
    }
};
```

有问题吗？

很简便，但不要这样做。为什么？因为这样做会影响性能。数据库的操作很重，一次读写操作花费 10~20ms 是很常见的，这样的耗时很容易造成界面的卡顿。所以通常情况下，如果可以的话一定要避免在主线程中处理数据库。所以为了提升性能，这段代码可以优化一下：

```
getUser(userId, new Callback<User>() {
    @Override
    public void success(User user) {
        new Thread() {
            @Override
            public void run() {
                processUser(user); // 尝试修正 User 数据
                runOnUiThread(new Runnable() { // 切回 UI 线程
                    @Override
                    public void run() {
                        userView.setUser(user);
                    }
                });
            }).start();
    }

    @Override
    public void failure(RetrofitError error) {
        // Error handling
        ...
    }
};
```

性能问题解决，但……这代码实在是太乱了，迷之缩进啊！杂乱的代码往往不仅仅是美观问题，因为代码越乱往往就越难读懂，而如果项目中充斥着杂乱的代码，无疑会降低代码的可读性，造成团队开发效率的降低和出错率的升高。

这时候，如果用 RxJava 的形式，就好办多了。 RxJava 形式的代码是这样的：

```
getUser(userId)
    .doOnNext(new Action1<User>() {
        @Override
        public void call(User user) {
            processUser(user);
        })
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<User>() {
        @Override
        public void onNext(User user) {
            userView.setUser(user);
        }

        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable error) {
            // Error handling
            ...
        }
    });
```

后台代码和前台代码全都写在一条链中，明显清晰了很多。

再举一个例子：假设 /user 接口并不能直接访问，而需要填入一个在线获取的 token，代码应该怎么写？

Callback 方式，可以使用嵌套的 Callback：

```
@GET("/token")
public void getToken(Callback<String> callback);

@GET("/user")
public void getUser(@Query("token") String token, @Query("userId") String userId, Callb
ack<User> callback);

...

getToken(new Callback<String>() {
    @Override
```

```
    public void success(String token) {
        getUser(token, userId, new Callback<User>() {
            @Override
            public void success(User user) {
                userView.setUser(user);
            }

            @Override
            public void failure(RetrofitError error) {
                // Error handling
                ...
            }
        };
    }

    @Override
    public void failure(RetrofitError error) {
        // Error handling
        ...
    }
});
```

倒是没有什么性能问题，可是迷之缩进毁一生，你懂我也懂，做过大项目的人应该更懂。

而使用 RxJava 的话，代码是这样的：

```
@GET("/token")
public Observable<String> getToken();

@GET("/user")
public Observable<User> getUser(@Query("token") String token, @Query("userId") String u
serId);

...

getToken()
    .flatMap(new Func1<String, Observable<User>>() {
        @Override
        public Observable<User> onNext(String token) {
            return getUser(token, userId);
        })
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Observer<User>() {
        @Override
        public void onNext(User user) {
            userView.setUser(user);
        }
```

```
        @Override
        public void onCompleted() {
        }

        @Override
        public void onError(Throwable error) {
            // Error handling
            ...
        }
    });
```

用一个 flatMap() 就搞定了逻辑，依然是一条链。看着就很爽，是吧？

2016/03/31 更新，加上我写的一个 Sample 项目： rengwuxian RxJava Samples

好，Retrofit 部分就到这里。

4.2 RxBinding

RxBinding 是 Jake Wharton 的一个开源库，它提供了一套在 Android 平台上的基于 RxJava 的 Binding API。所谓 Binding，就是类似设置 OnClickListener 、设置 TextWatcher 这样的注册绑定对象的 API。

举个设置点击监听的例子。使用 RxBinding ，可以把事件监听用这样的方法来设置：

```
Button button = ...;
RxView.clickEvents(button) // 以 Observable 形式来反馈点击事件
    .subscribe(new Action1<ViewClickEvent>() {
        @Override
        public void call(ViewClickEvent event) {
            // Click handling
        }
    });
```

看起来除了形式变了没什么区别，实质上也是这样。甚至如果你看一下它的源码，你会发现它连实现都没什么惊喜：它的内部是直接用一个包裹着的 setOnClickListener() 来实现的。然而，仅仅这一个形式的改变，却恰好就是 RxBinding 的目的：扩展性。通过 RxBinding 把点击监听转换成 Observable 之后，就有了对它进行扩展的可能。扩展的方式有很多，根据需求而定。一个例子是前面提到过的 throttleFirst() ，用于去抖动，也就是消除手抖导致的快速连环点击：

```
RxView.clickEvents(button)
    .throttleFirst(500, TimeUnit.MILLISECONDS)
    .subscribe(clickAction);
```

如果想对 RxBinding 有更多了解，可以去它的 GitHub 项目 下面看看。

4.3 各种异步操作

前面举的 Retrofit 和 RxBinding 的例子，是两个可以提供现成的 Observable 的库。而如果你有某些异步操作无法用这些库来自动生成 Observable，也完全可以自己写。例如数据库的读写、大图片的载入、文件压缩/解压等各种需要放在后台工作的耗时操作，都可以用 RxJava 来实现，有了之前几章的例子，这里应该不用再举例了。

4.4 RxBus

RxBus 名字看起来像一个库，但它并不是一个库，而是一种模式，它的思想是使用 RxJava 来实现了 EventBus，而让你不再需要使用 Otto 或者 GreenRobot 的 EventBus。至于什么是 RxBus，可以看这篇文章。顺便说一句，Flipboard 已经用 RxBus 替换掉了 Otto，目前为止没有不良反应。

最后

对于 Android 开发者来说，RxJava 是一个很难上手的库，因为它对于 Android 开发者来说有太多陌生的概念了。可是它真的很牛逼。因此，我写了这篇《给 Android 开发者的 RxJava 详解》，希望能给始终搞不明白什么是 RxJava 的人一些入门的指引，或者能让正在使用 RxJava 但仍然心存疑惑的人看到一些更深入的解析。无论如何，只要能给各位同为 Android 工程师的你们提供一些帮助，这篇文章的目的就达到了。

关于作者

朱凯（扔物线），Android 工程师。

- HenCoder：给高级 Android 工程师的进阶手册
- 微博：扔物线
- GitHub：rengwuxian

为什么写这个？

与两三年前的境况不同，中国现在已经不缺初级 Android 工程师，但中级和高级工程师严重供不应求。因此我决定从今天开始不定期地发布我的技术分享，只希望能够和大家共同提升，通过我们的成长来解决一点点国内互联网公司人才稀缺的困境，也提升各位技术党的收入。所以，不仅要写这篇，我还会写更多。至于内容的定位，我计划只定位真正的干货，一些边边角角的小技巧和炫酷的黑科技应该都不会写，总之希望每篇文章都能帮读者提升真正的实力。

> 原文链接：http://gank.io/post/56e80c2c677659311bed9841

前言

RxJava和Retrofit也火了一段时间了，不过最近一直在学习ReactNative和Node相关的姿势，一直没有时间研究这些新东西，最近有个项目准备写，打算先用Android写一个Demo出来，却发现Android的世界发生了天翻地覆的变化，EventBus和OKHttp啥的都不见了，RxJava和Retrofit是什么鬼？

好吧，到Github上耐着性子看过了RxJava和Retrofit的介绍和几个Demo，原来Android的大神Jake Wharton为Retrofit这个项目贡献了这么多的代码，没有道理不用了。

如果你对RxJava不熟悉请先看给 Android 开发者的 RxJava 详解这篇文章。

如果你对Retrofit不熟悉就先看Retrofit官网。

当然也有很多RxJava与Retrofit的文章，但是我觉得很多大家都很纠结的功能都没有被总结出来，所以才有了此篇文章。

欢迎大家拍砖。

接下来进入正文，我是从下面几个角度去思考RxJava与Retrofit结合的。

1. RxJava如何与Retrofit结合
2. 相同格式的Http请求数据该如何封装
3. 相同格式的Http请求数据统一进行预处理
4. 如何取消一个Http请求 -- 观察者之间的对决，Oberver VS Subscriber
5. 一个需要ProgressDialog的Subscriber该有的样子

1.RxJava如何与Retrofit结合

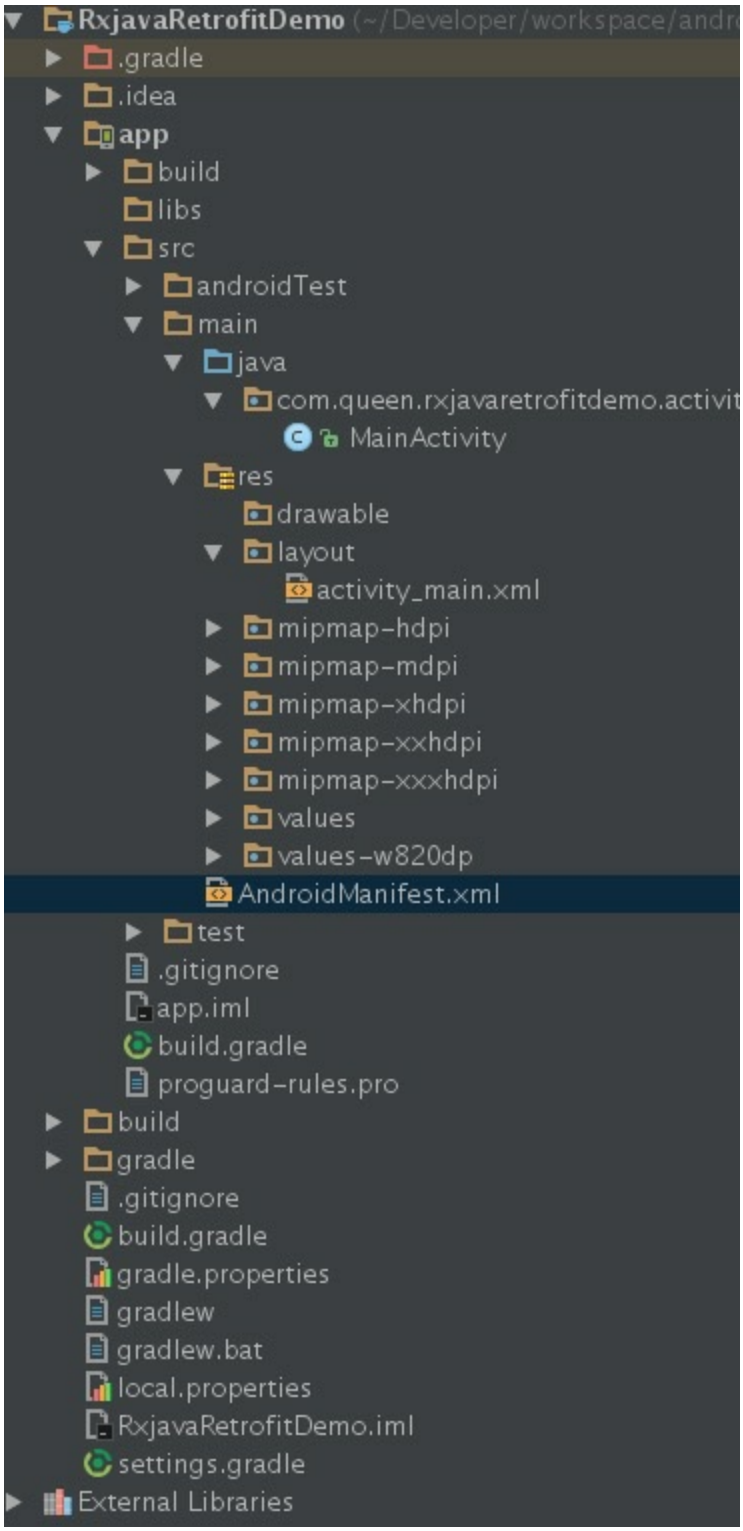
1.1 基本页面

先扔出build.gradle文件的内容

```
dependencies {
    compile fileTree(dir: 'libs', include: ['*.jar'])
    testCompile 'junit:junit:4.12'
    compile 'com.android.support:appcompat-v7:23.2.0'
    compile 'io.reactivex:rxjava:1.1.0'
    compile 'io.reactivex:rxandroid:1.1.0'
    compile 'com.squareup.retrofit2:retrofit:2.0.0-beta4'
    compile 'com.squareup.retrofit2:converter-gson:2.0.0-beta4'
    compile 'com.squareup.retrofit2:adapter-rxjava:2.0.0-beta4'
    compile 'com.google.code.gson:gson:2.6.2'
    compile 'com.jakewharton:butterknife:7.0.1'
}
```

也就是说本文是基于RxJava1.1.0和Retrofit 2.0.0-beta4来进行的。 添加rxandroid是因为rxjava中的线程问题。

下面先搭建一个基本的页面，页面很简单，先来看文件目录结构



activity_main.xml的代码如下：

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".activity.MainActivity">

    <Button
        android:id="@+id/click_me_BN"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_alignParentBottom="true"
        android:padding="5dp"
        android:text="点我"
        android:textSize="16sp"/>
    <TextView
        android:id="@+id/result_TV"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_above="@id/click_me_BN"
        android:text="Hello World!"
        android:textSize="16sp"/>
</RelativeLayout>
```

MainActivity.java的代码如下：

```
package com.queen.rxjavaretrofitdemo.activity;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.widget.Button;
import android.widget.TextView;

import com.queen.rxjavaretrofitdemo.R;

import butterknife.Bind;
import butterknife.ButterKnife;
import butterknife.OnClick;

public class MainActivity extends AppCompatActivity {

    @Bind(R.id.click_me_BN)
    Button clickMeBN;
    @Bind(R.id.result_TV)
```

```
    TextView resultTV;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        ButterKnife.bind(this);
    }

    @OnClick(R.id.click_me_BN)
    public void onClick() {
        getMovie();
    }

    //进行网络请求
    private void getMovie(){

    }
}
```

注意不要忘记加网络权限

```
<uses-permission android:name="android.permission.INTERNET"/>
```

1.2 只用Retrofit

我们准备在**getMovie**方法中进行网络请求，我们先来看看只使用**Retrofit**是如何进行的。

我们使用豆瓣电影的Top250做测试连接，目标地址为

> https://api.douban.com/v2/movie/top250?start=0&count=10

至于返回的数据格式，大家自己访问下链接就看到了，太长就不放进来了。

首先我们要根据返回的结果封装一个Entity，暂命名为**MovieEntity**，代码就不贴了。

接下来我们要创建一个接口取名为**MovieService**，代码如下：

```
public interface MovieService {
    @GET("top250")
    Call<MovieEntity> getTopMovie(@Query("start") int start, @Query("count") int count)
;
}
```

回到**MainActivity**之中，我们来写**getMovie**方法的代码

```
//进行网络请求
```

```
private void getMovie(){
    String baseUrl = "https://api.douban.com/v2/movie/";

    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(baseUrl)
            .addConverterFactory(GsonConverterFactory.create())
            .build();

    MovieService movieService = retrofit.create(MovieService.class);
    Call<MovieEntity> call = movieService.getTopMovie(0, 10);
    call.enqueue(new Callback<MovieEntity>() {
        @Override
        public void onResponse(Call<MovieEntity> call, Response<MovieEntity> response)
{
            resultTV.setText(response.body().toString());
        }

        @Override
        public void onFailure(Call<MovieEntity> call, Throwable t) {
            resultTV.setText(t.getMessage());
        }
    });
}
```

以上为没有经过封装的、原生态的Retrofit写网络请求的代码。

我们可以封装创建Retrofit和service部分的代码，然后Activity用创建一个Callback作为参数给Call，这样Activity中只关注请求的结果，而且Call有cancel方法可以取消一个请求，好像没Rxjava什么事了，我觉得可以写到这就下班了

接下来我们要面对的问题是这样的 如果我的Http返回数据是一个统一的格式，例如

```
{
    "resultCode": 0,
    "resultMessage": "成功",
    "data": {}
}
```

我们如何对返回结果进行一个统一的处理呢？

另外，我的ProgressDialog的show方法应该在哪调用呢？看样子只能在getMovie()这个方法里面调用了，换个地方发出请求就要在对应的Listener里面写一遍show()的代码，其实挺闹心。

而且错误请求我也想集中处理掉不要贴重复的代码。

我们先来看结合了Rxjava之后，事情有没有变化的可能。当然即便是不用Rxjava，依旧能够做很多的封装，只是比较麻烦。

如需查看项目代码 --> 代码地址:

> https://github.com/tough1985/RxjavaRetrofitDemo

选择Tag -> step1

1.3 添加Rxjava

Retrofit本身对Rxjava提供了支持。

添加Retrofit对Rxjava的支持需要在**Gradle**文件中添加 `compile 'com.squareup.retrofit2:adapter-rxjava:2.0.0-beta4'` 当然我们已经添加过了。

然后在创建Retrofit的过程中添加如下代码：

```
Retrofit retrofit = new Retrofit.Builder()
        .baseUrl(baseUrl)
        .addConverterFactory(GsonConverterFactory.create())
        .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
        .build();
```

这样一来我们定义的service返回值就不在是一个**Call**了，而是一个**Observable**

重新定义**MovieService**

```
public interface MovieService {
    @GET("top250")
    Observable<MovieEntity> getTopMovie(@Query("start") int start, @Query("count") int
count);
}
```

getMovie方法改为：

```
//进行网络请求
private void getMovie(){
    String baseUrl = "https://api.douban.com/v2/movie/";

    Retrofit retrofit = new Retrofit.Builder()
            .baseUrl(baseUrl)
            .addConverterFactory(GsonConverterFactory.create())
            .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
            .build();

    MovieService movieService = retrofit.create(MovieService.class);

    movieService.getTopMovie(0, 10)
            .subscribeOn(Schedulers.io())
```

```
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Subscriber<MovieEntity>() {
                @Override
                public void onCompleted() {
                    Toast.makeText(MainActivity.this, "Get Top Movie Completed", Toast.
LENGTH_SHORT).show();
                }

                @Override
                public void onError(Throwable e) {
                    resultTV.setText(e.getMessage());
                }

                @Override
                public void onNext(MovieEntity movieEntity) {
                    resultTV.setText(movieEntity.toString());
                }
            });
}
```

这样基本上就完成了Retrofit和Rxjava的结合，但是我知道你们当然不会满意的。

接下来我们把创建**Retrofit**的过程封装一下，然后希望Activity创建**Subscriber**对象传进来。

如需查看项目代码 --> 代码地址:

> https://github.com/tough1985/RxjavaRetrofitDemo

选择Tag -> step2

1.4 将请求过程进行封装

创建一个对象**HttpMethods**

```
public class HttpMethods {

    public static final String BASE_URL = "https://api.douban.com/v2/movie/";

    private static final int DEFAULT_TIMEOUT = 5;

    private Retrofit retrofit;
    private MovieService movieService;

    //构造方法私有
    private HttpMethods() {
        //手动创建一个OkHttpClient并设置超时时间
        OkHttpClient.Builder httpClientBuilder = new OkHttpClient.Builder();
        httpClientBuilder.connectTimeout(DEFAULT_TIMEOUT, TimeUnit.SECONDS);
```

```
        retrofit = new Retrofit.Builder()
                .client(httpClientBuilder.build())
                .addConverterFactory(GsonConverterFactory.create())
                .addCallAdapterFactory(RxJavaCallAdapterFactory.create())
                .baseUrl(BASE_URL)
                .build();

        movieService = retrofit.create(MovieService.class);
    }

    //在访问HttpMethods时创建单例
    private static class SingletonHolder{
        private static final HttpMethods INSTANCE = new HttpMethods();
    }

    //获取单例
    public static HttpMethods getInstance(){
        return SingletonHolder.INSTANCE;
    }

    /**
     * 用于获取豆瓣电影Top250的数据
     * @param subscriber  由调用者传过来的观察者对象
     * @param start 起始位置
     * @param count 获取长度
     */
    public void getTopMovie(Subscriber<MovieEntity> subscriber, int start, int count){
        movieService.getTopMovie(start, count)
                .subscribeOn(Schedulers.io())
                .unsubscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(subscriber);
    }
}
```

用一个单例来封装该对象，在构造方法中创建Retrofit和对应的Service。 如果需要访问不同的基地址，那么你可能需要创建多个Retrofit对象，或者干脆根据不同的基地址封装不同的HttpMethod类。

我们回头再来看MainActivity中的**getMovie**方法：

```
private void getMovie(){
     subscriber = new Subscriber<MovieEntity>() {
         @Override
         public void onCompleted() {
             Toast.makeText(MainActivity.this, "Get Top Movie Completed", Toast.LENGTH_
SHORT).show();
```

```
        }

        @Override
        public void onError(Throwable e) {
            resultTV.setText(e.getMessage());
        }

        @Override
        public void onNext(MovieEntity movieEntity) {
            resultTV.setText(movieEntity.toString());
        }
    };
    HttpMethods.getInstance().getTopMovie(subscriber, 0, 10);
}
```

其中subscriber是MainActivity的成员变量。

如需查看项目代码 --> 代码地址:

> https://github.com/tough1985/RxjavaRetrofitDemo

选择Tag -> step3

2.相同格式的Http请求数据该如何封装

第二部分和第三部分我参考了知乎上的一个问答： RxJava+Retrofit，在联网返回后如何先进行统一的判断？ 不过没有完整的示例，所以在这写一个完整的示例出来。

这个段落我们来聊一下有些Http服务返回一个固定格式的数据的问题。 例如：

```
{
    "resultCode": 0,
    "resultMessage": "成功",
    "data": {}
}
```

大部分的Http服务可能都是这样设置，resultCode和resultMessage的内容相对比较稳定，而data的内容变化多端，72变都不一定够变的，有可能是个User对象，也有可能是个订单对象，还有可能是个订单列表。 按照我们之前的用法，使用Gson转型需要我们在创建subscriber对象是指定返回值类型，如果我们对不同的返回值进行封装的话，那可能就要有上百个Entity了，看着明明是很清晰的结构，却因为data的不确定性无奈了起来。

少年，不必烦恼，来来来~ 老衲赐你宝典葵花，老衲就是练了这个才出家。。。

我们可以创建一个HttpResult类

```
public class HttpResult<T> {
```

```
    private int resultCode;
    private String resultMessage;

    private T data;
}
```

如果data是一个User对象的话。那么在定义Service方法的返回值就可以写为

```
Observable<HttpResult<User>>
```

这样一来HttpResult就相当于一个包装类，将结果包装了起来，但是在使用的时候要给出一个明确的类型。

在上面的示例中，我也创建了一个HttpResult类，用来模仿这个形式，将其中的Subject单独封装了起来。

```
public class HttpResult<T> {

    //用来模仿resultCode和resultMessage
    private int count;
    private int start;
    private int total;
    private String title;

    //用来模仿Data
    private T subjects;
}
```

这样泛型的时候就要写为：

```
Observable<HttpResult<List<Subject>>>
```

如需查看项目代码 --> 代码地址:

> https://github.com/tough1985/RxjavaRetrofitDemo

选择Tag -> step4

3.相同格式的Http请求数据统一进行预处理

既然我们有了相同的返回格式，那么我们可能就需要在获得数据之后进行一个统一的预处理。

当接收到了一个Http请求结果之后，由于返回的结构统一为

```
{
```

```
    "resultCode": 0,
    "resultMessage": "成功",
    "data": {}
}
```

我们想要对**resultCode**和**resultMessage**先做一个判断，因为如果 `resultCode == 0` 代表**success**，那么 `resultCode != 0` 时**data**一般都是**null**。

Activity或Fragment对**resultCode**和**resultMessage**基本没有兴趣，他们只对**请求状态**和**data**数据感兴趣。

基于这种考虑，我们在 `resultCode != 0` 的时候，抛出个自定义的**ApiException**。这样就会进入到subscriber的onError中，我们可以在onError中处理错误信息。

另外，请求成功时，需要将data数据转换为目标数据类型传递给subscriber，因为，Activity和Fragment只想拿到和他们真正相关的数据。

使用Observable的map方法可以完成这一功能。

在**HttpMethods**中创建一个内部类**HttpResultFunc**，代码如下：

```
/**
 * 用来统一处理Http的resultCode,并将HttpResult的Data部分剥离出来返回给subscriber
 *
 * @param <T>   Subscriber真正需要的数据类型，也就是Data部分的数据类型
 */
private class HttpResultFunc<T> implements Func1<HttpResult<T>, T>{

    @Override
    public T call(HttpResult<T> httpResult) {
        if (httpResult.getResultCode() != 0) {
            throw new ApiException(httpResult.getResultCode());
        }
        return httpResult.getData();
    }
}
```

然后我们的**getTopMovie**方法改为：

```
public void getTopMovie(Subscriber<List<Subject>> subscriber, int start, int count){

    movieService.getTopMovie(start, count)
            .map(new HttpResultFunc<List<Subject>>())
            .subscribeOn(Schedulers.io())
            .unsubscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(subscriber);
}
```

由于HttpResult中的泛型T就是我们希望传递给subscriber的数据类型，而数据可以通过httpResult的getData方法获得，这样我们就处理了泛型问题，错误处理问题，还有将请求数据部分剥离出来给subscriber

这样我们只需要关注Data数据的类型，而不必在关心整个过程了。

需要注意一点，就是在定义Service的时候，泛型是

```
HttpResult<User>
//or
HttpResult<List<Subject>>
```

而在定义Subscriber的时候泛型是

```
User
//or
List<Subject>
```

不然你会得到一个转型错误。

如需查看项目代码 --> 代码地址:

> https://github.com/tough1985/RxjavaRetrofitDemo

选择Tag -> step5

代码中我是用豆瓣数据模拟了HttpResult中的resultCode和resultMessage，与文档中的代码略有出入。

4.如何取消一个Http请求 -- 观察者之间的对决，Observer VS Subscriber

4.1 取消一个Http请求

这一部分我们来聊一下关于取消Http请求的事情，已经Oberver和Subscriber这两个体位我们哪个更容易给我们G点。

如果没有使用Rxjava，那么Service返回的是一个Call，而这个Call对象有一个cancel方法可以用来取消Http请求。那么用了Rxjava之后，如何来取消一个请求呢？因为返回值是一个Observable。我们能做的视乎之有解除对Observable对象的订阅，其他的什么也做不了。

好在Retrofit已经帮我们考虑到了这一点。 答案在RxJavaCallAdapterFactory这个类的源码中可以找到

```
static final class CallOnSubscribe<T> implements Observable.OnSubscribe<Response<T>> {
```

```
  private final Call<T> originalCall;

  CallOnSubscribe(Call<T> originalCall) {
    this.originalCall = originalCall;
  }

  @Override public void call(final Subscriber<? super Response<T>> subscriber) {
    // Since Call is a one-shot type, clone it for each new subscriber.
    final Call<T> call = originalCall.clone();

    // Attempt to cancel the call if it is still in-flight on unsubscription.
    subscriber.add(Subscriptions.create(new Action0() {
      @Override public void call() {
        call.cancel();
      }
    }));

    try {
      Response<T> response = call.execute();
      if (!subscriber.isUnsubscribed()) {
        subscriber.onNext(response);
      }
    } catch (Throwable t) {
      Exceptions.throwIfFatal(t);
      if (!subscriber.isUnsubscribed()) {
        subscriber.onError(t);
      }
      return;
    }

    if (!subscriber.isUnsubscribed()) {
      subscriber.onCompleted();
    }
  }
}
```

我们看到call方法中，给subscriber添加了一个Subscription对象，Subscription对象很简单，主要就是取消订阅用的，如果你查看Subscriptions.create的源码，发现是这样的

```
public static Subscription create(final Action0 unsubscribe) {
     return BooleanSubscription.create(unsubscribe);
}
```

利用了一个BooleanSubscription类来创建一个Subscription，如果你点进去看BooleanSubscription.create方法一切就清晰了，当接触绑定的时候，subscriber会调用Subscription的unsubscribe方法，然后触发创建Subscription时候的传递进来的Action0的call方法。

RxJavaCallAdapterFactory帮我们给subscriber添加的是call.cancel(),

总结起来就是说，我们在Activity或者Fragment中创建subscriber对象，想要取消请求的时候调用subscriber的unsubscribe方法就可以了。

对不起这一节有太多的**Subscriber**和**Subscription**以及**Observer**和**Observable**，老衲当时看的时候也是不知道吐了多少次了，习惯了就好了。

4.2 为什么会提到Oberver

提到Observer的过程是这样的。由于Subscriber一旦调用了unsubscribe方法之后，就没有用了。且当事件传递到onError或者onCompleted之后，也会自动的解绑。这样出现的一个问题就是每次发送请求都要创建新的Subscriber对象。

这样我们就把注意力放到了Observer，Observer本身是一个接口，他的特性是不管你怎么用，都不会解绑，为什么呢？因为他没有解绑的方法。所以就达到了复用的效果，一开始我一直美滋滋的用Observer。事实上，如果你用的是Observer，在调用Observable对象的subscribe方法的时候，会自动的将Observer对象转换成Subscriber对象。

下面是源码：

```
public final Subscription subscribe(final Observer<? super T> observer) {
    if (observer instanceof Subscriber) {
        return subscribe((Subscriber<? super T>)observer);
    }
    return subscribe(new Subscriber<T>() {

        @Override
        public void onCompleted() {
            observer.onCompleted();
        }

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

        @Override
        public void onNext(T t) {
            observer.onNext(t);
        }

    });
}
```

后来发现了问题，

> 问题1 无法取消，因为Observer没有unsubscribe方法
>
> 问题2 没有onStart方法 这个一会聊

这两个问题是很痛苦的。所以，为了后面更好的高潮，我们还是选择用Subscriber。

5.一个需要ProgressDialog的Subscriber该有的样子

我们希望有一个Subscriber在我们每次发送请求的时候能够弹出一个ProgressDialog，然后在请求接受的时候让这个ProgressDialog消失，同时在我们取消这个ProgressDialog的同时能够取消当前的请求，而我们只需要处理里面的数据就可以了。

我们先来创建一个类，就叫**ProgressSubscriber**，让他继承**Subscriber**。

Subscriber给我们提供了onStart、onNext、onError、onCompleted四个方法。

其中只有onNext方法返回了数据，那我们自然希望能够在onNext里面处理数据相关的逻辑。

onStart方法我们用来启动一个ProgressDialog。 **onError**方法我们集中处理错误，同时也停止ProgressDialog **onComplated**方法里面停止ProgressDialog

其中我们需要解决两个问题

> 问题1 onNext的处理
>
> 问题2 cancel掉一个ProgressDialog的时候取消请求

我们先来解决问题1

5.1处理onNext

我们希望这里能够让Activity或者Fragment自己处理onNext之后的逻辑，很自然的我们想到了用接口。问题还是泛型的问题，这里面我们必须指定明确的类型。所以接口还是需要泛型。

我们先来定义一个接口，命名**SubscriberOnNextListener**

```
public interface SubscriberOnNextListener<T> {
    void onNext(T t);
}
```

代码很简单。再来看一下ProgressSubscriber现在的代码

```
public class ProgressSubscriber<T> extends Subscriber<T> {

    private SubscriberOnNextListener mSubscriberOnNextListener;
    private Context context;

    public ProgressSubscriber(SubscriberOnNextListener mSubscriberOnNextListener, Conte
```

```
xt context) {
        this.mSubscriberOnNextListener = mSubscriberOnNextListener;
        this.context = context;
    }

    @Override
    public void onStart() {
    }

    @Override
    public void onCompleted() {
        Toast.makeText(context, "Get Top Movie Completed", Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onError(Throwable e) {
        Toast.makeText(context, "error:" + e.getMessage(), Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onNext(T t) {
        mSubscriberOnNextListener.onNext(t);
    }
}
```

我知道传Context不好，不过为了演示而已，大家可以自己封装一下Toast。

MainActivity使用是这样的：

先来定义一个SubscriberOnNextListener对象，可以在onCreate里面创建这个对象

```
private SubscriberOnNextListener getTopMovieOnNext;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    ButterKnife.bind(this);

    getTopMovieOnNext = new SubscriberOnNextListener<List<Subject>>() {
        @Override
        public void onNext(List<Subject> subjects) {
            resultTV.setText(subjects.toString());
        }
    };
}
```

getMovie方法这么写：

```
private void getMovie(){
    HttpMethods.getInstance().getTopMovie(
        new ProgressSubscriber(getTopMovieOnNext, MainActivity.this),
        0, 10);
}
```

这样Activity或Fragment就只需要关注拿到结果之后的逻辑了，其他的完全不用操心。

如需查看项目代码 --> 代码地址:

> https://github.com/tough1985/RxjavaRetrofitDemo

选择Tag -> step6

5.2处理ProgressDialog

我们希望当cancel掉ProgressDialog的时候，能够取消订阅，也就取消了当前的Http请求。 所以我们先来创建个接口来处理这件事情。

```
public interface ProgressCancelListener {
    void onCancelProgress();
}
```

然后我们用ProgressSubscriber来实现这个接口，这样ProgressSubscriber就有了一个onCancelProgress方法，在这里面取消订阅。

```
@Override
public void onCancelProgress() {
    if (!this.isUnsubscribed()) {
        this.unsubscribe();
    }
}
```

然后我用了一个Handler来封装了ProgressDialog。

```
public class ProgressDialogHandler extends Handler {

    public static final int SHOW_PROGRESS_DIALOG = 1;
    public static final int DISMISS_PROGRESS_DIALOG = 2;

    private ProgressDialog pd;

    private Context context;
    private boolean cancelable;
    private ProgressCancelListener mProgressCancelListener;
```

```
    public ProgressDialogHandler(Context context, ProgressCancelListener mProgressCance
lListener,
                                 boolean cancelable) {
        super();
        this.context = context;
        this.mProgressCancelListener = mProgressCancelListener;
        this.cancelable = cancelable;
    }

    private void initProgressDialog(){
        if (pd == null) {
            pd = new ProgressDialog(context);

            pd.setCancelable(cancelable);

            if (cancelable) {
                pd.setOnCancelListener(new DialogInterface.OnCancelListener() {
                    @Override
                    public void onCancel(DialogInterface dialogInterface) {
                        mProgressCancelListener.onCancelProgress();
                    }
                });
            }

            if (!pd.isShowing()) {
                pd.show();
            }
        }
    }

    private void dismissProgressDialog(){
        if (pd != null) {
            pd.dismiss();
            pd = null;
        }
    }

    @Override
    public void handleMessage(Message msg) {
        switch (msg.what) {
            case SHOW_PROGRESS_DIALOG:
                initProgressDialog();
                break;
            case DISMISS_PROGRESS_DIALOG:
                dismissProgressDialog();
                break;
        }
    }
}
```

Handler接收两个消息来控制显示Dialog还是关闭Dialog。 创建Handler的时候我们需要传入ProgressCancelListener的对象实例。

最后贴出ProgressSubscriber的完整代码：

```
public class ProgressSubscriber<T> extends Subscriber<T> implements ProgressCancelListe
ner{

    private SubscriberOnNextListener mSubscriberOnNextListener;
    private ProgressDialogHandler mProgressDialogHandler;

    private Context context;

    public ProgressSubscriber(SubscriberOnNextListener mSubscriberOnNextListener, Conte
xt context) {
        this.mSubscriberOnNextListener = mSubscriberOnNextListener;
        this.context = context;
        mProgressDialogHandler = new ProgressDialogHandler(context, this, true);
    }

    private void showProgressDialog(){
        if (mProgressDialogHandler != null) {
            mProgressDialogHandler.obtainMessage(ProgressDialogHandler.SHOW_PROGRESS_DI
ALOG).sendToTarget();
        }
    }

    private void dismissProgressDialog(){
        if (mProgressDialogHandler != null) {
            mProgressDialogHandler.obtainMessage(ProgressDialogHandler.DISMISS_PROGRESS
_DIALOG).sendToTarget();
            mProgressDialogHandler = null;
        }
    }

    @Override
    public void onStart() {
        showProgressDialog();
    }

    @Override
    public void onCompleted() {
        dismissProgressDialog();
        Toast.makeText(context, "Get Top Movie Completed", Toast.LENGTH_SHORT).show();
    }

    @Override
```

```
    public void onError(Throwable e) {
        dismissProgressDialog();
        Toast.makeText(context, "error:" + e.getMessage(), Toast.LENGTH_SHORT).show();
    }

    @Override
    public void onNext(T t) {
        mSubscriberOnNextListener.onNext(t);
    }

    @Override
    public void onCancelProgress() {
        if (!this.isUnsubscribed()) {
            this.unsubscribe();
        }
    }
}
```

目前为止，就封装完毕了。以上是我在用Rxjava和Retrofit过程中踩过的一些坑，最后整合出来的，由于没有在实际的项目中跑过，有问题的话希望能够提出来大家讨论一下，拍砖也欢迎。

现在我们再写一个新的网络请求，步骤是这样的：

1. 在**Service**中定义一个新的方法。
2. 在**HttpMethods**封装对应的请求（代码基本可以copy）
3. 创建一个**SubscriberOnNextListener**处理请求数据并刷新UI。

最后

如果你觉得写更改线程的代码觉得也很烦的话，可以把订阅这部分也封装起来：

```
public void getTopMovie(Subscriber<List<Subject>> subscriber, int start, int count){
        //原来的样子
//        movieService.getTopMovie(start, count)
//                .map(new HttpResultFunc<List<Subject>>())
//                .subscribeOn(Schedulers.io())
//                .unsubscribeOn(Schedulers.io())
//                .observeOn(AndroidSchedulers.mainThread())
//                .subscribe(subscriber);

    //修改之后的样子
    Observable observable = movieService.getTopMovie(start, count)
            .map(new HttpResultFunc<List<Subject>>());

    toSubscribe(observable, subscriber);
}
```

```
//添加线程管理并订阅
private void toSubscribe(Observable o, Subscriber s){
     o.subscribeOn(Schedulers.io())
            .unsubscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(s);
}
```

让你每次写一个请求的时候，写的代码尽量少，更多的精力放在业务逻辑本身。

最后的最后

如果你的httpResult格式本身没有问题，但是data中的内容是这样的：

```
{
    "resultCode": 0,
    "resultMessage": "成功",
    "data": {"user": {}, "orderArray": []}
}
```

这样的情况还能不能继续使用这样的框架呢？我的解决方法是封装一个类，把user和orderArray作为类的属性。但是如果你的服务器一会data本身是一个完整的user数据，一会又是这样：`"data": {"user": {}, "orderArray": []}` 那我觉得你有必要跟你的服务端好好聊聊了，请他吃顿饭和顿酒，大不了献出菊花就是了。

但是如果服务已经上线了！！！

对不起，骚年……

老衲会在你坟前念300遍**Thinking in java**替你超度的

希望你用Retrofit和Rxjava的新体位能够享受到新的高潮。

前言

上个月RxJava2正式版发布了，但目前国内的资料还比较少，以前使用过RxJava1只需要看看更新文档就知道怎么使用了，但还有一些以前没用过RxJava的朋友可能就不知道怎么办了，不知道该看RxJava1还是直接跳到RxJava2。所以写下这个教程， 帮助那些没有用过RxJava的朋友入门。

> 注：如果你觉得写得不好，请直接批评指出。

我先回答这个问题：学习RxJava2需要先学习RxJava1吗？

这个问题就像论坛经常问学Java前需要先学习C语言吗，这里就不引战了！（PHP是世界上最好的语言！！）

答案明显不是，如果你以前学过RxJava1，那么对于RxJava2只需要看看更新了哪些东西就行了，其最核心的思想并没有变化，如果你没学过RxJava1，没有关系，直接学习RxJava2。所以作为一个RxJava2的教程，本文中所有的名词都属于RxJava2中，并不涉及RxJava1。

要在Android中使用RxJava2, 先添加Gradle配置:

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

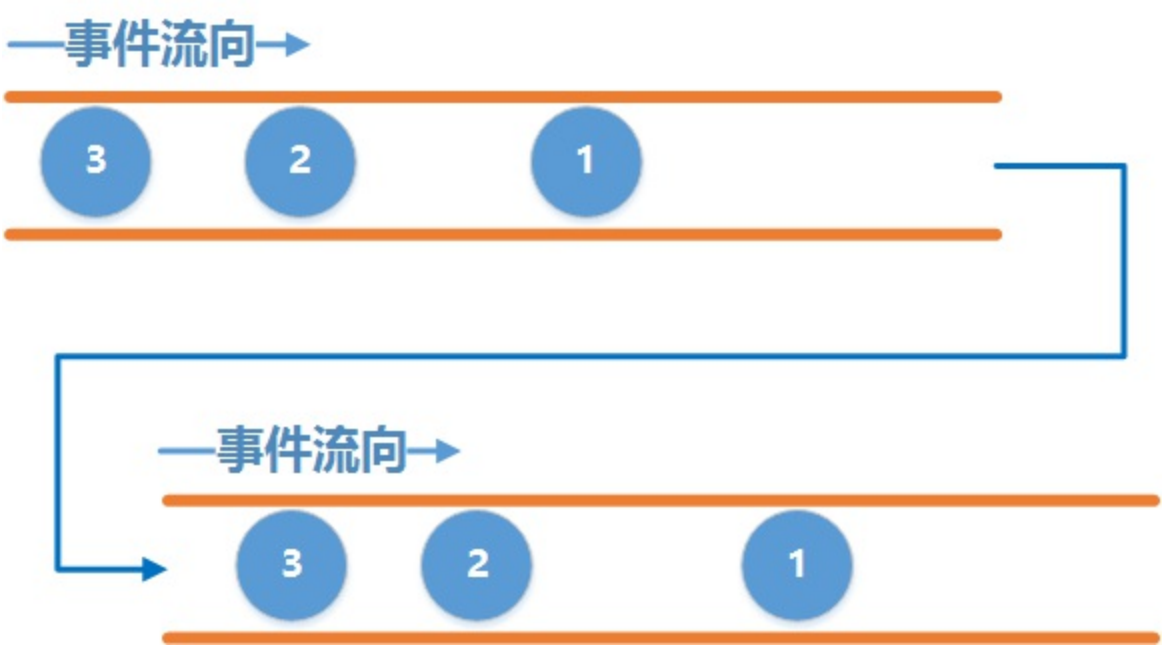
正题

在开始学习之前，先来介绍点原理性的东西。

网上也有很多介绍RxJava原理的文章，通常这些文章都从观察者模式开始，先讲观察者，被观察者，订阅关系巴拉巴拉一大堆，说实话，当我第一次看到这些文章的时候已经被这些名词给绕晕了，用了很长的时间才理清楚它们之间的关系。可能是我太蠢了，境界不够，领会不到那么多高大上的名词.

今天我用两根水管代替观察者和被观察者, 试图用通俗易懂的话把它们的关系解释清楚, 在这里我将从事件流这个角度来说明RxJava的基本工作原理。

先假设有两根水管：



上面一根水管为事件产生的水管，叫它 `上游` 吧，下面一根水管为事件接收的水管叫它 `下游` 吧。

两根水管通过一定的方式连接起来，使得上游每产生一个事件，下游就能收到该事件。注意这里和官网的事件图是反过来的，这里的事件发送的顺序是先1，后2，后3这样的顺序，事件接收的顺序也是先1,后2,后3的顺序，我觉得这样更符合我们普通人的思维，简单明了

这里的 `上游` 和 `下游` 就分别对应着RxJava中的 `Observable` 和 `Observer` ，它们之间的连接就对应着 `subscribe()` ，因此这个关系用RxJava来表示就是

```
//创建一个上游 Observable:
Observable<Integer> observable = Observable.create(new ObservableOnSubscribe<Integer>()
 {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);
        emitter.onComplete();
    }
});
//创建一个下游 Observer
Observer<Integer> observer = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "subscribe");
    }

    @Override
    public void onNext(Integer value) {
        Log.d(TAG, "" + value);
    }
```

```
    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "error");
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "complete");
    }
};
//建立连接
observable.subscribe(observer);
```

这个运行的结果就是:

```
12-02 03:37:17.818 4166-4166/zlc.season.rxjava2demo D/TAG: subscribe
12-02 03:37:17.819 4166-4166/zlc.season.rxjava2demo D/TAG: 1
12-02 03:37:17.819 4166-4166/zlc.season.rxjava2demo D/TAG: 2
12-02 03:37:17.819 4166-4166/zlc.season.rxjava2demo D/TAG: 3
12-02 03:37:17.819 4166-4166/zlc.season.rxjava2demo D/TAG: complete
```

> 注意: 只有当上游和下游建立连接之后, 上游才会开始发送事件. 也就是调用了 `subscribe()` 方法之后才开始发送事件.

把这段代码连起来写就成了RxJava引以为傲的链式操作：

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);
        emitter.onComplete();
    }
}).subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "subscribe");
    }

    @Override
    public void onNext(Integer value) {
        Log.d(TAG, "" + value);
    }

    @Override
```

```
    public void onError(Throwable e) {
        Log.d(TAG, "error");
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "complete");
    }
});
```

接下来解释一下其中两个陌生的玩意：`ObservableEmitter` 和 `Disposable`

ObservableEmitter： Emitter是发射器的意思，那就很好猜了，这个就是用来发出事件的，它可以发出三种类型的事件，通过调用emitter的 `onNext(T value)` 、 `onComplete()` 和 `onError(Throwable error)` 就可以分别发出next事件、complete事件和error事件

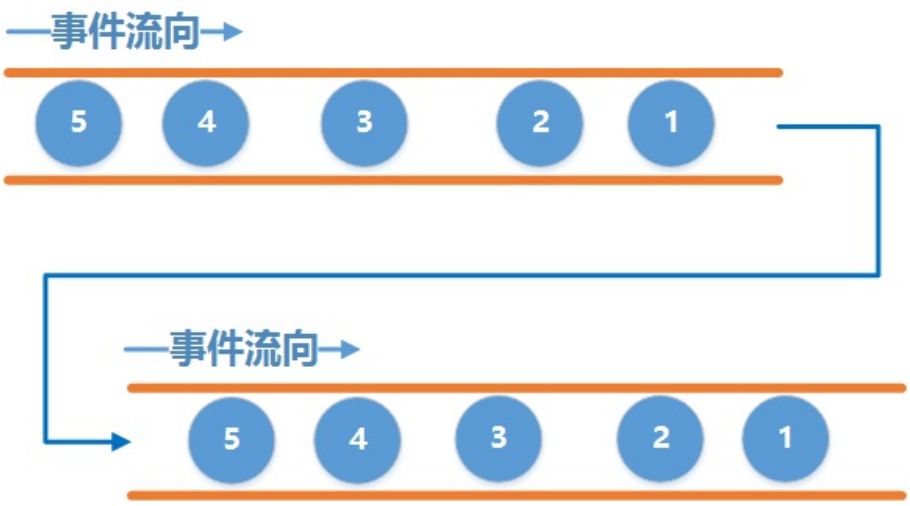
但是，请注意，并不意味着你可以随意乱七八糟发射事件，需要满足一定的规则：

- 上游可以发送无限个onNext, 下游也可以接收无限个onNext
- 当上游发送了一个onComplete后, 上游onComplete之后的事件将会 `继续` 发送, 而下游收到onComplete事件之后将 `不再继续` 接收事件
- 当上游发送了一个onError后, 上游onError之后的事件将 `继续` 发送, 而下游收到onError事件之后将 `不再继续` 接收事件.
- 上游可以不发送onComplete或onError
- 最为关键的是onComplete和onError必须唯一并且互斥, 即不能发多个onComplete, 也不能发多个onError, 也不能先发一个onComplete, 然后再发一个onError, 反之亦然

> 注: 关于onComplete和onError唯一并且互斥这一点, 是需要自行在代码中进行控制, 如果你的代码逻辑中违背了这个规则, **并不一定会导致程序崩溃.** 比如发送多个onComplete是可以正常运行的, 依然是收到第一个onComplete就不再接收了, 但若是发送多个onError, 则收到第二个onError事件会导致程序会崩溃.

以上几个规则用示意图表示如下:

	示意图

只发送onNext事件	
发送onComplete事件	
发送onError事件	

介绍了ObservableEmitter, 接下来介绍Disposable, 这个单词的字面意思是一次性用品,用完即可丢弃的. 那么在RxJava中怎么去理解它呢, 对应于上面的水管的例子, 我们可以把它理解成两根管道之间的一个机关, 当调用它的 `dispose()` 方法时, 它就会将两根管道切断, 从而导致下游收不到事件

> 注意: 调用dispose()并不会导致上游不再继续发送事件, 上游会继续发送剩余的事件.

来看个例子, 我们让上游依次发送 `1,2,3,complete,4` ,在下游收到第二个事件之后, 切断水管, 看看运行结果:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        Log.d(TAG, "emit 1");
        emitter.onNext(1);
        Log.d(TAG, "emit 2");
        emitter.onNext(2);
        Log.d(TAG, "emit 3");
        emitter.onNext(3);
        Log.d(TAG, "emit complete");
        emitter.onComplete();
        Log.d(TAG, "emit 4");
        emitter.onNext(4);
    }
}).subscribe(new Observer<Integer>() {
    private Disposable mDisposable;
    private int i;

    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "subscribe");
        mDisposable = d;
    }

    @Override
    public void onNext(Integer value) {
        Log.d(TAG, "onNext: " + value);
        i++;
        if (i == 2) {
            Log.d(TAG, "dispose");
            mDisposable.dispose();
            Log.d(TAG, "isDisposed : " + mDisposable.isDisposed());
        }
    }

    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "error");
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "complete");
    }
});
```

运行结果为:

```
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: subscribe
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: emit 1
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: onNext: 1
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: emit 2
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: onNext: 2
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: dispose
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: isDisposed : true
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: emit 3
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: emit complete
12-02 06:54:07.728 7404-7404/zlc.season.rxjava2demo D/TAG: emit 4
```

从运行结果我们看到, 在收到onNext 2这个事件后, 切断了水管, 但是上游仍然发送了3, complete, 4这几个事件, 而且上游并没有因为发送了onComplete而停止. 同时可以看到下游的 `onSubscribe()` 方法是最先调用的

Disposable的用处不止这些, 后面讲解到了线程的调度之后, 我们会发现它的重要性. 随着后续深入的讲解, 我们会在更多的地方发现它的身影

另外, `subscribe()` 有多个重载的方法:

```
public final Disposable subscribe() {}
public final Disposable subscribe(Consumer<? super T> onNext) {}
public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwabl
e> onError) {}
public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwabl
e> onError, Action onComplete) {}
public final Disposable subscribe(Consumer<? super T> onNext, Consumer<? super Throwabl
e> onError, Action onComplete, Consumer<? super Disposable> onSubscribe) {}
public final void subscribe(Observer<? super T> observer) {}
```

最后一个带有 `Observer` 参数的我们已经使用过了,这里对其他几个方法进行说明.

- 不带任何参数的 `subscribe()` 表示下游不关心任何事件,你上游尽管发你的数据去吧, 老子可不管你发什么.
- 带有一个 `Consumer` 参数的方法表示下游只关心onNext事件, 其他的事件我假装没看见, 因此我们如果只需要onNext事件可以这么写:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        Log.d(TAG, "emit 1");
        emitter.onNext(1);
        Log.d(TAG, "emit 2");
        emitter.onNext(2);
```

```
            Log.d(TAG, "emit 3");
            emitter.onNext(3);
            Log.d(TAG, "emit complete");
            emitter.onComplete();
            Log.d(TAG, "emit 4");
            emitter.onNext(4);
        }
    }).subscribe(new Consumer<Integer>() {
        @Override
        public void accept(Integer integer) throws Exception {
            Log.d(TAG, "onNext: " + integer);
        }
    });
```

- 其他几个方法同理, 这里就不一一解释了

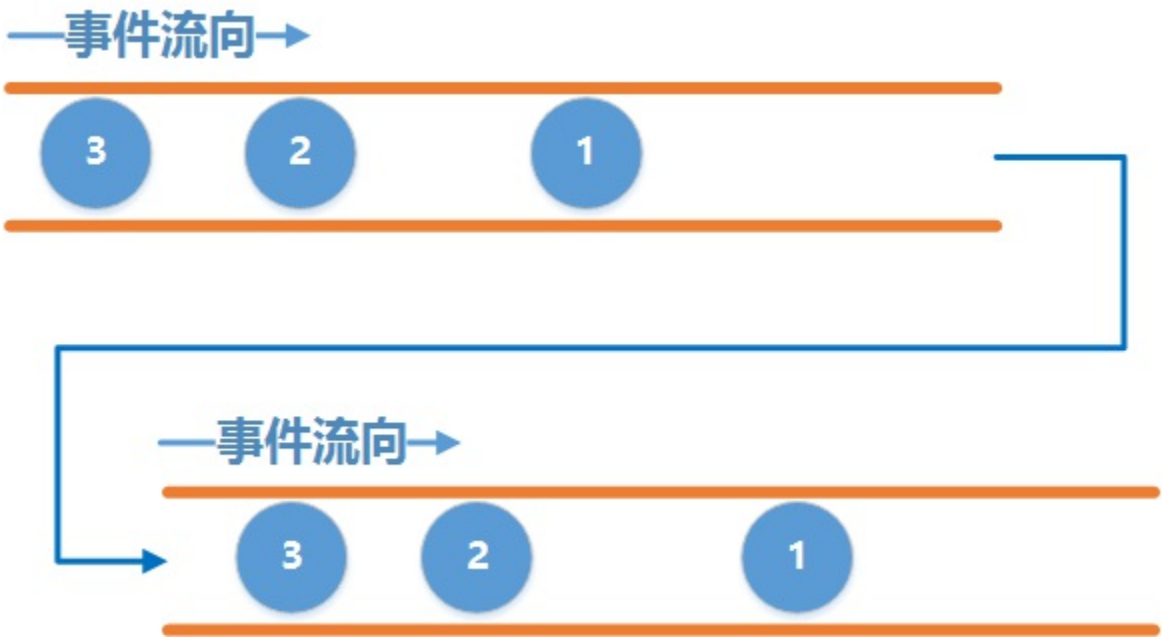
好了本次的教程到此结束, 下一节中我们将会学习RxJava强大的线程调度

前言

上一节教程讲解了最基本的RxJava2的使用, 在本节中, 我们将学习RxJava强大的线程控制.

正题

还是以之前的例子, 两根水管:



正常情况下, 上游和下游是工作在同一个线程中的, 也就是说上游在哪个线程发事件, 下游就在哪个线程接收事件.

怎么去理解呢, 以Android为例, 一个Activity的所有动作默认都是在主线程中运行的, 比如我们在onCreate中打出当前线程的名字:

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    Log.d(TAG, Thread.currentThread().getName());
}
```

结果便是:

```
D/TAG: main
```

回到RxJava中, 当我们在主线程中去创建一个上游Observable来发送事件, 则这个上游默认就在主线程发送事件.

当我们在主线程去创建一个下游Observer来接收事件, 则这个下游默认就在主线程中接收事件, 来看段代码:

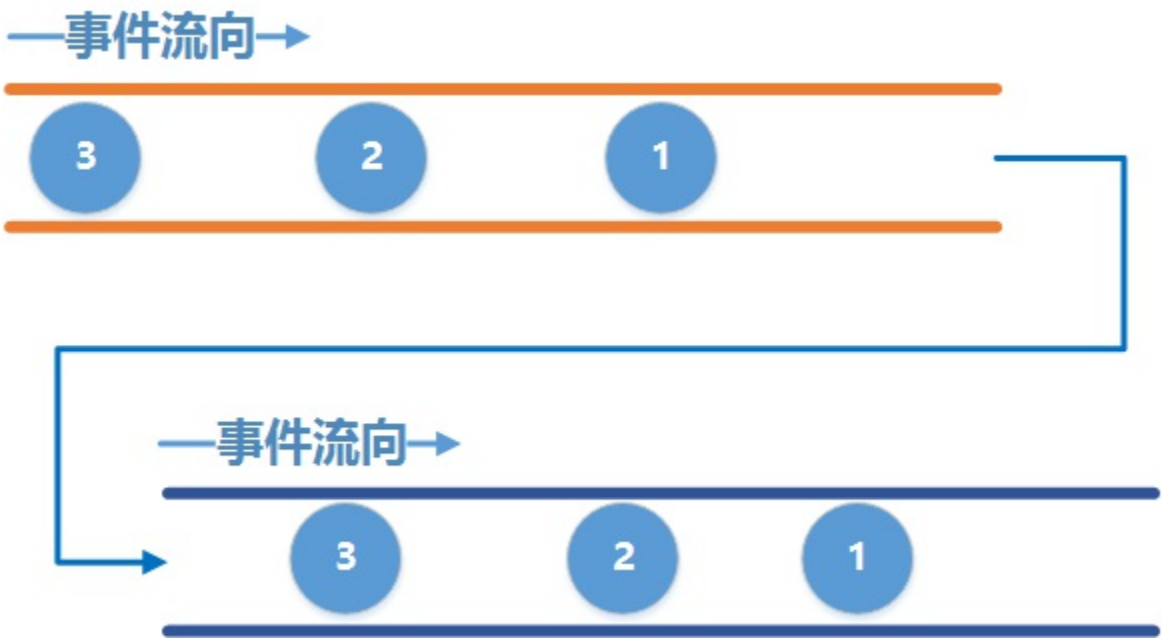
```
@Override

protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);


    Observable<Integer> observable = Observable.create(new ObservableOnSubscribe<Intege
r>() {
        @Override

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

            Log.d(TAG, "Observable thread is : " + Thread.currentThread().getName());

            Log.d(TAG, "emit 1");

            emitter.onNext(1);

        }

    });


    Consumer<Integer> consumer = new Consumer<Integer>() {

        @Override

        public void accept(Integer integer) throws Exception {

            Log.d(TAG, "Observer thread is :" + Thread.currentThread().getName());

            Log.d(TAG, "onNext: " + integer);

        }

    };


    observable.subscribe(consumer);

}
```

在主线程中分别创建上游和下游, 然后将他们连接在一起, 同时分别打印出它们所在的线程, 运行结果为:

```
D/TAG: Observable thread is : main
D/TAG: emit 1
D/TAG: Observer thread is :main
D/TAG: onNext: 1
```

这就验证了刚才所说, 上下游默认是在同一个线程工作.

这样肯定是满足不了我们的需求的, 我们更多想要的是这么一种情况, 在子线程中做耗时的操作, 然后回到主线程中来操作UI, 用图片来描述就是下面这个图片:



在这个图中, 我们用黄色水管表示子线程, 深蓝色水管表示主线程.

要达到这个目的, 我们需要先改变上游发送事件的线程, 让它去子线程中发送事件, 然后再改变下游的线程, 让它去主线程接收事件. 通过RxJava内置的线程调度器可以很轻松的做到这一点. 接下来看一段代码:

```
@Override

protected void onCreate(Bundle savedInstanceState) {

    super.onCreate(savedInstanceState);

    setContentView(R.layout.activity_main);


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

```
        @Override
        public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
            Log.d(TAG, "Observable thread is : " + Thread.currentThread().getName());
            Log.d(TAG, "emit 1");
            emitter.onNext(1);
        }
    });

    Consumer<Integer> consumer = new Consumer<Integer>() {
        @Override
        public void accept(Integer integer) throws Exception {
            Log.d(TAG, "Observer thread is :" + Thread.currentThread().getName());
            Log.d(TAG, "onNext: " + integer);
        }
    };

    observable.subscribeOn(Schedulers.newThread())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(consumer);
}
```

还是刚才的例子, 只不过我们太添加了一点东西, 先来看看运行结果:

```
D/TAG: Observable thread is : RxNewThreadScheduler-2
D/TAG: emit 1
D/TAG: Observer thread is :main
D/TAG: onNext: 1
```

可以看到, 上游发送事件的线程的确改变了, 是在一个叫 `RxNewThreadScheduler-2` 的线程中发送的事件, 而下游仍然在主线程中接收事件, 这说明我们的目的达成了, 接下来看看是如何做到的.

和上一段代码相比,这段代码只不过是增加了两行代码:

```
.subscribeOn(Schedulers.newThread())
.observeOn(AndroidSchedulers.mainThread())
```

作为一个初学者的入门教程, 并不会贴出一大堆源码来分析, 因此只需要让大家记住几个要点, 已达到如何正确的去使用这个目的才是我们的目标.

简单的来说, `subscribeOn()` 指定的是上游发送事件的线程, `observeOn()` 指定的是下游接收事件的线程.

多次指定上游的线程只有第一次指定的有效, 也就是说多次调用 `subscribeOn()` 只有第一次的有效, 其余的会被忽略.

多次指定下游的线程是可以的, 也就是说每调用一次 `observeOn()` , 下游的线程就会切换一次.

举个例子:

```
observable.subscribeOn(Schedulers.newThread())
        .subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .observeOn(Schedulers.io())
        .subscribe(consumer);
```

这段代码中指定了两次上游发送事件的线程, 分别是newThread和IO线程, 下游也指定了两次线程,分别是main和IO线程. 运行结果为:

```
D/TAG: Observable thread is : RxNewThreadScheduler-3
D/TAG: emit 1
D/TAG: Observer thread is :RxCachedThreadScheduler-1
D/TAG: onNext: 1
```

可以看到, 上游虽然指定了两次线程, 但只有第一次指定的有效, 依然是在 `RxNewThreadScheduler` 线程中, 而下游则跑到了 `RxCachedThreadScheduler` 中, 这个CacheThread其实就是IO线程池中的一个.

为了更清晰的看到下游的线程切换过程, 我们加点log:

```
observable.subscribeOn(Schedulers.newThread())
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .doOnNext(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    Log.d(TAG, "After observeOn(mainThread), current thread is: " + Thre
ad.currentThread().getName());
                }
```

```
            })
            .observeOn(Schedulers.io())
            .doOnNext(new Consumer<Integer>() {
                @Override
                public void accept(Integer integer) throws Exception {
                    Log.d(TAG, "After observeOn(io), current thread is : " + Thread.curr
entThread().getName());
                }
            })
            .subscribe(consumer);
```

我们在下游线程切换之后, 把当前的线程打印出来, 运行结果:

```
D/TAG: Observable thread is : RxNewThreadScheduler-1

D/TAG: emit 1

D/TAG: After observeOn(mainThread), current thread is: main

D/TAG: After observeOn(io), current thread is : RxCachedThreadScheduler-2

D/TAG: Observer thread is :RxCachedThreadScheduler-2

D/TAG: onNext: 1
```

可以看到, 每调用一次 `observeOn()` 线程便会切换一次, 因此如果我们有类似的需求时, 便可知道如何处理了.

在RxJava中, 已经内置了很多线程选项供我们选择, 例如有

- Schedulers.io() 代表io操作的线程, 通常用于网络,读写文件等io密集型的操作
- Schedulers.computation() 代表CPU计算密集型的操作, 例如需要大量计算的操作
- Schedulers.newThread() 代表一个常规的新线程
- AndroidSchedulers.mainThread() 代表Android的主线程

这些内置的Scheduler已经足够满足我们开发的需求, 因此我们应该使用内置的这些选项, 在RxJava内部使用的是线程池来维护这些线程, 所有效率也比较高.

实践

对于我们Android开发人员来说, 经常会将一些耗时的操作放在后台, 比如网络请求或者读写文件,操作数据库等等,等到操作完成之后回到主线程去更新UI, 有了上面的这些基础, 那么现在我们就可以轻松的去做到这样一些操作.

下面来举几个常用的场景.

网络请求

Android中有名的网络请求库就那么几个, Retrofit能够从中脱颖而出很大原因就是因为它支持RxJava的方式来调用, 下面简单讲解一下它的基本用法.

要使用Retrofit,先添加Gradle配置:

```
//retrofit
compile 'com.squareup.retrofit2:retrofit:2.1.0'
//Gson converter
compile 'com.squareup.retrofit2:converter-gson:2.1.0'
//RxJava2 Adapter
compile 'com.jakewharton.retrofit:retrofit2-rxjava2-adapter:1.0.0'
//okhttp
compile 'com.squareup.okhttp3:okhttp:3.4.1'
compile 'com.squareup.okhttp3:logging-interceptor:3.4.1'
```

随后定义Api接口:

```
public interface Api {
    @GET
    Observable<LoginResponse> login(@Body LoginRequest request);

    @GET
    Observable<RegisterResponse> register(@Body RegisterRequest request);
}
```

接着创建一个Retrofit客户端:

```
private static Retrofit create() {
            OkHttpClient.Builder builder = new OkHttpClient().newBuilder();
            builder.readTimeout(10, TimeUnit.SECONDS);
            builder.connectTimeout(9, TimeUnit.SECONDS);

            if (BuildConfig.DEBUG) {
                HttpLoggingInterceptor interceptor = new HttpLoggingInterceptor();
                interceptor.setLevel(HttpLoggingInterceptor.Level.BODY);
                builder.addInterceptor(interceptor);
            }

            return new Retrofit.Builder().baseUrl(ENDPOINT)
                    .client(builder.build())
                    .addConverterFactory(GsonConverterFactory.create())
                    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
                    .build();
}
```

发起请求就很简单了:

```
Api api = retrofit.create(Api.class);
api.login(request)
      .subscribeOn(Schedulers.io())               //在IO线程进行网络请求
     .observeOn(AndroidSchedulers.mainThread())  //回到主线程去处理请求结果
    .subscribe(new Observer<LoginResponse>() {
            @Override
        public void onSubscribe(Disposable d) {}

        @Override
        public void onNext(LoginResponse value) {}

        @Override
        public void onError(Throwable e) {
            Toast.makeText(mContext, "登录失败", Toast.LENGTH_SHORT).show();
        }

        @Override
        public void onComplete() {
            Toast.makeText(mContext, "登录成功", Toast.LENGTH_SHORT).show();
        }
    });
```

看似很完美, 但我们忽略了一点, 如果在请求的过程中Activity已经退出了, 这个时候如果回到主线程去更新UI, 那么APP肯定就崩溃了, 怎么办呢, 上一节我们说到了 `Disposable` , 说它是个开关, 调用它的 `dispose()` 方法时就会切断水管, 使得下游收不到事件, 既然收不到事件, 那么也就不会再去更新UI了. 因此我们可以在Activity中将这个 `Disposable` 保存起来, 当Activity退出时, 切断它即可.

那如果有多个 `Disposable` 该怎么办呢, RxJava中已经内置了一个容器 `CompositeDisposable` , 每当我们得到一个 `Disposable` 时就调用 `CompositeDisposable.add()` 将它添加到容器中, 在退出的时候, 调用 `CompositeDisposable.clear()` 即可切断所有的水管.

读写数据库

上面说了网络请求的例子, 接下来再看看读写数据库, 读写数据库也算一个耗时的操作, 因此我们也最好放在IO线程里去进行, 这个例子就比较简单, 直接上代码:

```
public Observable<List<Record>> readAllRecords() {
        return Observable.create(new ObservableOnSubscribe<List<Record>>() {
            @Override
            public void subscribe(ObservableEmitter<List<Record>> emitter) throws Excep
tion {
                Cursor cursor = null;
                try {
                    cursor = getReadableDatabase().rawQuery("select * from " + TABLE_NA
```

```
ME, new String[]{});
                    List<Record> result = new ArrayList<>();
                    while (cursor.moveToNext()) {
                        result.add(Db.Record.read(cursor));
                    }
                    emitter.onNext(result);
                    emitter.onComplete();
                } finally {
                    if (cursor != null) {
                        cursor.close();
                    }
                }
            }
        }).subscribeOn(Schedulers.io()).observeOn(AndroidSchedulers.mainThread());
    }
```

好了本次的教程就到这里吧, 后面的教程将会教大家如何使用RxJava中强大的操作符. 通过使用这些操作符可以很轻松的做到各种吊炸天的效果. 敬请期待

前言

上一节讲解了线程调度, 并且举了两个实际中的例子, 其中有一个登录的例子, 不知大家有没有想过这么一个问题, 如果是一个新用户, 必须先注册, 等注册成功之后再自动登录该怎么做呢.

很明显, 这是一个嵌套的网络请求, 首先需要去请求注册, 待注册成功回调了再去请求登录的接口.

我们当然可以想当然的写成这样:

```
private void login() {
    api.login(new LoginRequest())
            .subscribeOn(Schedulers.io())               //在IO线程进行网络请求
            .observeOn(AndroidSchedulers.mainThread())  //回到主线程去处理请求结果
            .subscribe(new Consumer<LoginResponse>() {
                @Override
                public void accept(LoginResponse loginResponse) throws Exception {
                    Toast.makeText(MainActivity.this, "登录成功", Toast.LENGTH_SHORT).s
how();
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    Toast.makeText(MainActivity.this, "登录失败", Toast.LENGTH_SHORT).s
how();
                }
            });
}

private void register() {
    api.register(new RegisterRequest())
            .subscribeOn(Schedulers.io())               //在IO线程进行网络请求
            .observeOn(AndroidSchedulers.mainThread())  //回到主线程去处理请求结果
            .subscribe(new Consumer<RegisterResponse>() {
                @Override
                public void accept(RegisterResponse registerResponse) throws Exception
{
                    Toast.makeText(MainActivity.this, "注册成功", Toast.LENGTH_SHORT).s
how();
                    login();   //注册成功, 调用登录的方法
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(Throwable throwable) throws Exception {
                    Toast.makeText(MainActivity.this, "注册失败", Toast.LENGTH_SHORT).s
how();
                }
            });
```

```
}
```

(其实能写成这样的代码的人也很不错了, 至少没写到一起...)

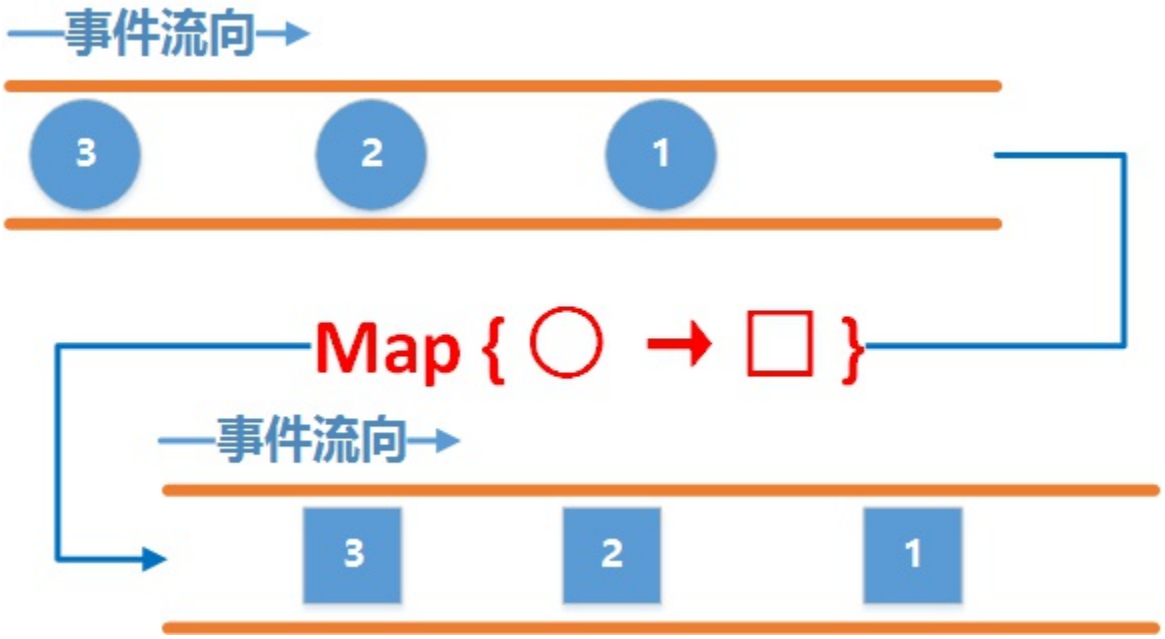
这样的代码能够工作, 但不够优雅, 通过本节的学习, 可以让我们用一种更优雅的方式来解决这个问题.

正题

先来看看最简单的变换操作符map吧

Map

map是RxJava中最简单的一个变换操作符了, 它的作用就是对上游发送的每一个事件应用一个函数, 使得每一个事件都按照指定的函数去变化. 用事件图表示如下:



图中map中的函数作用是将圆形事件转换为矩形事件, 从而导致下游接收到的事件就变为了矩形.用代码来表示这个例子就是:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
        @Override
        public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
            emitter.onNext(1);
            emitter.onNext(2);
            emitter.onNext(3);
        }
    }).map(new Function<Integer, String>() {
        @Override
        public String apply(Integer integer) throws Exception {
```

```
            return "This is result " + integer;
        }
    }).subscribe(new Consumer<String>() {
        @Override
        public void accept(String s) throws Exception {
            Log.d(TAG, s);
        }
    });
```

在上游我们发送的是数字类型, 而在下游我们接收的是String类型, 中间起转换作用的就是map操作符, 运行结果为:

```
D/TAG: This is result 1
D/TAG: This is result 2
D/TAG: This is result 3
```

通过Map, 可以将上游发来的事件转换为任意的类型, 可以是一个Object, 也可以是一个集合, 如此强大的操作符你难道不想试试?

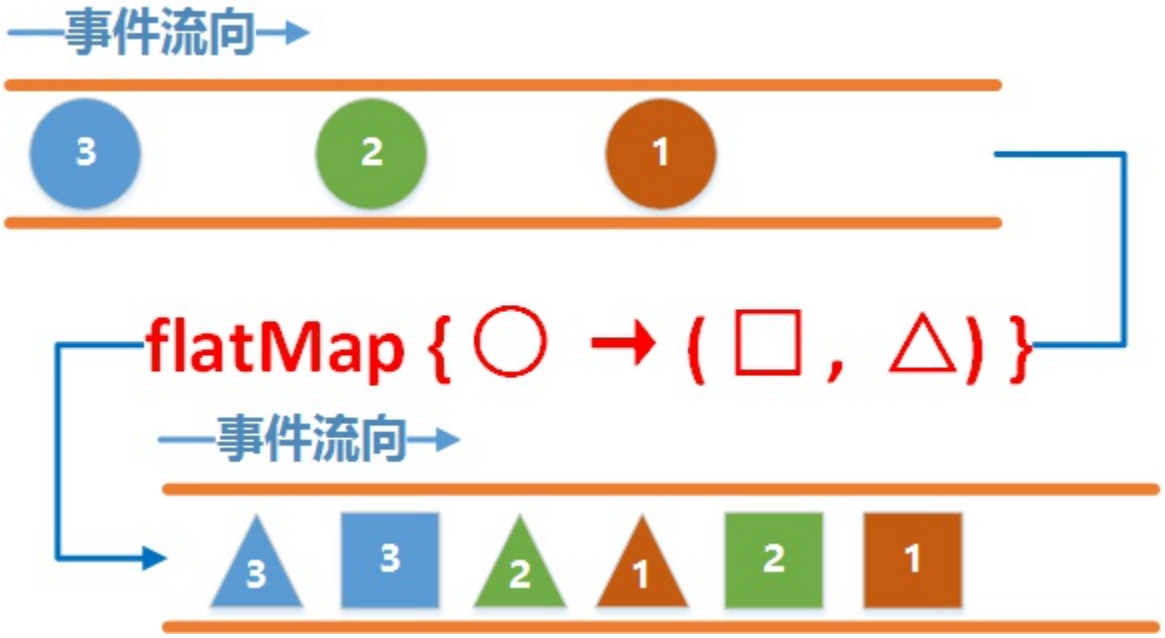
接下来我们来看另外一个广为人知的操作符flatMap.

FlatMap

flatMap是一个非常强大的操作符, 先用一个比较难懂的概念说明一下:

`FlatMap` 将一个发送事件的上游Observable变换为多个发送事件的Observables， 然后将它们发射的事件合并后放进一个单独的Observable里.

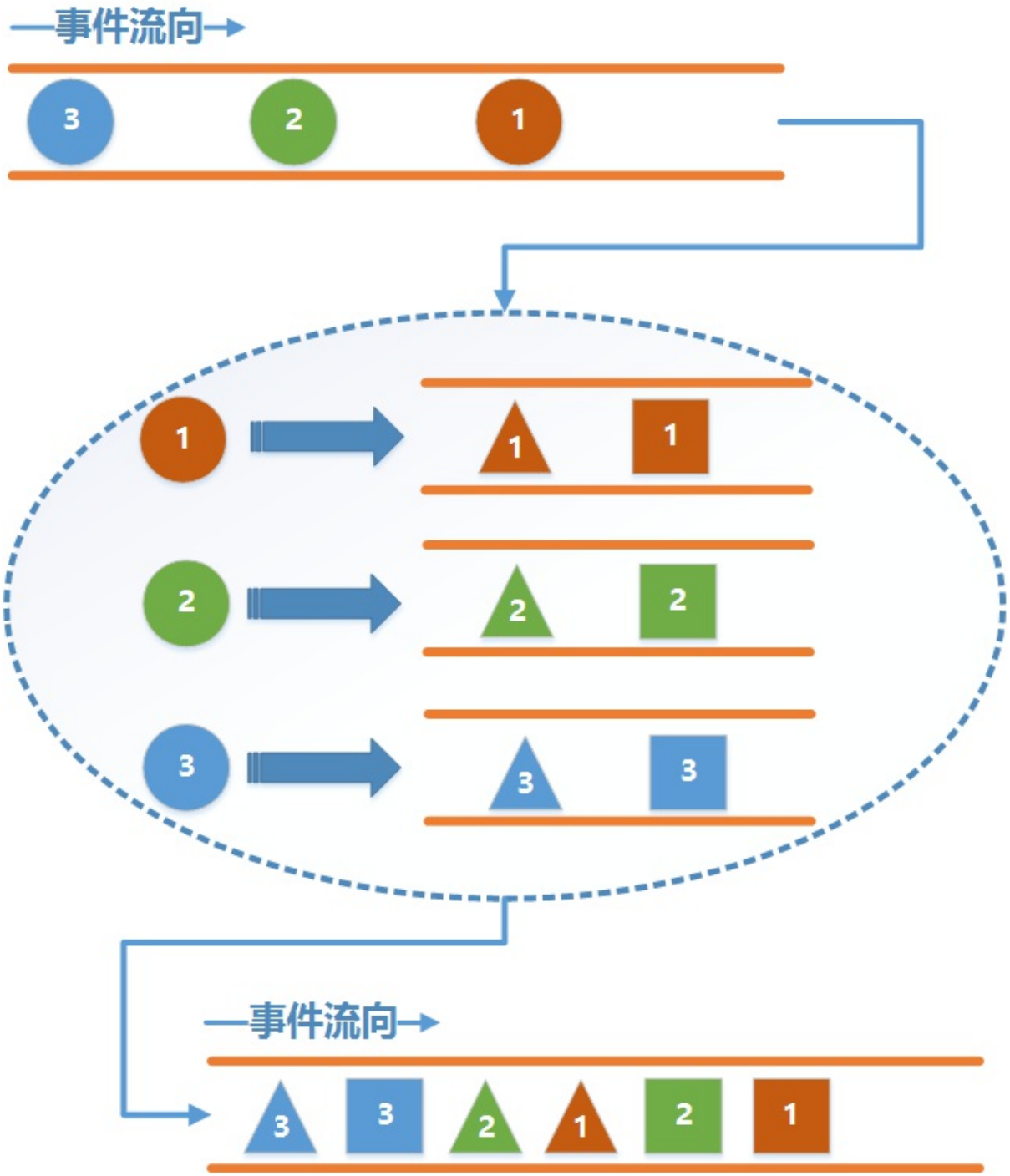
这句话比较难以理解, 我们先通俗易懂的图片来详细的讲解一下, 首先先来看看整体的一个图片:



先看看上游, 上游发送了三个事件, 分别是1,2,3, 注意它们的颜色.

中间flatMap的作用是将圆形的事件转换为一个发送矩形事件和三角形事件的新的上游Observable.

还是不能理解? 别急, 再来看看分解动作:



这样就很好理解了吧 !!!

上游每发送一个事件, flatMap都将创建一个新的水管, 然后发送转换之后的新的事件, 下游接收到的就是这些新的水管发送的数据. **这里需要注意的是, flatMap并不保证事件的顺序,** 也就是图中所看到的, 并不是事件1就在事件2的前面. 如果需要保证顺序则需要使用 `concatMap` .

说了原理, 我们还是来看看实际中的代码如何写吧:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        emitter.onNext(1);
        emitter.onNext(2);
        emitter.onNext(3);
    }
}).flatMap(new Function<Integer, ObservableSource<String>>() {
    @Override
    public ObservableSource<String> apply(Integer integer) throws Exception {
        final List<String> list = new ArrayList<>();
        for (int i = 0; i < 3; i++) {
            list.add("I am value " + integer);
        }
        return Observable.fromIterable(list).delay(10,TimeUnit.MILLISECONDS);
    }
}).subscribe(new Consumer<String>() {
    @Override
    public void accept(String s) throws Exception {
        Log.d(TAG, s);
    }
});
```

如代码所示, 我们在flatMap中将上游发来的每个事件转换为一个新的发送三个String事件的水管, 为了看到flatMap结果是无序的,所以加了10毫秒的延时, 来看看运行结果吧:

```
D/TAG: I am value 1
D/TAG: I am value 1
D/TAG: I am value 1
D/TAG: I am value 3
D/TAG: I am value 3
D/TAG: I am value 3
D/TAG: I am value 2
D/TAG: I am value 2
D/TAG: I am value 2
```

结果也确实验证了我们之前所说.

这里也简单说一下 `concatMap` 吧, 它和flatMap的作用几乎一模一样, 只是它的结果是严格按照上游发送的顺序来发送的, 来看个代码吧:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        emitter.onNext(1);
```

```
            emitter.onNext(2);
            emitter.onNext(3);
        }
    }).concatMap(new Function<Integer, ObservableSource<String>>() {
        @Override
        public ObservableSource<String> apply(Integer integer) throws Exception {
            final List<String> list = new ArrayList<>();
            for (int i = 0; i < 3; i++) {
                list.add("I am value " + integer);
            }
            return Observable.fromIterable(list).delay(10,TimeUnit.MILLISECONDS);
        }
    }).subscribe(new Consumer<String>() {
        @Override
        public void accept(String s) throws Exception {
            Log.d(TAG, s);
        }
    });
```

只是将之前的flatMap改为了concatMap, 其余原封不动, 运行结果如下:

```
D/TAG: I am value 1
D/TAG: I am value 1
D/TAG: I am value 1
D/TAG: I am value 2
D/TAG: I am value 2
D/TAG: I am value 2
D/TAG: I am value 3
D/TAG: I am value 3
D/TAG: I am value 3
```

可以看到, 结果仍然是有序的.

好了关于RxJava的操作符最基本的使用就讲解到这里了, RxJava中内置了许许多多的操作符, 这里通过讲解 `map` 和 `flatMap` 只是起到一个抛砖引玉的作用, 关于其他的操作符只要大家按照本文的思路去理解, 再仔细阅读文档, 应该是没有问题的了, 如果大家有需要也可以将需要讲解的操作符列举出来, 我可以根据大家的需求讲解一下.

实践

学习了FlatMap操作符, 我们就可以回答文章开头提出的那个问题了.

如何优雅的解决嵌套请求, 只需要用flatMap转换一下就行了.

先回顾一下上一节的请求接口:

```
public interface Api {
    @GET
    Observable<LoginResponse> login(@Body LoginRequest request);

    @GET
    Observable<RegisterResponse> register(@Body RegisterRequest request);
}
```

可以看到登录和注册返回的都是一个上游Observable, 而我们的flatMap操作符的作用就是把一个Observable转换为另一个Observable, 因此结果就很显而易见了:

```
api.register(new RegisterRequest())             //发起注册请求
    .subscribeOn(Schedulers.io())               //在IO线程进行网络请求
    .observeOn(AndroidSchedulers.mainThread())  //回到主线程去处理请求注册结果
    .doOnNext(new Consumer<RegisterResponse>() {
        @Override
        public void accept(RegisterResponse registerResponse) throws Exception {
            //先根据注册的响应结果去做一些操作
        }
    })
    .observeOn(Schedulers.io())                 //回到IO线程去发起登录请求
    .flatMap(new Function<RegisterResponse, ObservableSource<LoginResponse>>() {
        @Override
        public ObservableSource<LoginResponse> apply(RegisterResponse registerResponse)
throws Exception {
            return api.login(new LoginRequest());
        }
    })
    .observeOn(AndroidSchedulers.mainThread())  //回到主线程去处理请求登录的结果
    .subscribe(new Consumer<LoginResponse>() {
        @Override
        public void accept(LoginResponse loginResponse) throws Exception {
            Toast.makeText(MainActivity.this, "登录成功", Toast.LENGTH_SHORT).show();
        }
    }, new Consumer<Throwable>() {
        @Override
        public void accept(Throwable throwable) throws Exception {
            Toast.makeText(MainActivity.this, "登录失败", Toast.LENGTH_SHORT).show();
        }
    });
```

从这个例子也可以看到我们切换线程是多么简单.

好了本次的教程就到这里了. 下一节我们将会学到 `Flowable` 以及理解 `Backpressure` 背压的概念. 敬请期待

前言

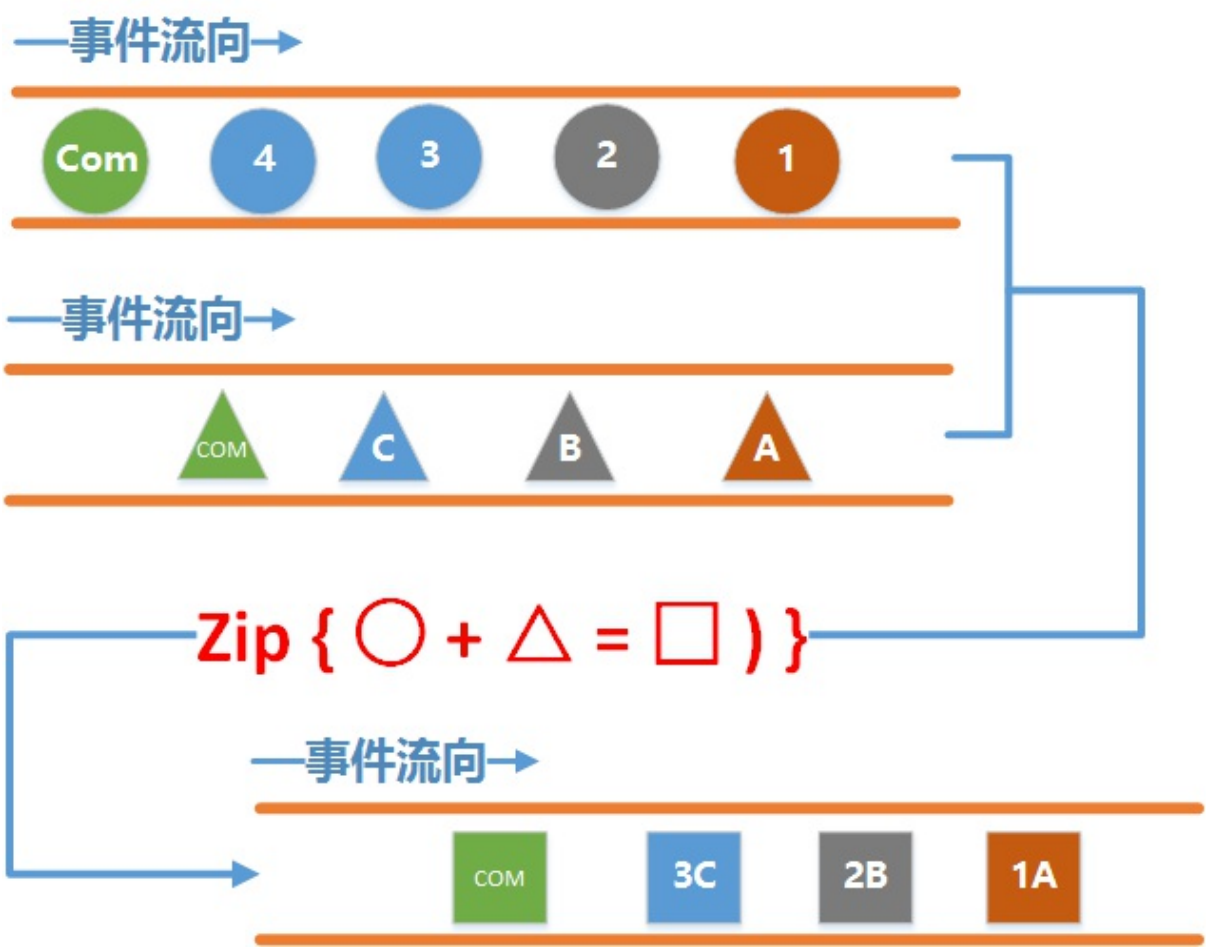
在上一节中, 我们提到了 `Flowable` 和 `Backpressure` 背压, 本来这一节的确是想讲这两个东西的, 可是写到一半感觉还是差点火候, 感觉时机未到, 因此, 这里先来做个准备工作, 先带大家学习 `zip` 这个操作符, 这个操作符也是比较牛逼的东西了, 涉及到的东西也比较多, 主要是一些细节上的东西太多, 通过学习这个操作符, 可以为我们下一节的 `Backpressure` 做个铺垫.

正题

照惯例我们还是先贴上一下比较正式的解释吧.

`Zip` 通过一个函数将多个Observable发送的事件结合到一起, 然后发送这些组合到一起的事件. 它按照严格的顺序应用这个函数。它只发射与发射数据项最少的那个Observable一样多的数据。

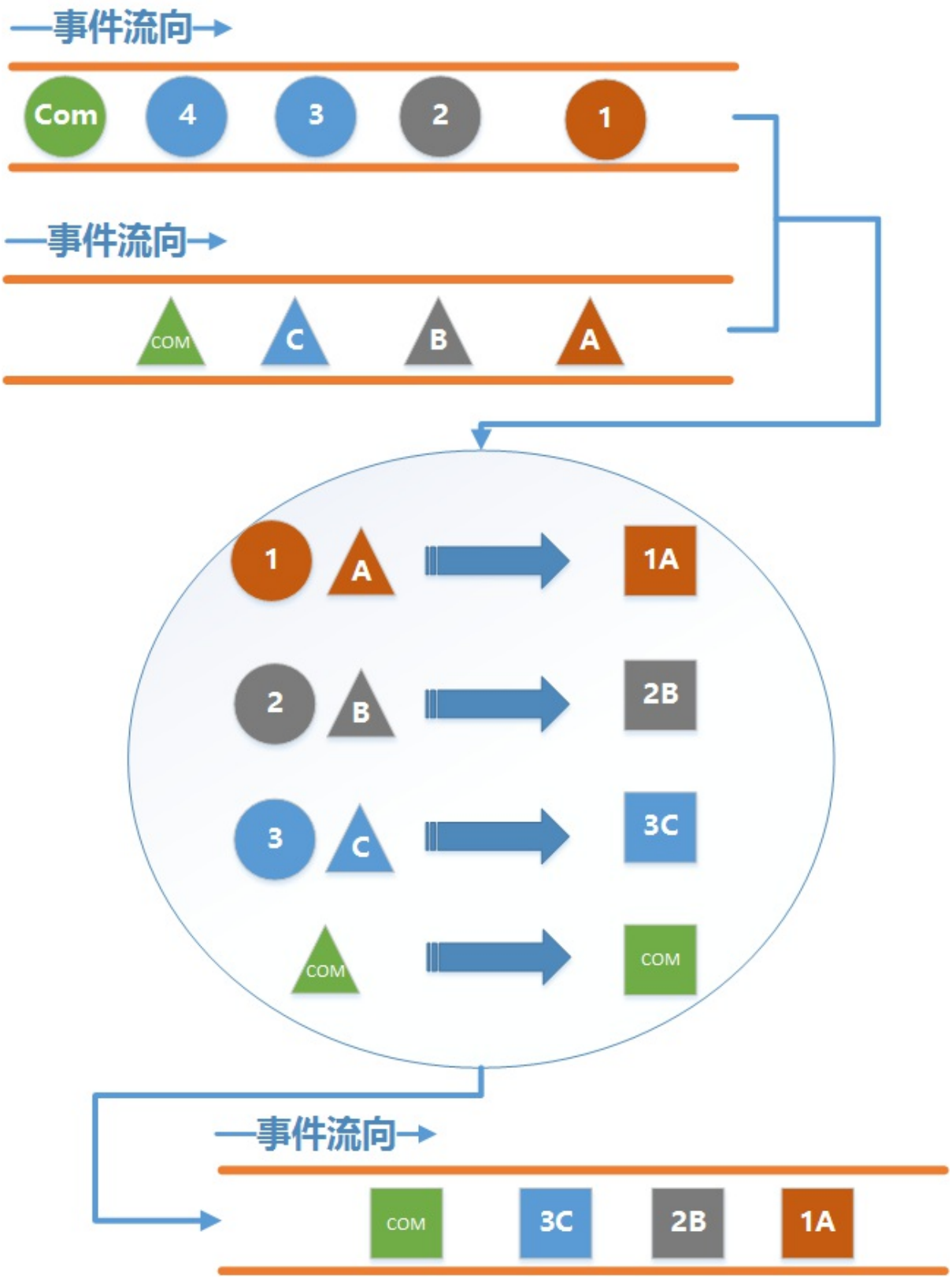
我们再用通俗易懂的图片来解释一下:



从这个图中可以看见, 这次上游和以往不同的是, 我们有两根水管了.

其中一根水管负责发送 `圆形事件` , 另外一根水管负责发送 `三角形事件` , 通过Zip操作符, 使得 `圆形事件` 和 `三角形事件` 合并为了一个 `矩形事件` .

下面我们再来看看分解动作：



通过分解动作我们可以看出:

- 组合的过程是 分别从 两根水管里 各取出一个事件 来进行组合, 并且一个事件 只能被使用一

次，组合的顺序是严格 按照事件发送的顺利 来进行的, 也就是说不会出现 圆形1 事件和 三角形B 事件进行合并, 也不可能出现 圆形2 和 三角形A 进行合并的情况.

- 最终 下游收到的事件数量 是和 上游中发送事件最少的那一根水管的事件数量 相同. 这个也很好理解, 因为是从 每一根水管 里取一个事件来进行合并, 最少的 那个肯定就 最先取完 , 这个时候其他的水管 尽管还有事件 , 但是已经没有足够的事件来组合了, 因此下游就不会收到剩余的事件了.

分析了大概的原理, 我们还是劳逸结合, 先来看看实际中的代码怎么写吧:

```
Observable<Integer> observable1 = Observable.create(new ObservableOnSubscribe<Integer>(
) {
    @Override

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

        Log.d(TAG, "emit 1");

        emitter.onNext(1);

        Log.d(TAG, "emit 2");

        emitter.onNext(2);

        Log.d(TAG, "emit 3");

        emitter.onNext(3);

        Log.d(TAG, "emit 4");

        emitter.onNext(4);

        Log.d(TAG, "emit complete1");

        emitter.onComplete();

    }

});

Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<String>()
{
    @Override

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

        Log.d(TAG, "emit A");

        emitter.onNext("A");
```

```
            Log.d(TAG, "emit B");

            emitter.onNext("B");

            Log.d(TAG, "emit C");

            emitter.onNext("C");

            Log.d(TAG, "emit complete2");

            emitter.onComplete();

        }

});

Observable.zip(observable1, observable2, new BiFunction<Integer, String, String>() {

    @Override

    public String apply(Integer integer, String s) throws Exception {

        return integer + s;

    }

}).subscribe(new Observer<String>() {
    @Override

    public void onSubscribe(Disposable d) {

        Log.d(TAG, "onSubscribe");

    }


    @Override

    public void onNext(String value) {

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

    }


    @Override

    public void onError(Throwable e) {
```

```
        Log.d(TAG, "onError");

    }


    @Override

    public void onComplete() {

        Log.d(TAG, "onComplete");

    }

});
```

我们分别创建了两个上游水管, 一个发送1,2,3,4,Complete, 另一个发送A,B,C,Complete, 接着用Zip把发出的事件组合, 来看看运行结果吧:

```
D/TAG: onSubscribe
D/TAG: emit 1
D/TAG: emit 2
D/TAG: emit 3
D/TAG: emit 4
D/TAG: emit complete1
D/TAG: emit A
D/TAG: onNext: 1A
D/TAG: emit B
D/TAG: onNext: 2B
D/TAG: emit C
D/TAG: onNext: 3C
D/TAG: emit complete2
D/TAG: onComplete
```

结果似乎是对的... 但是总感觉什么地方不对劲...

哪儿不对劲呢, 为什么感觉是水管一发送完了之后, 水管二才开始发送啊? 到底是不是呢, 我们来验证一下:

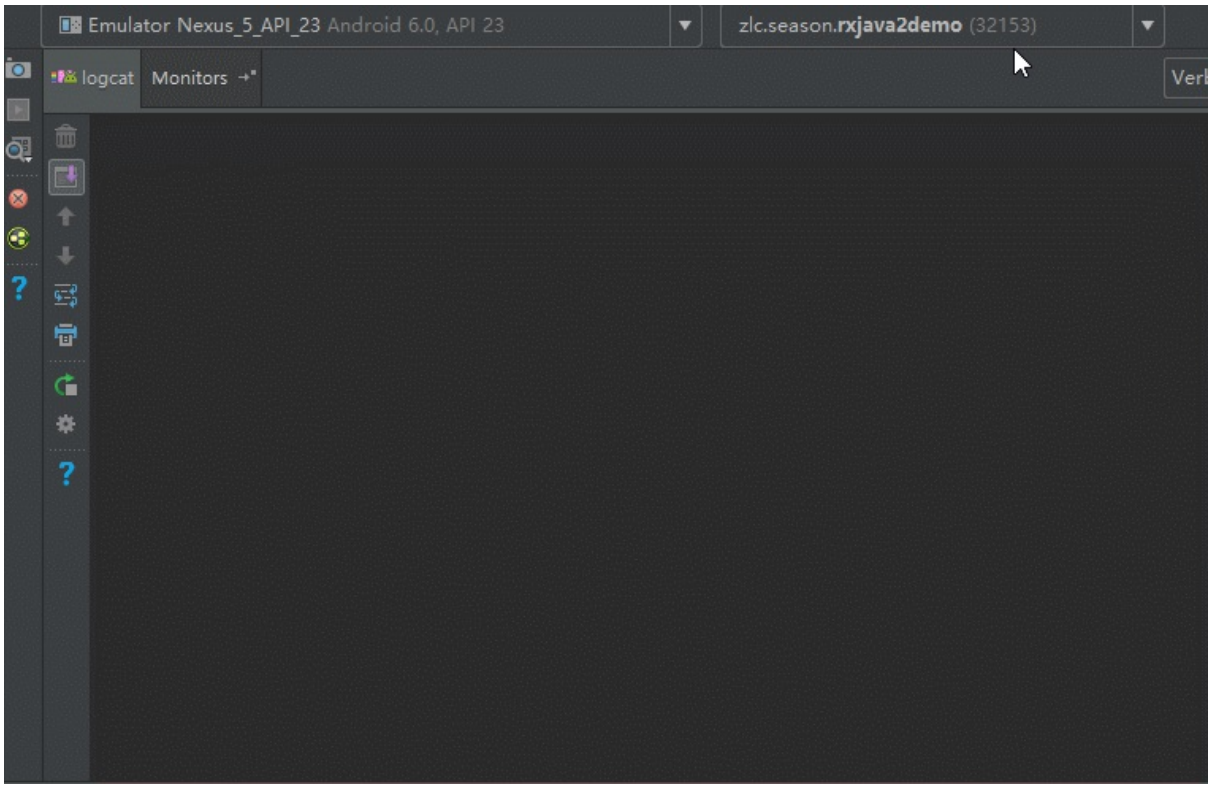
```
Observable<Integer> observable1 = Observable.create(new ObservableOnSubscribe<Integer>(
) {
    @Override

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

        Log.d(TAG, "emit 1");

        emitter.onNext(1);
```

```
        Thread.sleep(1000);


        Log.d(TAG, "emit 2");

        emitter.onNext(2);

        Thread.sleep(1000);


        Log.d(TAG, "emit 3");

        emitter.onNext(3);

        Thread.sleep(1000);


        Log.d(TAG, "emit 4");

        emitter.onNext(4);

        Thread.sleep(1000);


        Log.d(TAG, "emit complete1");

        emitter.onComplete();

    }

});


Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<String>()
{
    @Override

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

        Log.d(TAG, "emit A");

        emitter.onNext("A");

        Thread.sleep(1000);


        Log.d(TAG, "emit B");
```

```
            emitter.onNext("B");
            Thread.sleep(1000);

            Log.d(TAG, "emit C");
            emitter.onNext("C");
            Thread.sleep(1000);

            Log.d(TAG, "emit complete2");
            emitter.onComplete();
        }
    });

    Observable.zip(observable1, observable2, new BiFunction<Integer, String, String>() {
        @Override
        public String apply(Integer integer, String s) throws Exception {
            return integer + s;
        }
    }).subscribe(new Observer<String>() {
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "onSubscribe");
        }

        @Override
        public void onNext(String value) {
            Log.d(TAG, "onNext: " + value);
        }
```

```
    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "onError");
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "onComplete");
    }
});
```

这次我们在每发送一个事件之后加入了一秒钟的延时, 来看看运行结果吧, 注意这是个GIF图:



(贴心的我怕大家看不清楚, 特意调成了老年字体呢)

阿西吧, 好像真的是先发送的水管一再发送的水管二呢, 为什么会有这种情况呢? 因为我们两根水管都是运行在同一个线程里, 同一个线程里执行代码肯定有先后顺序呀.

因此我们来稍微改一下, 不让他们在同一个线程, 不知道怎么切换线程的, 请掉头看前面几节.

```
Observable<Integer> observable1 = Observable.create(new ObservableOnSubscribe<Integer>(
) {
    @Override

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

        Log.d(TAG, "emit 1");

        emitter.onNext(1);

        Thread.sleep(1000);


        Log.d(TAG, "emit 2");

        emitter.onNext(2);

        Thread.sleep(1000);


        Log.d(TAG, "emit 3");

        emitter.onNext(3);

        Thread.sleep(1000);


        Log.d(TAG, "emit 4");

        emitter.onNext(4);

        Thread.sleep(1000);


        Log.d(TAG, "emit complete1");

        emitter.onComplete();

    }

}).subscribeOn(Schedulers.io());


Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<String>()
{
    @Override
```

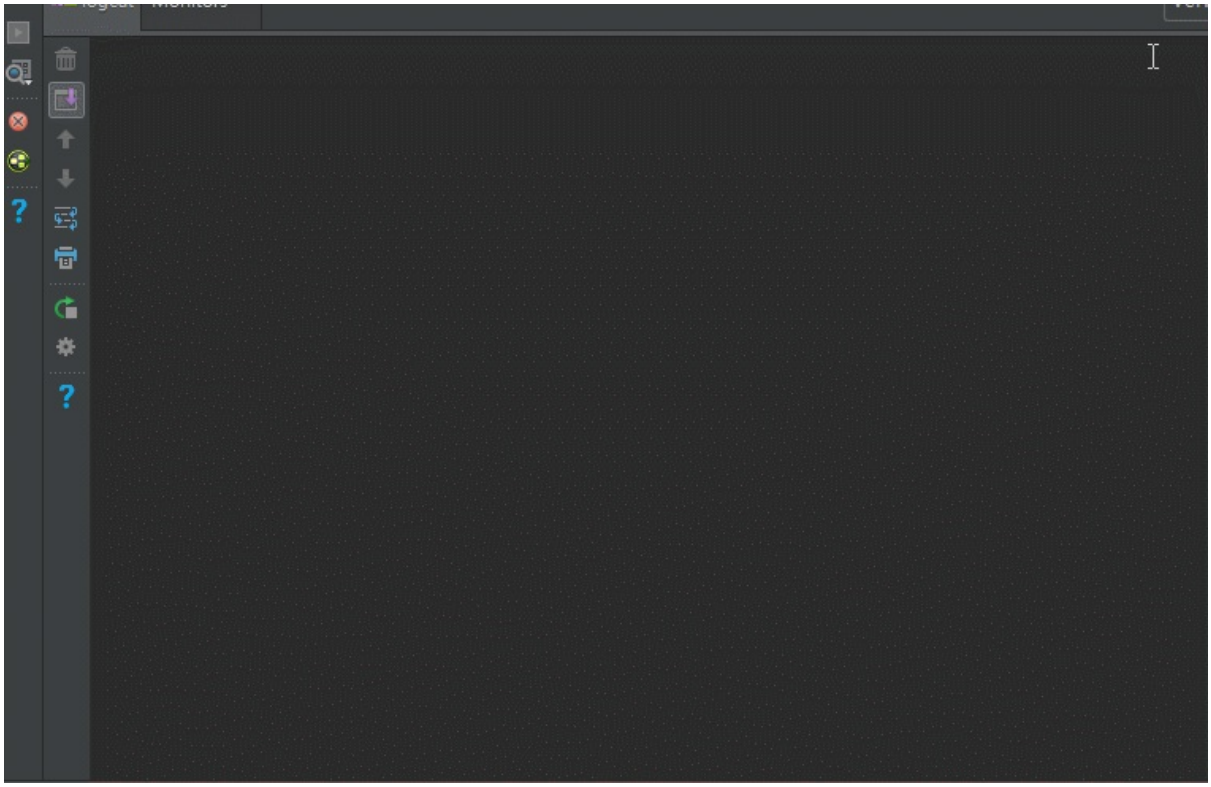
```
    public void subscribe(ObservableEmitter<String> emitter) throws Exception {
        Log.d(TAG, "emit A");
        emitter.onNext("A");
        Thread.sleep(1000);

        Log.d(TAG, "emit B");
        emitter.onNext("B");
        Thread.sleep(1000);

        Log.d(TAG, "emit C");
        emitter.onNext("C");
        Thread.sleep(1000);

        Log.d(TAG, "emit complete2");
        emitter.onComplete();
    }
}).subscribeOn(Schedulers.io());

Observable.zip(observable1, observable2, new BiFunction<Integer, String, String>() {
    @Override
    public String apply(Integer integer, String s) throws Exception {
        return integer + s;
    }
}).subscribe(new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "onSubscribe");
```

```
    }

    @Override
    public void onNext(String value) {
        Log.d(TAG, "onNext: " + value);
    }

    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "onError");
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "onComplete");
    }
});
```

好了, 这次我们让水管都在IO线程里发送事件, 再来看看运行结果:

```
D/TAG: onSubscribe
D/TAG: emit A
D/TAG: emit 1
D/TAG: onNext: 1A
D/TAG: emit B
D/TAG: emit 2
D/TAG: onNext: 2B
D/TAG: emit C
D/TAG: emit 3
D/TAG: onNext: 3C
D/TAG: emit complete2
D/TAG: onComplete
```

GIF图:



诶! 这下就对了嘛, 两根水管同时开始发送, 每发送一个, Zip就组合一个, 再将组合结果发送给下游.

不对呀! 可能细心点的朋友又看出端倪了, 第一根水管明明发送了四个数据+一个Complete, 之前明明还有的, 为啥到这里没了呢?

这是因为我们之前说了, zip发送的事件数量跟上游中发送事件最少的那一根水管的事件数量是有关的, 在这个例子里我们第二根水管只发送了三个事件然后就发送了Complete, 这个时候尽管第一根水管还有 `事件4` 和事件 `Complete` 没有发送, 但是它们发不发送还有什么意义呢? 所以本着节约是美德的思想, 就干脆打断它的狗腿, 不让它发了.

至于前面的例子为什么会发送, 刚才不是已经说了是！在！同！一！个！线！程！里！吗！！！！再问老子打死你！

有好事的程序员可能又要问了， 那我不发送Complete呢？ 答案是显然的, 上游会继续发送事件, 但是下游仍然收不到那些多余的事件. 不信你可以试试.

实践

学习了Zip的基本用法, 那么它在Android有什么用呢, 其实很多场景都可以用到Zip. 举个例子.

比如一个界面需要展示用户的一些信息, 而这些信息分别要从两个服务器接口中获取, 而只有当两个都获取到了之后才能进行展示, 这个时候就可以用Zip了:

首先分别定义这两个请求接口:

```
public interface Api {
    @GET
    Observable<UserBaseInfoResponse> getUserBaseInfo(@Body UserBaseInfoRequest request)
;

    @GET
    Observable<UserExtraInfoResponse> getUserExtraInfo(@Body UserExtraInfoRequest reque
st);

}
```

接着用Zip来打包请求:

```
Observable<UserBaseInfoResponse> observable1 =

        api.getUserBaseInfo(new UserBaseInfoRequest()).subscribeOn(Schedulers.io());


Observable<UserExtraInfoResponse> observable2 =

        api.getUserExtraInfo(new UserExtraInfoRequest()).subscribeOn(Schedulers.io());


Observable.zip(observable1, observable2,

        new BiFunction<UserBaseInfoResponse, UserExtraInfoResponse, UserInfo>() {

            @Override

            public UserInfo apply(UserBaseInfoResponse baseInfo,

                                  UserExtraInfoResponse extraInfo) throws Exception {

                return new UserInfo(baseInfo, extraInfo);

            }

        }).observeOn(AndroidSchedulers.mainThread())

        .subscribe(new Consumer<UserInfo>() {

            @Override

            public void accept(UserInfo userInfo) throws Exception {

                //do something;
```

```
            }

        });
```

好了, 本次的教程就到这里吧. 又到周末鸟, 下周见.

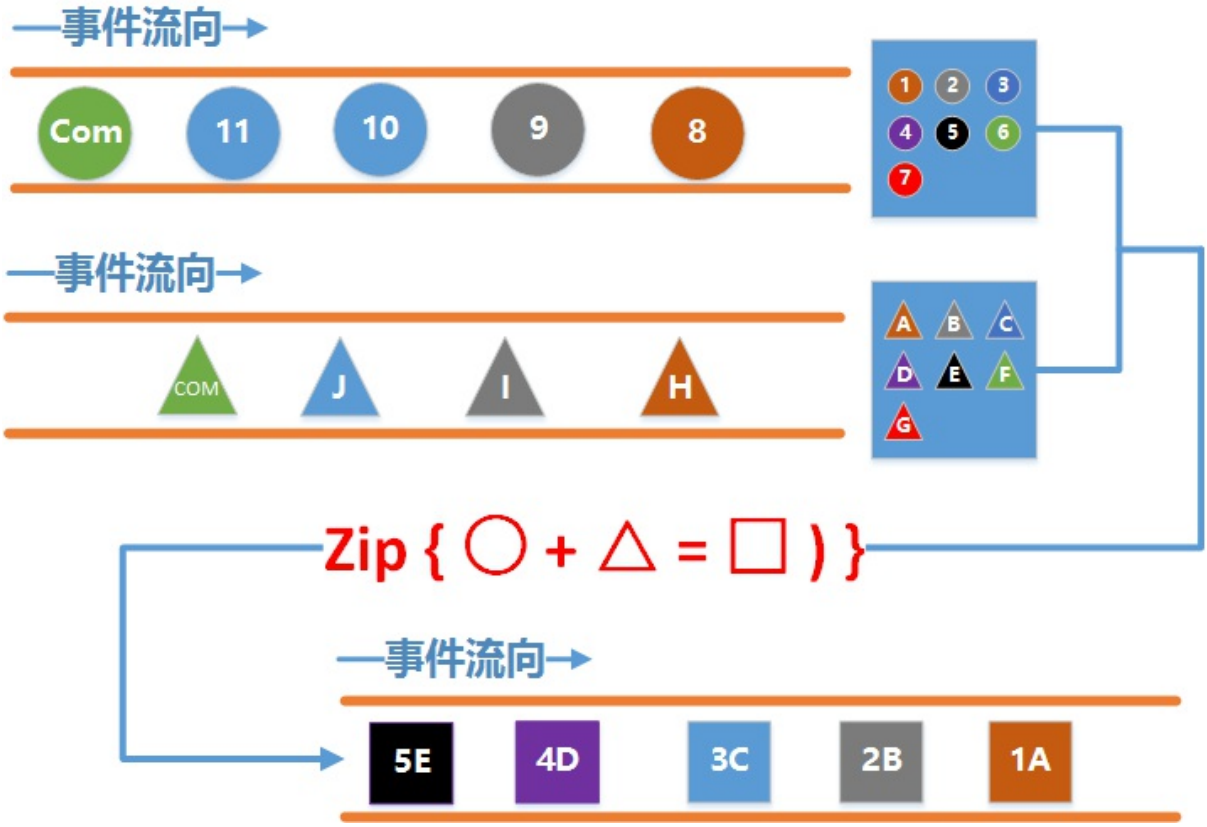
前言

大家喜闻乐见的 `Backpressure` 来啦.

这一节中我们将来学习 `Backpressure` . 我看好多吃瓜群众早已坐不住了, 别急, 我们先来回顾一下上一节讲的 `Zip` .

正题

上一节中我们说到Zip可以将多个上游发送的事件组合起来发送给下游, 那大家有没有想过一个问题, 如果其中一个 `水管A` 发送事件特别快, 而另一个 `水管B` 发送事件特别慢, 那就可能出现这种情况, 发得快的 `水管A` 已经发送了1000个事件了, 而发的慢的 `水管B` 才发一个出来, 组合了一个之后 `水管A` 还剩999个事件, 这些事件需要继续等待 `水管B` 发送事件出来组合, 那么这么多的事件是放在哪里的呢? 总有一个地方保存吧? 没错, Zip给我们的每一根水管都弄了一个 `水缸` , 用来保存这些事件, 用通俗易懂的图片来表示就是:



如图中所示, 其中蓝色的框框就是 `zip` 给我们的 `水缸` ! 它将每根水管发出的事件保存起来, 等两个水缸都有事件了之后就分别从水缸中取出一个事件来组合, 当其中一个水缸是空的时候就处于等待的状态.

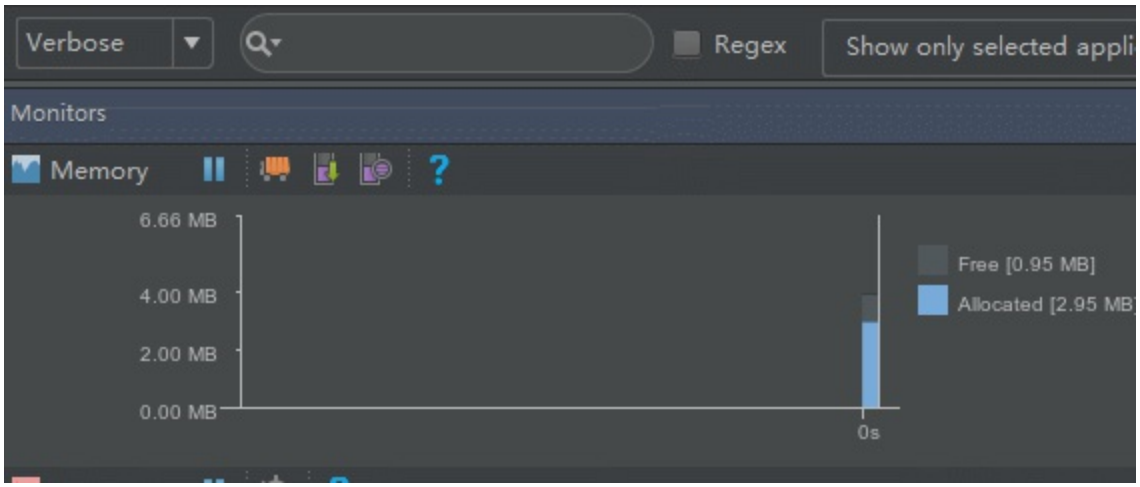
题外话: 大家来分析一下这个水缸有什么特点呢? 它是按顺序保存的, 先进来的事件先取出来, 这个特点是不是很熟悉呀? 没错, 这就是我们熟知的队列, 这个水缸在Zip内部的实现就是用的队列, 感兴趣的可以翻看源码查看.

好了回到正题上来, 这个水缸有大小限制吗? 要是一直往里存会怎样? 我们来看个例子:

```
Observable<Integer> observable1 = Observable.create(new ObservableOnSubscribe<Integer>(
) {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        for (int i = 0; ; i++) {   //无限循环发事件

            emitter.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io());

Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<String>()
{
    @Override
    public void subscribe(ObservableEmitter<String> emitter) throws Exception {
        emitter.onNext("A");
    }
}).subscribeOn(Schedulers.io());

Observable.zip(observable1, observable2, new BiFunction<Integer, String, String>() {

    @Override
    public String apply(Integer integer, String s) throws Exception {
        return integer + s;
    }
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<String>() {

    @Override
    public void accept(String s) throws Exception {
        Log.d(TAG, s);
    }
}, new Consumer<Throwable>() {
    @Override
    public void accept(Throwable throwable) throws Exception {
        Log.w(TAG, throwable);
    }
});
```

在这个例子中, 我们分别创建了两根水管, 第一根水管用机器指令的执行速度来无限循环发送事件, 第二根水管随便发送点什么, 由于我们没有发送 `Complete` 事件, 因此第一根水管会一直发事件到它对应的水缸里去, 我们来看看运行结果是什么样.

运行结果GIF图:



我勒个草, 内存占用以斜率为1的直线迅速上涨, 几秒钟就300多M , 最终报出了OOM:

```
zlc.season.rxjava2demo W/art: Throwing OutOfMemoryError "Failed to allocate a 28 byte a
llocation with
4194304 free bytes and 8MB until OOM;
zlc.season.rxjava2demo W/art: "main" prio=5 tid=1 Runnable
zlc.season.rxjava2demo W/art:   | group="main" sCount=0 dsCount=0 obj=0x75188710 self=0
x7fc0efe7ba00
zlc.season.rxjava2demo W/art:   | sysTid=32686 nice=0 cgrp=default sched=0/0 handle=0x7
fc0f37dc200
zlc.season.rxjava2demo W/art:   | state=R schedstat=( 0 0 0 ) utm=948 stm=120 core=1 HZ
=100
zlc.season.rxjava2demo W/art:   | stack=0x7fff971e8000-0x7fff971ea000 stackSize=8MB

zlc.season.rxjava2demo W/art:   | held mutexes= "mutator lock"(shared held)
zlc.season.rxjava2demo W/art:     at java.lang.Integer.valueOf(Integer.java:742)
```

出现这种情况肯定是我们不想看见的, 这里就可以引出我们的 `Backpressure` 了, 所谓的 `Backpressure` 其实就是为了控制流量, 水缸存储的能力毕竟有限, 因此我们还得从 `源头` 去解决问题, 既然你发那么快, 数据量那么大, 那我就想办法不让你发那么快呗.

那么这个 `源头` 到底在哪里, 究竟什么时候会出现这种情况, 这里只是说的Zip这一个例子, 其他的地方会出现吗? 带着这个问题我们来探究一下.

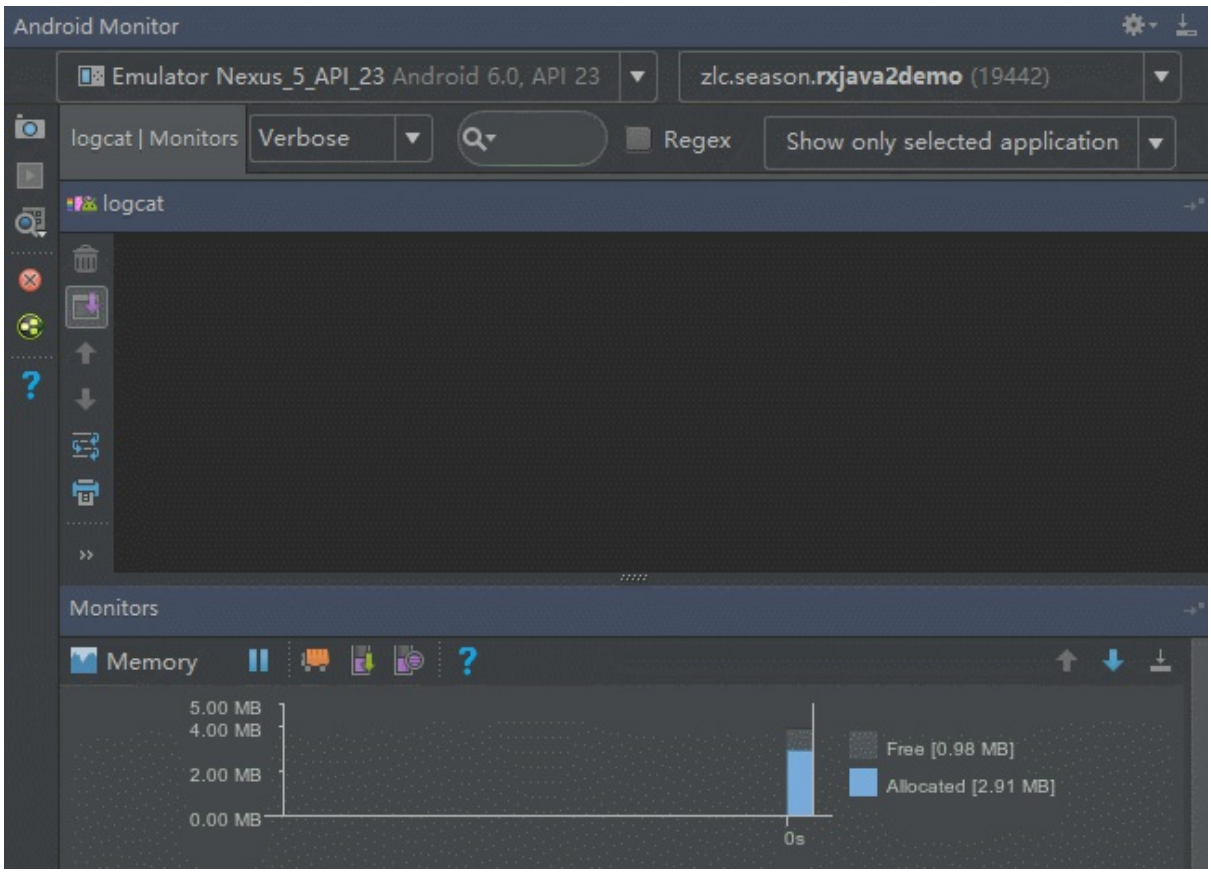
我们让事情变得简单一点, 从一个单一的 `Observable` 说起.

来看段代码:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        for (int i = 0; ; i++) {   //无限循环发事件
```

```
            emitter.onNext(i);
        }
    }
}).subscribe(new Consumer<Integer>() {
    @Override
    public void accept(Integer integer) throws Exception {
        Thread.sleep(2000);
        Log.d(TAG, "" + integer);
    }
});
```

这段代码很简单, 上游同样无限循环的发送事件, 在下游每次接收事件前延时2秒. 上下游工作在 `同一个线程` 里, 来看下运行结果:



哎卧槽, 怎么如此平静, 感觉像是走错了片场.

为什么呢, 因为上下游工作在 `同一个线程` 呀骚年们! 这个时候上游每次调用 `emitter.onNext(i)` 其实就相当于直接调用了Consumer中的:

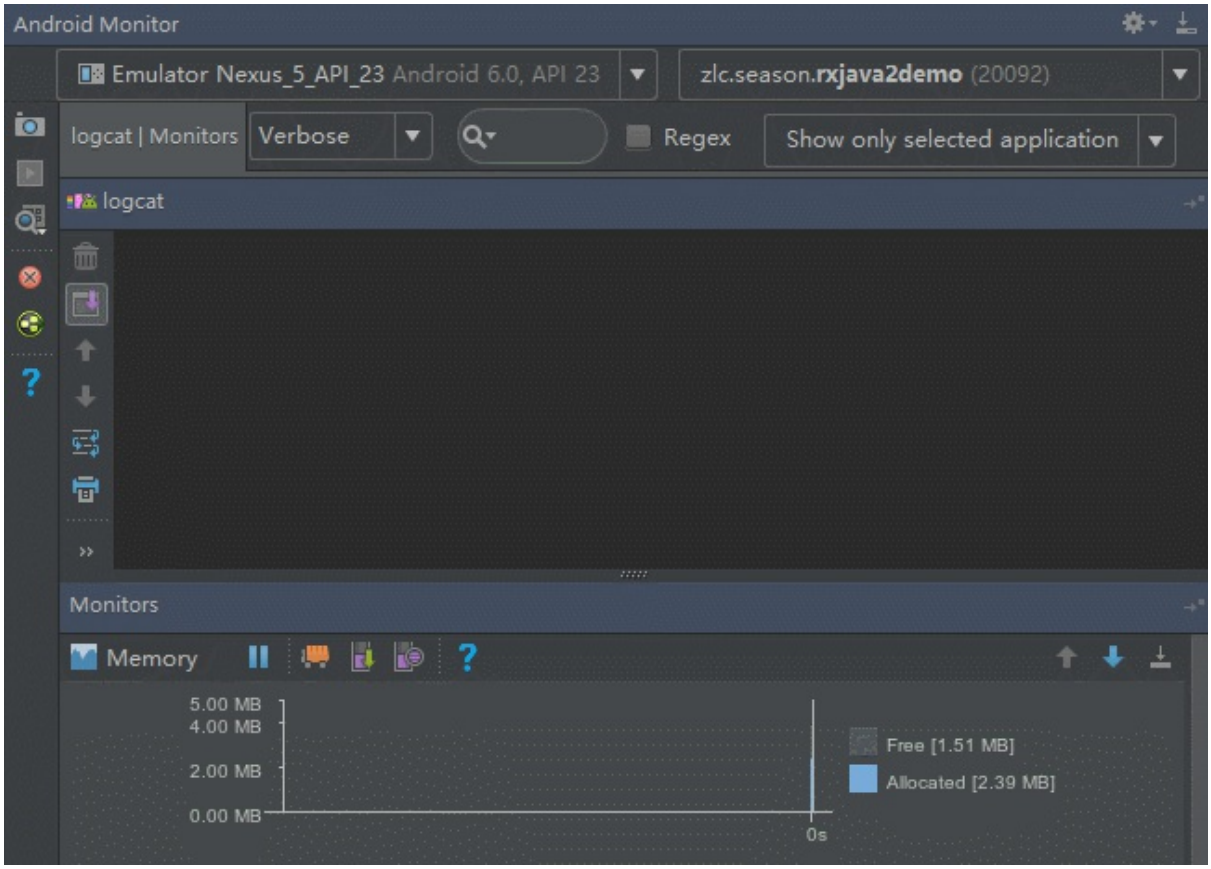
```
public void accept(Integer integer) throws Exception {
     Thread.sleep(2000);
     Log.d(TAG, "" + integer);
}
```

所以这个时候其实就是上游每延时2秒发送一次. 最终的结果也说明了这一切.

那我们加个线程呢, 改成这样:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {
        for (int i = 0; ; i++) {    //无限循环发事件

            emitter.onNext(i);
        }
    }
}).subscribeOn(Schedulers.io())
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<Integer>() {
            @Override
            public void accept(Integer integer) throws Exception {
                Thread.sleep(2000);
                Log.d(TAG, "" + integer);
            }
        });
```

这个时候把上游切换到了IO线程中去, 下游到主线程去接收, 来看看运行结果呢:



可以看到, 给上游加了个线程之后, 它就像脱缰的野马一样, 内存又爆掉了.

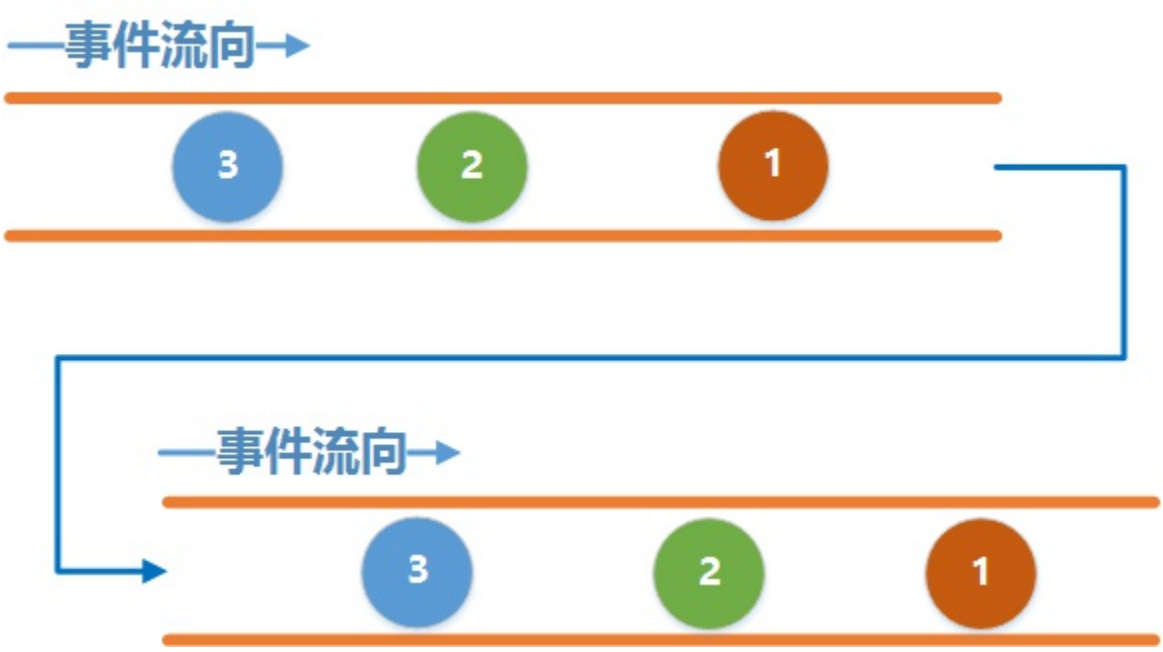
为什么不加线程和加上线程区别这么大呢, 这就涉及了 同步 和 异步 的知识了.

当上下游工作在 同一个线程 中时, 这时候是一个 同步 的订阅关系, 也就是说 上游 每发送一个事件 必须 等到 下游 接收处理完了以后才能接着发送下一个事件.

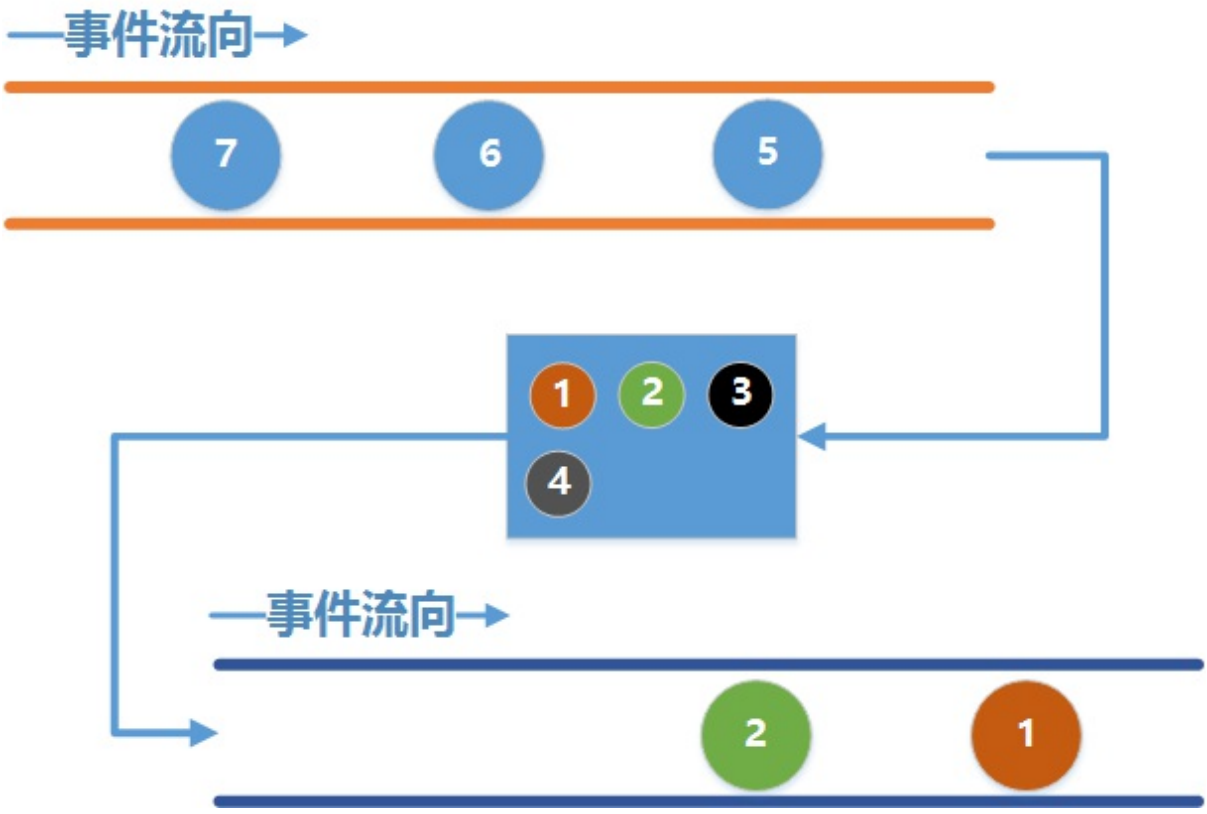
当上下游工作在 不同的线程 中时, 这时候是一个 异步 的订阅关系, 这个时候 上游 发送数据 不需要 等待 下游 接收, 为什么呢, 因为两个线程并不能直接进行通信, 因此上游发送的事件并不能直接到下游里去, 这个时候就需要一个田螺姑娘来帮助它们俩, 这个田螺姑娘就是我们刚才说的 水缸 ! 上游把事件发送到水缸里去, 下游从水缸里取出事件来处理, 因此, 当上游发事件的速度太快, 下游取事件的速度太慢, 水缸就会迅速装满, 然后溢出来, 最后就OOM了.

这两种情况用图片来表示如下:

同步:



异步:



从图中我们可以看出, `同步和异步` 的区别仅仅在于是否有 `水缸` .

相信通过这个例子大家对线程之间的通信也有了比较清楚的认知和理解.

源头找到了, 只要有 `水缸` , 就会出现上下游发送事件速度不平衡的情况, 因此当我们以后遇到这种情况时, 仔细思考一下水缸在哪里, 找到水缸, 你就找到了解决问题的办法.

既然源头找到了, 那么下一节我们就要来学习如何去解决了. 下节见

前言

在上一节中, 我们找到了上下游流速不均衡从而导致BackPressureException出现的源头 , 在这一节里我们将学习如何去治理它 . 可能很多看过其他人写的文章的朋友都会觉得只有 `Flowable` 才能解决 , 所以大家对这个 `Flowable` 都抱有很大的期许 , 其实呐 , 你们毕竟图样图森破 , 今天我们先抛开 `Flowable` , 仅仅依靠我们自己的 `双手和智慧` , 来看看我们如何去治理 , 通过本节的学习之后我们再来看 `Flowable` , 你会发现它其实并没有想象中那么牛叉, 它只是被其他人过度神化了.

正题

我们接着来看上一节的这个例子:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exception
{
                for (int i = 0; ; i++) {  //无限循环发送事件
                    emitter.onNext(i);
                }
            }
        }).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        Log.d(TAG, "" + integer);
                    }
                });
```

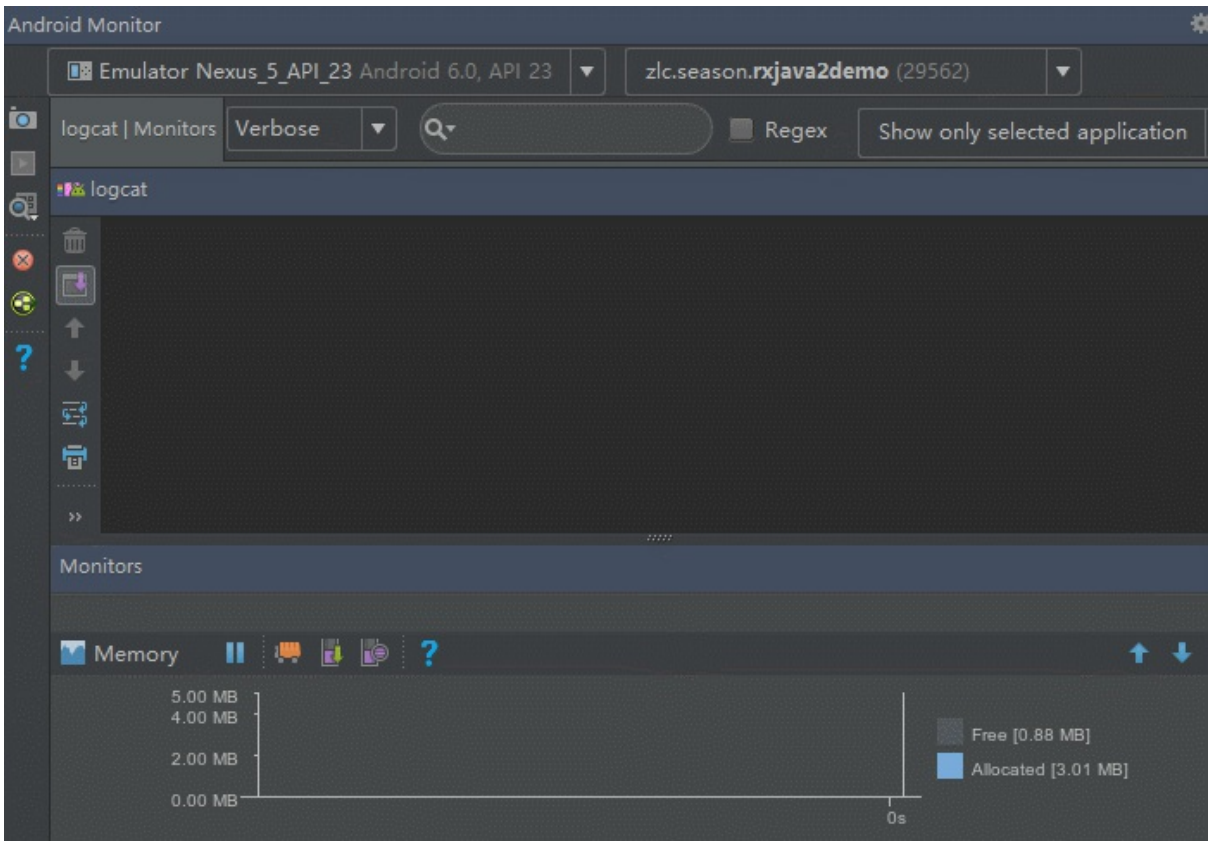
上一节中我们看到了它的运行结果是直接爆掉了内存, 也明白它为什么就爆掉了内存, 那么我们能做些什么, 才能不让这种情况发生呢.

之前我们说了, 上游发送的所有事件都放到水缸里了, 所以瞬间水缸就满了, 那我们可以只放我们需要的事件到水缸里呀, 只放一部分数据到水缸里, 这样不就不会溢出来了吗, 因此, 我们把上面的代码修改一下:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exception
{
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                }
            }
```

```
        }).subscribeOn(Schedulers.io())
                .filter(new Predicate<Integer>() {
                    @Override
                    public boolean test(Integer integer) throws Exception {
                        return integer % 10 == 0;
                    }
                })
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        Log.d(TAG, "" + integer);
                    }
                });
```

在这段代码中我们增加了一个filter, 只允许能被10整除的事件通过, 再来看看运行结果:



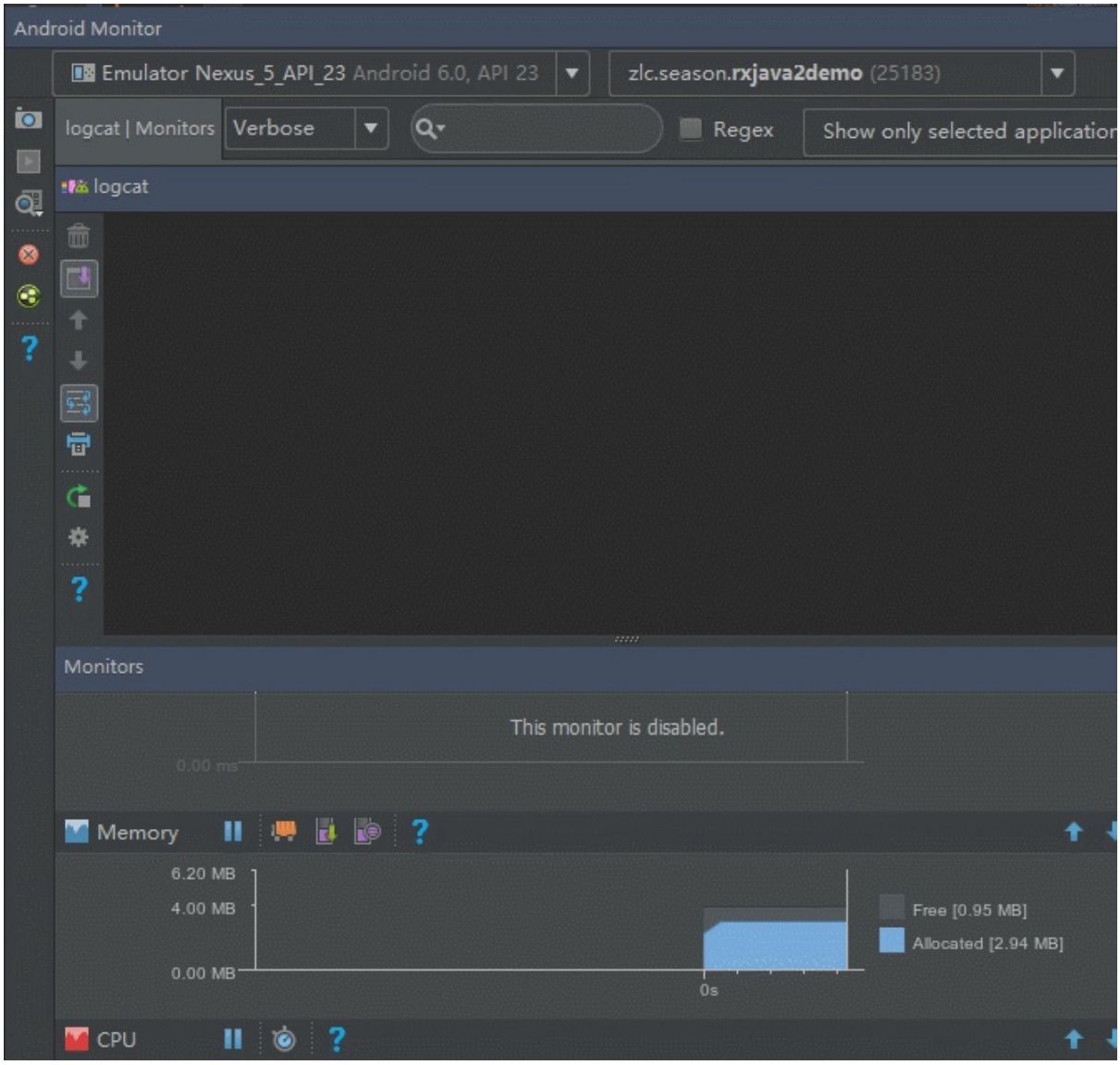
可以看到, 虽然内存依然在增长, 但是增长速度相比之前, 已经减少了太多了, 至少在我录完GIF之前还没有爆掉内存, 大家可以试着改成能被100整除试试.

可以看到, 通过减少进入水缸的事件数量的确可以缓解上下游流速不均衡的问题, 但是力度还不够, 我们再来看一段代码:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
```

```
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exception
{
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                }
            }
        }).subscribeOn(Schedulers.io())
                .sample(2, TimeUnit.SECONDS)  //sample取样
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        Log.d(TAG, "" + integer);
                    }
                });
```

这里用了一个`sample`操作符, 简单做个介绍, 这个操作符每隔指定的时间就从上游中取出一个事件发送给下游. 这里我们让它每隔2秒取一个事件给下游, 来看看这次的运行结果吧:



这次我们可以看到, 虽然上游仍然一直在不停的发事件, 但是我们只是 每隔一定时间 取一个放进水缸里, 并没有全部放进水缸里, 因此这次内存仅仅只占用了5M.

> 大家以后可以出去吹牛逼了: 我曾经通过技术手段去优化一个程序, 最终使得内存占用从300多M变成不到5M. ~(≧▽≦)/~

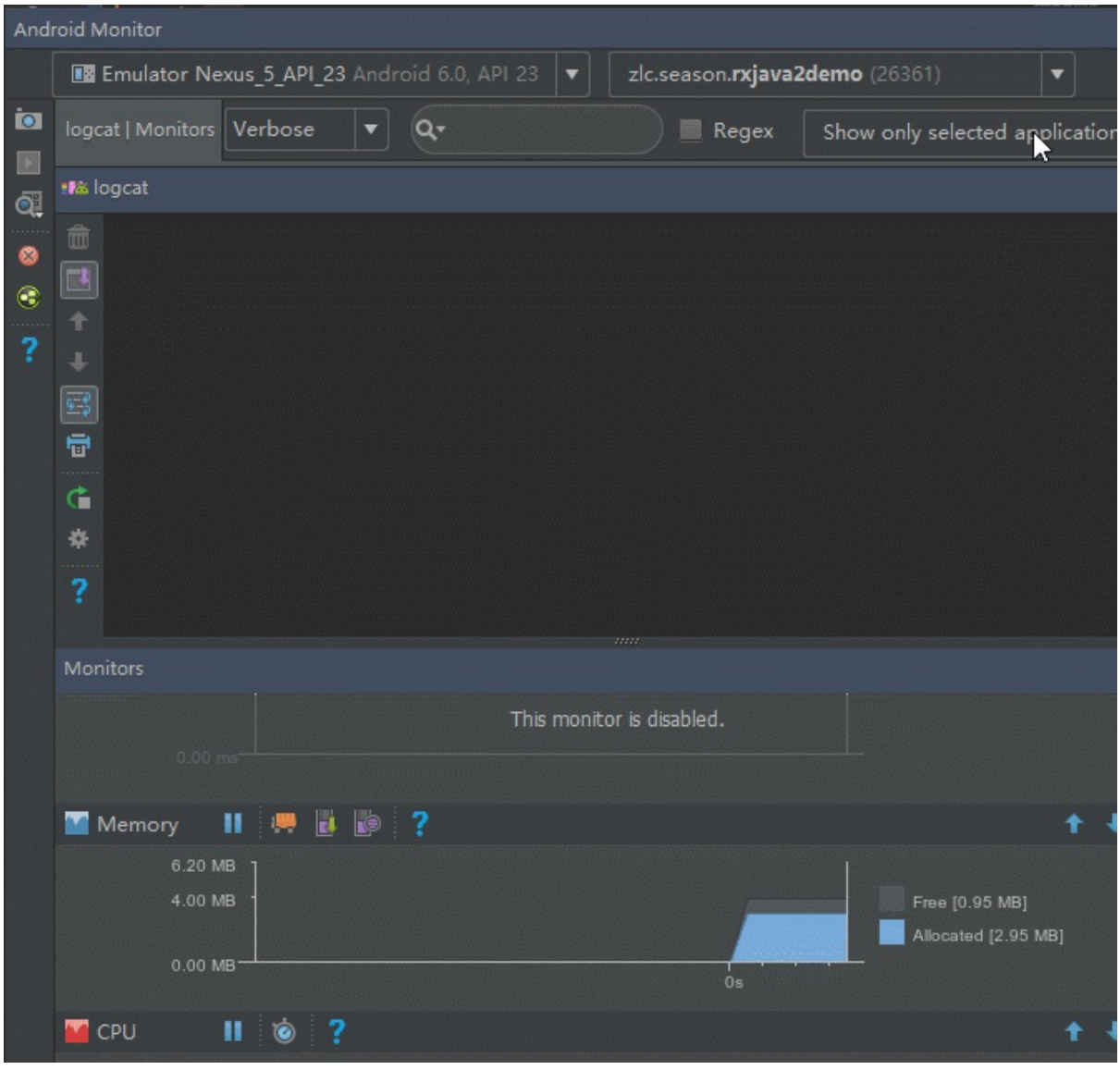
前面这两种方法归根到底其实就是减少放进水缸的事件的数量, 是以 数量 取胜, 但是这个方法有个 缺点 , 就是 丢失了大部分的事件 .

那么我们换一个角度来思考, 既然上游发送事件的速度太快, 那我们就适当减慢发送事件的速度, 从 速度 上取胜, 听上去不错, 我们来试试:

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exception
{
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
```

```
                    Thread.sleep(2000);  //每次发送完事件延时2秒
                }
            }
        }).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(Integer integer) throws Exception {
                        Log.d(TAG, "" + integer);
                    }
                });
```

这次我们让上游每次发送完事件后都延时了2秒, 来看看运行结果:



完美！一切都是那么完美！

可以看到, 我们给上游加上延时了之后, 瞬间一头发情的公牛就变得跟只小绵羊一样, 如此温顺, 如此平静, 如此平稳的内存线, 美妙极了. 而且 `事件也没有丢失` , `上游` 通过适当的 `延时` , 不但 `减缓了` 事件进入水缸的 `速度` , 也可以让 `下游` 有 `充足的时间` 从水缸里取出事件来处理 , 这样一来, 就不至于导致大量的事件涌进水缸, 也就不会OOM啦.

到目前为止, 我们没有依靠任何其他的工具, 就轻易解决了上下游流速不均衡的问题.

因此我们总结一下, 本节中的治理的办法就两种:

- 一是从数量上进行治理, 减少发送进水缸里的事件
- 二是从速度上进行治理, 减缓事件发送进水缸的速度

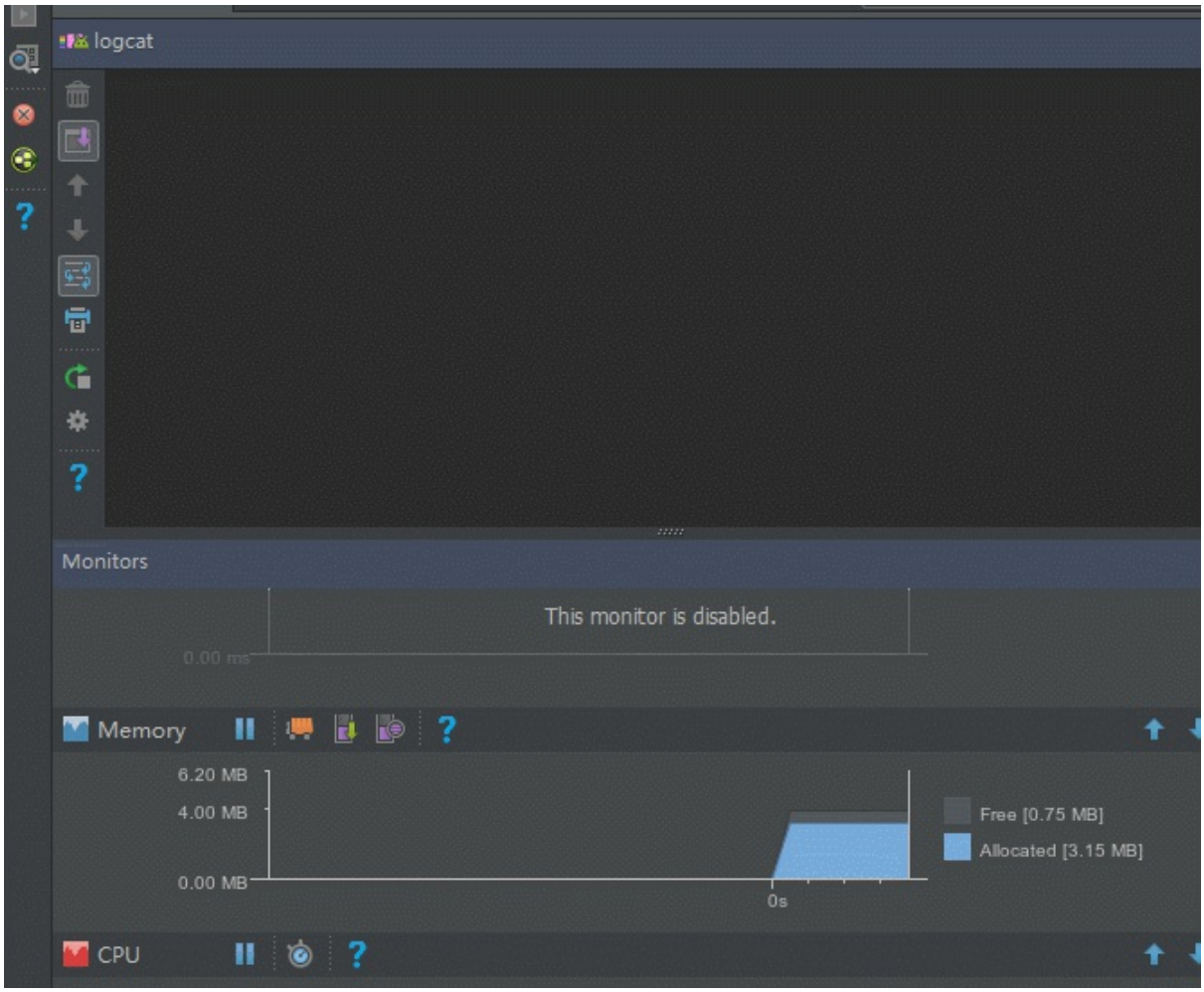
大家一定没忘记, 在上一节还有个Zip的例子, 这个例子也爆了我们的内存, 现学现用, 我们用刚学到的办法来试试能不能惩奸除恶, 先来看看第一种办法.

先来减少进入水缸的事件的数量:

```
Observable<Integer> observable1 = Observable.create(new ObservableOnSubscribe<Integer>(
) {
            @Override
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exception
{
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                }
            }
        }).subscribeOn(Schedulers.io()).sample(2, TimeUnit.SECONDS); //进行sample采样

        Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<St
ring>() {
            @Override
            public void subscribe(ObservableEmitter<String> emitter) throws Exception {
                emitter.onNext("A");
            }
        }).subscribeOn(Schedulers.io());

        Observable.zip(observable1, observable2, new BiFunction<Integer, String, String
>() {
            @Override
            public String apply(Integer integer, String s) throws Exception {
                return integer + s;
            }
        }).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                Log.d(TAG, s);
            }
        }, new Consumer<Throwable>() {
            @Override
```

```
            public void accept(Throwable throwable) throws Exception {
                Log.w(TAG, throwable);
            }
        });
```

来试试运行结果吧:



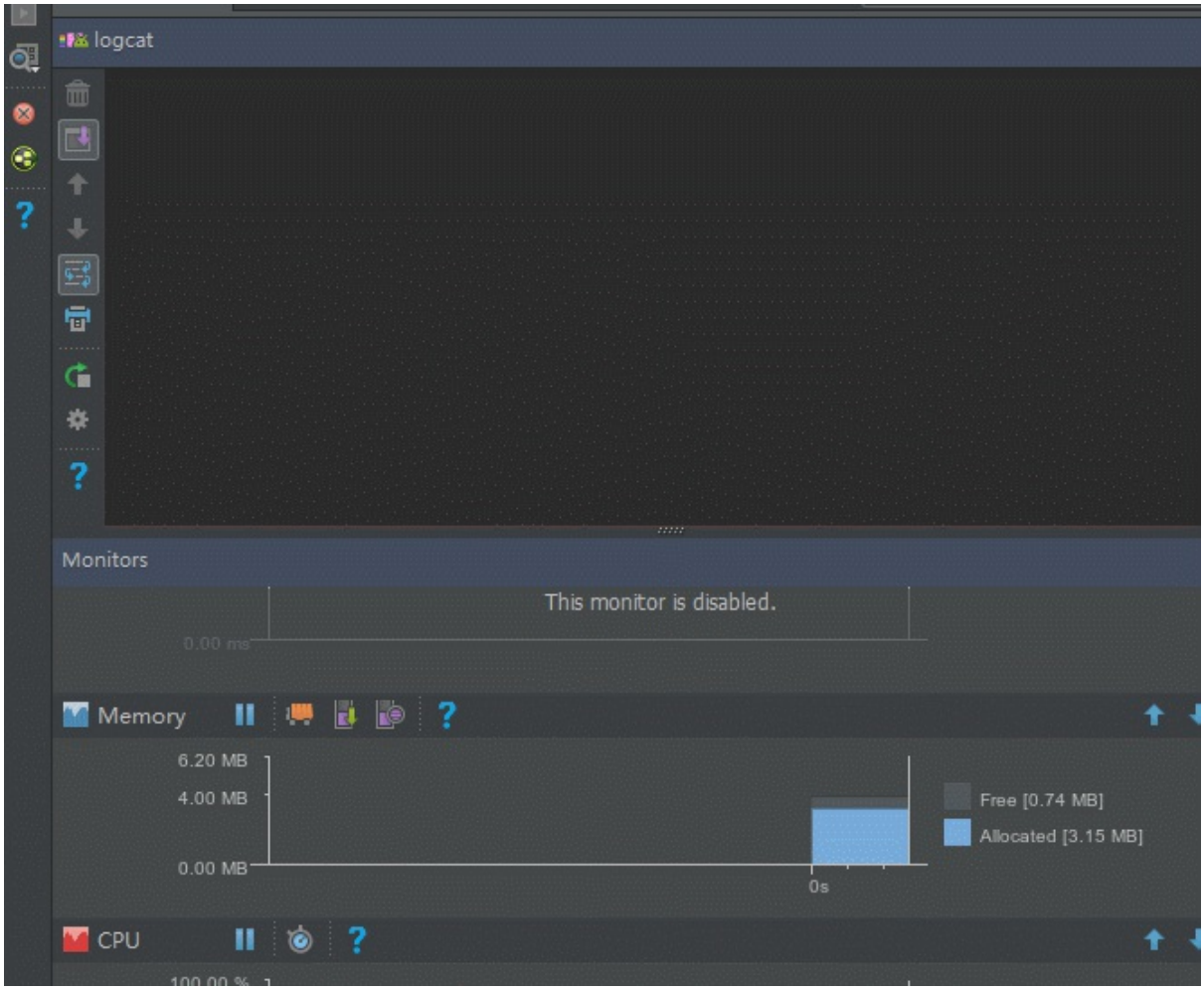
哈哈, 成功了吧, 再来用第二种办法试试.

这次我们来减缓速度:

```
Observable<Integer> observable1 = Observable.create(new ObservableOnSubscribe<Integer>(
) {
            @Override
            public void subscribe(ObservableEmitter<Integer> emitter) throws Exception
{
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                    Thread.sleep(2000);  //发送事件之后延时2秒
                }
            }
        }).subscribeOn(Schedulers.io());
```

```
        Observable<String> observable2 = Observable.create(new ObservableOnSubscribe<St
ring>() {
            @Override
            public void subscribe(ObservableEmitter<String> emitter) throws Exception {
                emitter.onNext("A");
            }
        }).subscribeOn(Schedulers.io());

        Observable.zip(observable1, observable2, new BiFunction<Integer, String, String
>() {
            @Override
            public String apply(Integer integer, String s) throws Exception {
                return integer + s;
            }
        }).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<String>() {
            @Override
            public void accept(String s) throws Exception {
                Log.d(TAG, s);
            }
        }, new Consumer<Throwable>() {
            @Override
            public void accept(Throwable throwable) throws Exception {
                Log.w(TAG, throwable);
            }
        });
```

来看看运行结果吧:



果然也成功了, 这里只打印出了下游收到的事件, 所以只有一个. 如果你对这个结果看不懂, 请自觉掉头看前面几篇文章.

通过本节的学习, 大家应该对如何处理上下游流速不均衡已经有了基本的认识了, 大家也可以看到, 我们并没有使用 `Flowable` , 所以很多时候仔细去分析问题, 找到问题的原因, 从源头去解决才是最根本的办法. 后面我们讲到 `Flowable` 的时候, 大家就会发现它其实没什么神秘的, 它用到的办法和我们本节所讲的基本上是一样的, 只是它稍微做了点封装.

好了, 今天的教程就到这里吧, 下一节中我们就会来学习你们喜闻乐见的 `Flowable`

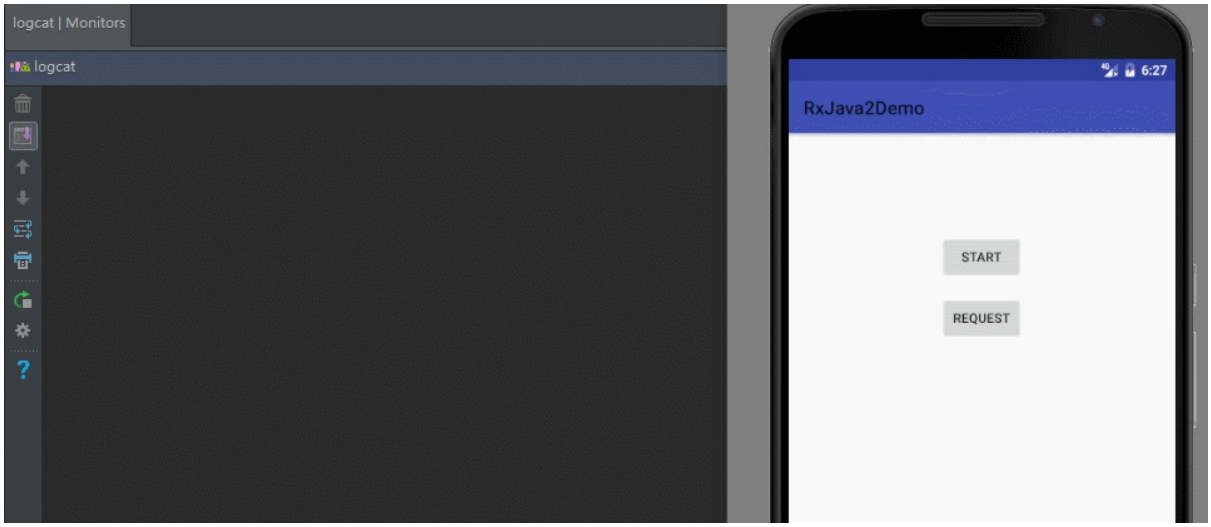
前言

上一节里我们学习了只使用 `Observable` 如何去解决上下游流速不均衡的问题, 之所以学习这个是因为 `Observable` 还是有很多它使用的场景, 有些朋友自从听说了 `Flowable` 之后就觉得 `Flowable` 能解决任何问题, 甚至有抛弃 `Observable` 这种想法, 这是万万不可的, 它们都有各自的优势和不足.

在这一节里我们先来学习如何使用 `Flowable` , 它东西比较多, 也比较繁琐, 解释起来也比较麻烦, 但我还是尽量用 `通俗易懂` 的话来说清楚, 毕竟, 这是一个 `通俗易懂` 的教程.

正题

我们还是以两根水管举例子:



之前我们所的上游和下游分别是 `Observable` 和 `Observer` , 这次不一样的是上游变成了 `Flowable` , 下游变成了 `Subscriber` , 但是水管之间的连接还是通过 `subscribe()` , 我们来看看最基本的用法吧:

```
Flowable<Integer> upstream = Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                Log.d(TAG, "emit 1");
                emitter.onNext(1);
                Log.d(TAG, "emit 2");
                emitter.onNext(2);
                Log.d(TAG, "emit 3");
                emitter.onNext(3);
                Log.d(TAG, "emit complete");
                emitter.onComplete();
            }
        }, BackpressureStrategy.ERROR); //增加了一个参数
```

```
        Subscriber<Integer> downstream = new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
                s.request(Long.MAX_VALUE);  //注意这句代码
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);

            }

            @Override
            public void onError(Throwable t) {
                 Log.w(TAG, "onError: ", t);
            }

            @Override
            public void onComplete() {
                Log.d(TAG, "onComplete");
            }
        };

        upstream.subscribe(downstream);
```

这段代码中,分别创建了一个上游 `Flowable` 和下游 `Subscriber` , 上下游工作在同一个线程中, 和之前的 `Observable` 的使用方式只有一点点的区别, 先来看看运行结果吧:

```
D/TAG: onSubscribe
D/TAG: emit 1
D/TAG: onNext: 1
D/TAG: emit 2
D/TAG: onNext: 2
D/TAG: emit 3
D/TAG: onNext: 3
D/TAG: emit complete
D/TAG: onComplete
```

结果也和我们预期的是一样的.

我们注意到这次和 `Observable` 有些不同. 首先是创建 `Flowable` 的时候增加了一个参数, 这个参数是用来选择背压,也就是出现上下游流速不均衡的时候应该怎么处理的办法, 这里我们直接用 `BackpressureStrategy.ERROR` 这种方式, 这种方式会在出现上下游流速不均衡的时候直接抛出一个异常,这个异常就是著名的 `MissingBackpressureException` . 其余的策略后面再来讲解.

另外的一个区别是在下游的 `onSubscribe` 方法中传给我们的不再是 `Disposable` 了, 而是 `Subscription` , 它俩有什么区别呢, 首先它们都是上下游中间的一个开关, 之前我们说调用 `Disposable.dispose()` 方法可以切断水管, 同样的调用 `Subscription.cancel()` 也可以切断水管, 不同的地方在于 `Subscription` 增加了一个 `void request(long n)` 方法, 这个方法有什么用呢, 在上面的代码中也有这么一句代码:

```
s.request(Long.MAX_VALUE);
```

这句代码有什么用呢, 不要它可以吗? 我们来试试:

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                Log.d(TAG, "emit 1");
                emitter.onNext(1);
                Log.d(TAG, "emit 2");
                emitter.onNext(2);
                Log.d(TAG, "emit 3");
                emitter.onNext(3);
                Log.d(TAG, "emit complete");
                emitter.onComplete();
            }
        }, BackpressureStrategy.ERROR).subscribe(new Subscriber<Integer>() {

            @Override
            public void onSubscribe(Subscription s) {
                Log.d(TAG, "onSubscribe");
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "onNext: " + integer);

            }

            @Override
            public void onError(Throwable t) {
                Log.w(TAG, "onError: ", t);
            }

            @Override
            public void onComplete() {
                Log.d(TAG, "onComplete");
            }
        });
```

这次我们取消掉了request这句代码, 来看看运行结果:

```
zlc.season.rxjava2demo D/TAG: onSubscribe
zlc.season.rxjava2demo D/TAG: emit 1
zlc.season.rxjava2demo W/TAG: onError:
io.reactivex.exceptions.MissingBackpressureException: create: could not emit value due
to lack of requests
    at io.reactivex.internal.operators.flowable.FlowableCreate$ErrorAsyncEmitter.onOver
flow(FlowableCreate.java:411)
    at io.reactivex.internal.operators.flowable.FlowableCreate$NoOverflowBaseAsyncEmitt
er.onNext(FlowableCreate.java:377)
    at zlc.season.rxjava2demo.demo.ChapterSeven$3.subscribe(ChapterSeven.java:77)
    at io.reactivex.internal.operators.flowable.FlowableCreate.subscribeActual(Flowable
Create.java:72)
    at io.reactivex.Flowable.subscribe(Flowable.java:12218)
    at zlc.season.rxjava2demo.demo.ChapterSeven.demo2(ChapterSeven.java:111)
    at zlc.season.rxjava2demo.MainActivity$2.onClick(MainActivity.java:36)
    at android.view.View.performClick(View.java:5637)
    at android.view.View$PerformClick.run(View.java:22429)
    at android.os.Handler.handleCallback(Handler.java:751)
    at android.os.Handler.dispatchMessage(Handler.java:95)
    at android.os.Looper.loop(Looper.java:154)
    at android.app.ActivityThread.main(ActivityThread.java:6119)
    at java.lang.reflect.Method.invoke(Native Method)
    at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.java:886)
    at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:776)
zlc.season.rxjava2demo D/TAG: emit 2
zlc.season.rxjava2demo D/TAG: emit 3
zlc.season.rxjava2demo D/TAG: emit complete
```

哎哎哎, 大兄弟, 怎么一言不合就抛异常?

从运行结果中可以看到, 在 `上游` 发送 `第一个事件` 之后, `下游` 就抛出了一个著名的 `MissingBackpressureException` 异常, 并且 `下游` 没有收到 `任何其余的事件` . 可是这是一个 `同步` 的订阅呀, 上下游工作在 `同一个线程` , 上游每发送一个事件应该会等待下游处理完了才会继续发事件啊, 不可能出现上下游流速不均衡的问题呀.

带着这个疑问, 我们再来看看异步的情况:

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                Log.d(TAG, "emit 1");
                emitter.onNext(1);
                Log.d(TAG, "emit 2");
                emitter.onNext(2);
                Log.d(TAG, "emit 3");
```

```
                emitter.onNext(3);
                Log.d(TAG, "emit complete");
                emitter.onComplete();
            }
        }, BackpressureStrategy.ERROR).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
```

这次我们同样去掉了request这句代码, 但是让上下游工作在不同的线程, 来看看运行结果:

```
zlc.season.rxjava2demo D/TAG: onSubscribe
zlc.season.rxjava2demo D/TAG: emit 1
zlc.season.rxjava2demo D/TAG: emit 2
zlc.season.rxjava2demo D/TAG: emit 3
zlc.season.rxjava2demo D/TAG: emit complete
```

哎, 这次上游正确的发送了所有的事件, 但是下游一个事件也没有收到. 这是因为什么呢?

这是因为 `Flowable` 在设计的时候采用了一种新的思路也就是 `响应式拉取` 的方式来更好的解决上下游流速不均衡的问题, 与我们之前所讲的 `控制数量` 和 `控制速度` 不太一样, 这种方式用通俗易懂的话来说就好比是 `叶问打鬼子` , 我们把 `上游` 看成 `小日本` , 把 `下游` 当作 `叶问` , 当调用 `Subscription.request(1)` 时, `叶问` 就说 `我要打一个!` 然后 `小日本` 就拿出 `一个鬼子` 给叶问, 让

他打, 等叶问打死这个鬼子之后, 再次调用 `request(10)` , 叶问就又说 `我要打十个!` 然后小日本又派出 `十个鬼子` 给叶问, 然后就在边上看热闹, 看叶问能不能打死十个鬼子, 等叶问打死十个鬼子后再继续要鬼子接着打...

所以我们把request当做是一种能力, 当成 `下游处理事件` 的能力, 下游能处理几个就告诉上游我要几个, 这样只要上游根据下游的处理能力来决定发送多少事件, 就不会造成一窝蜂的发出一堆事件来, 从而导致OOM. 这也就完美的解决之前我们所学到的两种方式的缺陷, 过滤事件会导致事件丢失, 减速又可能导致性能损失. 而这种方式既解决了事件丢失的问题, 又解决了速度的问题, 完美！

但是太完美的东西也就意味着陷阱也会很多, 你可能只是被它的外表所迷惑, 失去了理智, 如果你滥用或者不遵守规则, 一样会吃到苦头.

比如这里需要注意的是, 只有当 `上游正确的实现了` 如何 `根据下游的处理能力` 来发送事件的时候, 才能达到这种效果, 如果上游根本不管下游的处理能力, 一股脑的瞎他妈发事件, 仍然会产生上下游流速不均衡的问题, 这就好比小日本管他叶问要打几个, 老子直接拿出1万个鬼子, 这尼玛有种打死给我看看? 那么 `如何正确的去实现上游` 呢, 这里先卖个关子, 之后我们再来讲解.

学习了request, 我们就可以解释上面的两段代码了.

首先第一个同步的代码, 为什么上游发送第一个事件后下游就抛出了 `MissingBackpressureException` 异常, 这是因为下游没有调用request, 上游就认为下游没有处理事件的能力, 而这又是一个同步的订阅, 既然下游处理不了, 那上游不可能一直等待吧, 如果是这样, 万一这两根水管工作在主线程里, 界面不就卡死了吗, 因此只能抛个异常来提醒我们. 那如何解决这种情况呢, 很简单啦, 下游直接调用request(Long.MAX_VALUE)就行了, 或者根据上游发送事件的数量来request就行了, 比如这里request(3)就可以了.

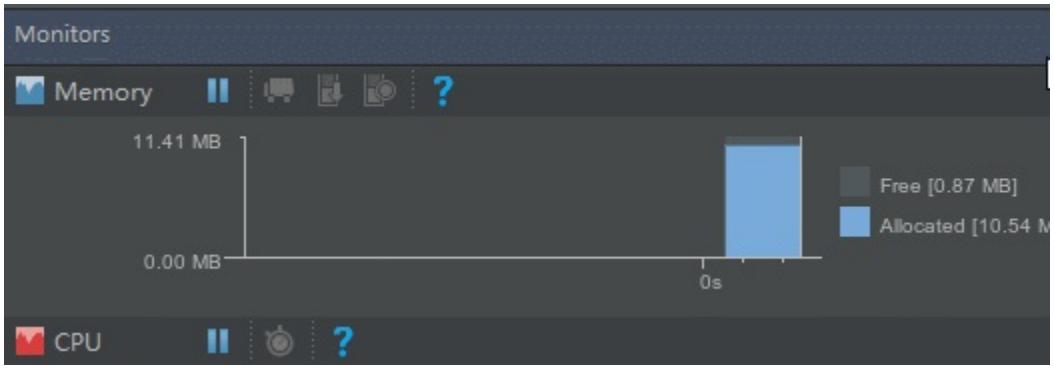
然后我们再来看看第二段代码, 为什么上下游没有工作在同一个线程时, 上游却正确的发送了所有的事件呢? 这是因为在 `Flowable` 里默认有一个 `大小为128` 的水缸, 当上下游工作在不同的线程中时, 上游就会先把事件发送到这个水缸中, 因此, 下游虽然没有调用request, 但是上游在水缸中保存着这些事件, 只有当下游调用request时, 才从水缸里取出事件发给下游.

是不是这样呢, 我们来验证一下:

```
public static void request(long n) {
    mSubscription.request(n); //在外部调用request请求上游
}

public static void demo3() {
    Flowable.create(new FlowableOnSubscribe<Integer>() {
        @Override
        public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
            Log.d(TAG, "emit 1");
            emitter.onNext(1);
            Log.d(TAG, "emit 2");
            emitter.onNext(2);
            Log.d(TAG, "emit 3");
            emitter.onNext(3);
```

```
            Log.d(TAG, "emit complete");
            emitter.onComplete();
        }
    }, BackpressureStrategy.ERROR).subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Subscriber<Integer>() {

                @Override
                public void onSubscribe(Subscription s) {
                    Log.d(TAG, "onSubscribe");
                    mSubscription = s;  //把Subscription保存起来
                }

                @Override
                public void onNext(Integer integer) {
                    Log.d(TAG, "onNext: " + integer);
                }

                @Override
                public void onError(Throwable t) {
                    Log.w(TAG, "onError: ", t);
                }

                @Override
                public void onComplete() {
                    Log.d(TAG, "onComplete");
                }
            });
}
```

这里我们把Subscription保存起来, 在界面上增加了一个按钮, 点击一次就调用 `Subscription.request(1)` , 来看看运行结果:



结果似乎像那么回事, 上游发送了四个事件保存到了水缸里, 下游每request一个, 就接收一个进行处理.

刚刚我们有说到水缸的大小为128, 有朋友就问了, 你说128就128吗, 又不是唯品会周年庆, 我不信. 那就来验证一下:

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                for (int i = 0; i < 128; i++) {
                    Log.d(TAG, "emit " + i);
                    emitter.onNext(i);
                }
            }
        }, BackpressureStrategy.ERROR).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
```

这里我们让上游一次性发送了128个事件, 下游一个也不接收, 来看看运行结果:

```
zlc.season.rxjava2demo D/TAG: onSubscribe
zlc.season.rxjava2demo D/TAG: emit 0
  ...
zlc.season.rxjava2demo D/TAG: emit 126
zlc.season.rxjava2demo D/TAG: emit 127
```

这段代码的运行结果很正常, 没有任何错误和异常, 上游仅仅是发送了128个事件.

那来试试129个呢, 把上面代码中的128改成129试试:

```
zlc.season.rxjava2demo D/TAG: onSubscribe
```

```
zlc.season.rxjava2demo D/TAG: emit 0
  ...
zlc.season.rxjava2demo D/TAG: emit 126
zlc.season.rxjava2demo D/TAG: emit 127
zlc.season.rxjava2demo D/TAG: emit 128  //这是第129个事件
zlc.season.rxjava2demo W/TAG: onError:
io.reactivex.exceptions.MissingBackpressureException: create: could not emit value due
to lack of requests
    at io.reactivex.internal.operators.flowable.FlowableCreate$ErrorAsyncEmitter.onOver
flow(FlowableCreate.java:411)
    at io.reactivex.internal.operators.flowable.FlowableCreate$NoOverflowBaseAsyncEmitt
er.onNext(FlowableCreate.java:377)
    at zlc.season.rxjava2demo.demo.ChapterSeven$7.subscribe(ChapterSeven.java:169)
    at io.reactivex.internal.operators.flowable.FlowableCreate.subscribeActual(Flowable
Create.java:72)
    at io.reactivex.Flowable.subscribe(Flowable.java:12218)
    at io.reactivex.internal.operators.flowable.FlowableSubscribeOn$SubscribeOnSubscrib
er.run(FlowableSubscribeOn.java:82)
    at io.reactivex.internal.schedulers.ScheduledRunnable.run(ScheduledRunnable.java:59
)
    at io.reactivex.internal.schedulers.ScheduledRunnable.call(ScheduledRunnable.java:5
1)
    at java.util.concurrent.FutureTask.run(FutureTask.java:237)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(Schedul
edThreadPoolExecutor.java:272)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1133)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:607)
    at java.lang.Thread.run(Thread.java:761)
```

这次可以看到, 在上游发送了第129个事件的时候, 就抛出了 `MissingBackpressureException` 异常, 提醒我们发洪水啦. 当然了, 这个128也不是我凭空捏造出来的, Flowable的源码中就有这个buffersize的大小定义, 可以自行查看.

注意这里我们是把上游发送的事件全部都存进了水缸里, 下游一个也没有消费, 所以就溢出了, 如果下游去消费了事件, `可能` 就不会导致水缸溢出来了. 这里我们说的是可能不会, 这也很好理解, 比如刚才这个例子上游发了129个事件, 下游只要快速的消费了一个事件, 就不会溢出了, 但如果下游过了十秒钟再来消费一个, 那肯定早就溢出了.

好了, 今天的教程就到这里了, 下一节我们将会更加深入的去学习FLowable, 敬请期待.

(哈哈, 给我的RxDownload打个广告: RxDownload是一个基于RxJava的多线程+断点续传的下载工具, 感兴趣的来GitHub点个star吧. 电梯直达->戳这里)

前言

在上一节中, 我们学习了FLowable的一些基本知识, 同时也挖了许多坑, 这一节就让我们来填坑吧.

正题

在上一节中最后我们有个例子, 当上游一次性发送128个事件的时候是没有任何问题的, 一旦超过128就会抛出 `MissingBackpressureException` 异常, 提示你上游发太多事件了, 下游处理不过来, 那么怎么去解决呢?

我们先来思考一下, 发送128个事件没有问题是因为 `FLowable` 内部有一个大小为128的水缸, 超过128就会装满溢出来, 那既然你水缸这么小, 那我给你换一个 `大水缸` 如何, 听上去很有道理的样子, 来试试:

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                for (int i = 0; i < 1000; i++) {
                    Log.d(TAG, "emit " + i);
                    emitter.onNext(i);
                }
            }
        }, BackpressureStrategy.BUFFER).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
```

```
                }
            });
```

这次我们直接让上游发送了1000个事件,下游仍然不调用request去请求, 与之前不同的是, 这次我们用的策略是 `BackpressureStrategy.BUFFER` , 这就是我们的 `新水缸` 啦, 这个水缸就比原来的水缸牛逼多了,如果说原来的水缸是95式步枪, 那这个新的水缸就好比黄金AK , 它没有大小限制, 因此可以存放许许多多的事件.

所以这次的运行结果就是:

```
zlc.season.rxjava2demo D/TAG: onSubscribe
zlc.season.rxjava2demo D/TAG: emit 0
zlc.season.rxjava2demo D/TAG: emit 1
zlc.season.rxjava2demo D/TAG: emit 2
...
zlc.season.rxjava2demo D/TAG: emit 997
zlc.season.rxjava2demo D/TAG: emit 998
zlc.season.rxjava2demo D/TAG: emit 999
```

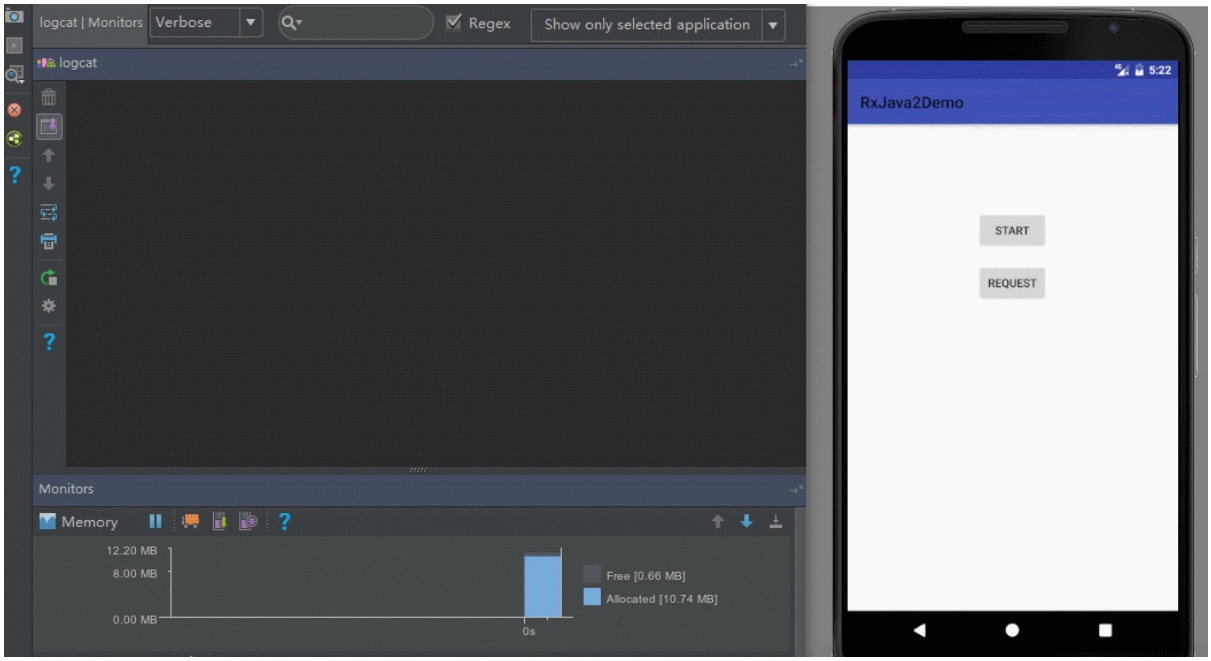
不知道大家有没有发现, 换了水缸的FLowable和Observable好像是一样的嘛...

不错, 这时的FLowable表现出来的特性的确和Observable一模一样, 因此, 如果你像这样单纯的使用FLowable, 同样需要注意OOM的问题, 例如下面这个例子:

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                }
            }
        }, BackpressureStrategy.BUFFER).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
```

```
                public void onError(Throwable t) {
                    Log.w(TAG, "onError: ", t);
                }

                @Override
                public void onComplete() {
                    Log.d(TAG, "onComplete");
                }
            });
```

按照我们以前学习Observable一样, 让上游无限循环发送事件, 下游一个也不去处理, 来看看运行结果吧:



同样可以看到, 内存迅速增长, 直到最后抛出OOM. 所以说不要迷恋FLowable, 它只是个传说.

> 可能有朋友也注意到了, 之前使用Observable测试的时候内存增长非常迅速, 几秒钟就OOM, 但这里增长速度却比较缓慢, 可以翻回去看之前的文章中的GIF图进行对比, 这也看出FLowable相比Observable, 在性能方面有些不足, 毕竟FLowable内部为了实现响应式拉取做了更多的操作, 性能有所丢失也是在所难免, 因此单单只是说因为FLowable是新兴产物就盲目的使用也是不对的, 也要具体分场景,

那除了给FLowable换一个大水缸还有没有其他的办法呢, 因为更大的水缸也只是缓兵之计啊, 动不动就OOM给你看.

想想看我们之前学习Observable的时候说到的如何解决上游发送事件太快的, 有一招叫从 `数量` 上取胜, 同样的FLowable中也有这种方法, 对应的就
是 `BackpressureStrategy.DROP` 和 `BackpressureStrategy.LATEST` 这两种策略.

从名字上就能猜到它俩是干啥的, Drop就是直接把存不下的事件丢弃,Latest就是只保留最新的事件, 来看看它们的实际效果吧.

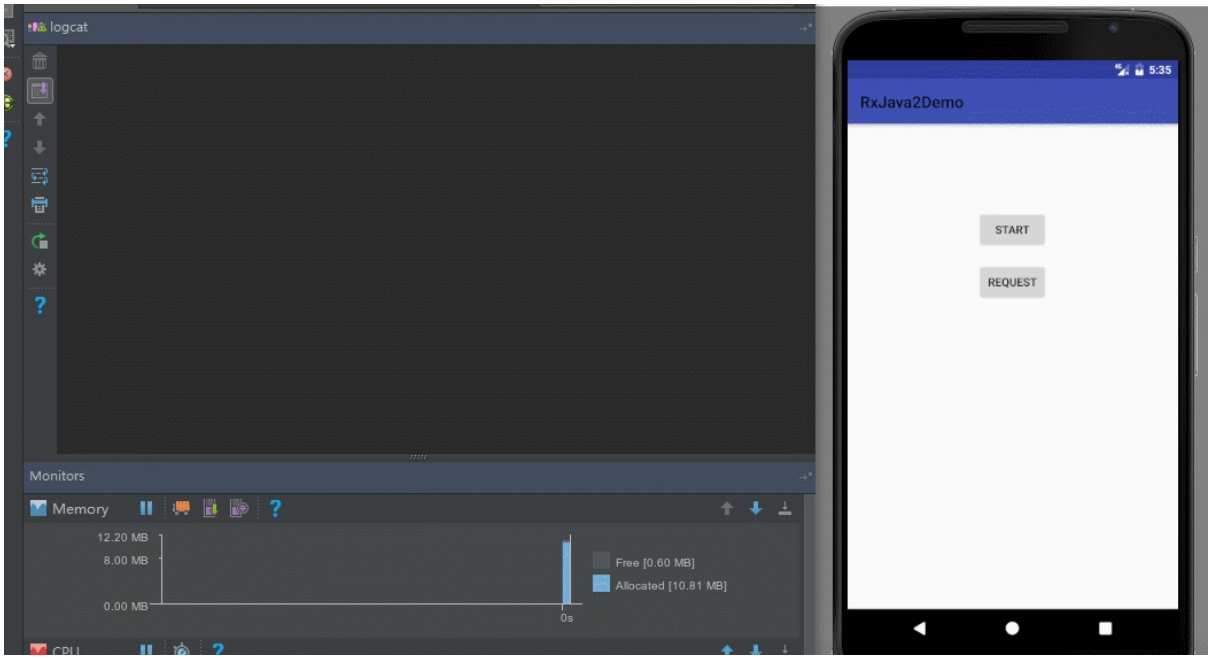
先来看看Drop:

```
public static void request() {
        mSubscription.request(128);
    }

public static void demo3() {
        Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                }
            }
        }, BackpressureStrategy.DROP).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

我们仍然让上游无限循环发送事件, 这次的策略选择了Drop, 同时把Subscription保存起来, 待会我们在外部调用request(128)时, 便可以看到运行的结果.

我们先来猜一下运行结果, 这里为什么request(128)呢, 因为之前不是已经说了吗, FLowable内部的默认的水缸大小为128, 因此, 它刚开始肯定会把0-127这128个事件保存起来, 然后丢弃掉其余的事件, 当我们request(128)的时候,下游便会处理掉这128个事件, 那么上游水缸中又会重新装进新的128个

事件, 以此类推, 来看看运行结果吧:



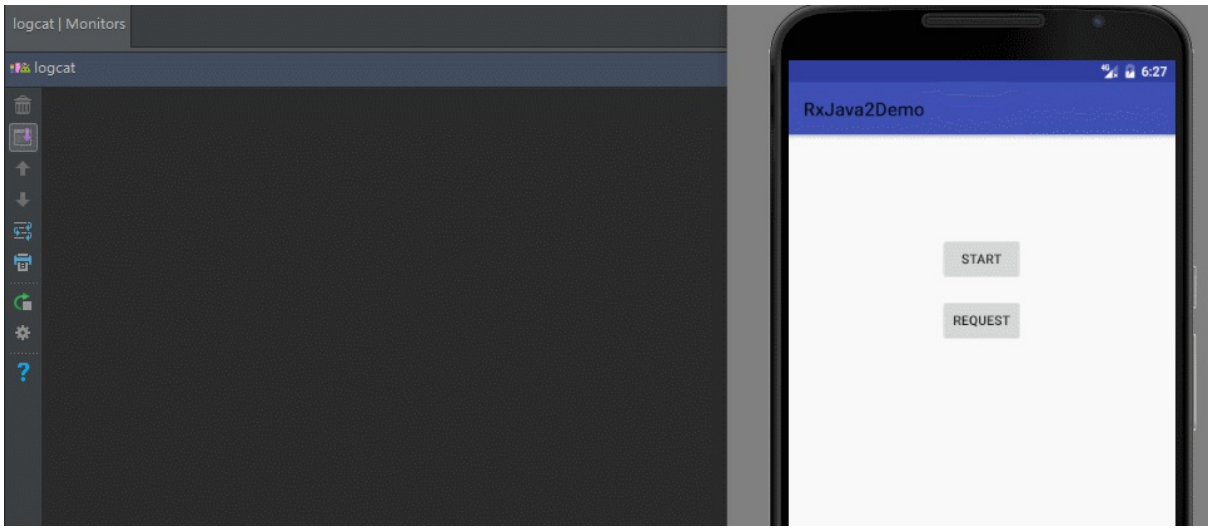
从运行结果中我们看到的确是如此, 第一次request的时候, 下游的确收到的是0-127这128个事件, 但第二次request的时候就不确定了, 因为上游一直在发送事件. 内存占用也很正常, drop的作用相信大家也很清楚了.

再来看看Latest吧:

```
public static void request() {
        mSubscription.request(128);
    }

public static void demo4() {
        Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                for (int i = 0; ; i++) {
                    emitter.onNext(i);
                }
            }
        }, BackpressureStrategy.LATEST).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
```

```
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

同样的, 上游无限循环发送事件, 策略选择Latest, 同时把Subscription保存起来, 方便在外部调用request(128).来看看这次的运行结果:



诶, 看上去好像和Drop差不多啊, Latest也首先保存了0-127这128个事件, 等下游把这128个事件处理了之后才进行之后的处理, 光从这里没有看出有任何区别啊...

> 古人云，师者，所以传道受业解惑也。人非生而知之者，孰能无惑？惑而不从师，其为惑也，终不解矣.

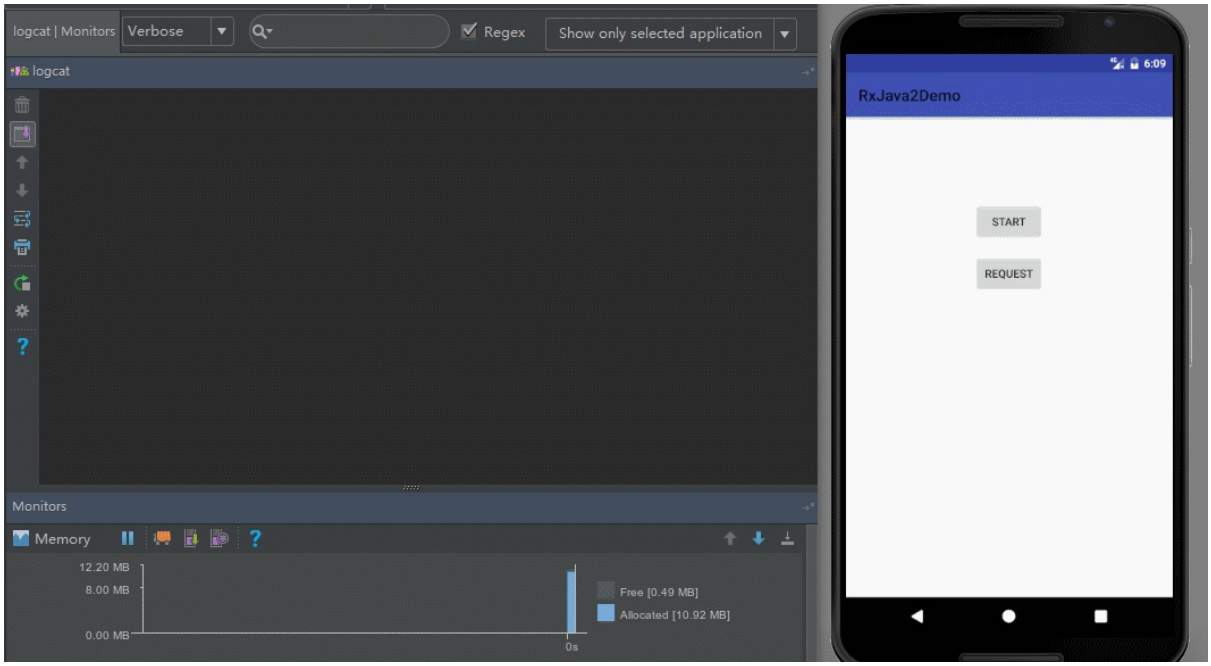
作为初学者的 `入门导师` , 是不能给大家留下一点点疑惑的, 来让我们继续揭开这个疑问.

我们把上面两段代码改良一下, 先来看看DROP的改良版:

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                for (int i = 0; i < 10000; i++) {  //只发1w个事件
                    emitter.onNext(i);
```

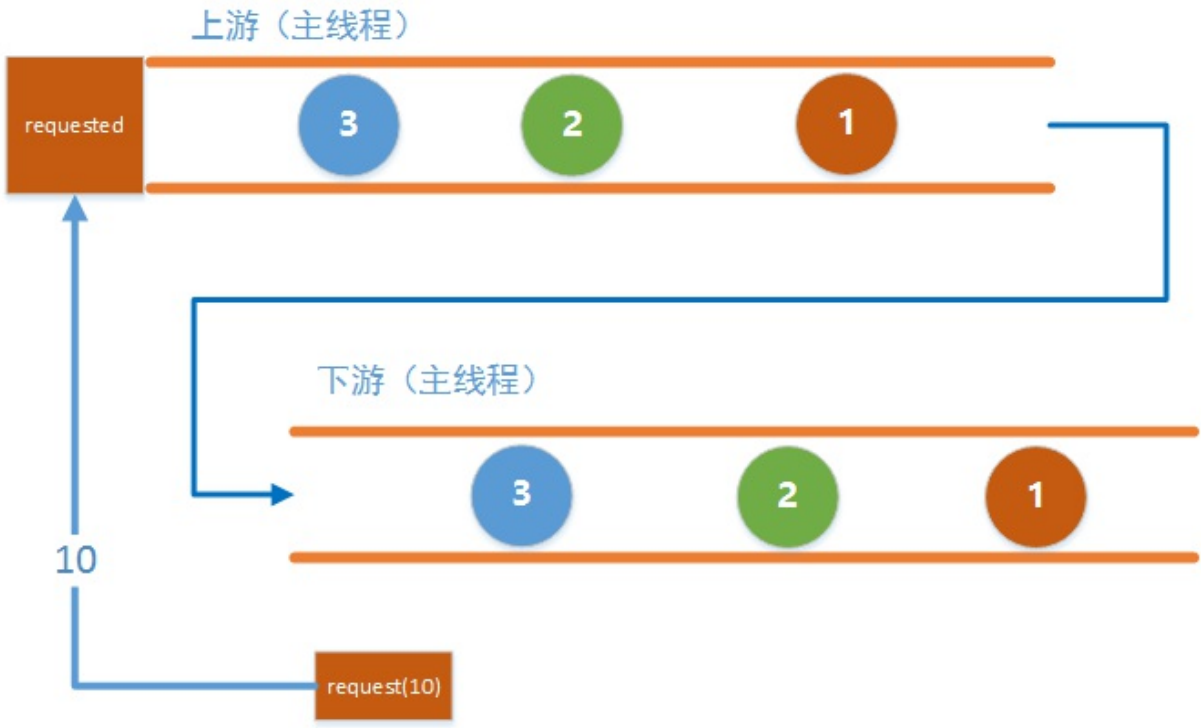
```
                }
            }
        }, BackpressureStrategy.DROP).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(128);  //一开始就处理掉128个事件
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
```

这段代码和之前有两点不同, 一是上游只发送了10000个事件, 二是下游在一开始就立马处理掉了128个事件, 然后我们在外部再调用request(128)试试, 来看看运行结果:



这次可以看到, 一开始下游就处理掉了128个事件, 当我们再次request的时候, 只得到了第3317的事件, 后面的事件直接被抛弃了.

再来看看Latest的运行结果吧:



从运行结果中可以看到, 除去前面128个事件, 与Drop不同, Latest总是能获取到最后最新的事件, 例如这里我们总是能获得最后一个事件9999.

好了, 关于FLowable的策略我们也讲完了, 有些朋友要问了, 这些FLowable是我自己创建的, 所以我可以选择策略, 那面对有些FLowable并不是我自己创建的, 该怎么办呢? 比如RxJava中的interval操作符, 这个操作符并不是我们自己创建的, 来看下面这个例子吧:

```
Flowable.interval(1, TimeUnit.MICROSECONDS)
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Long>() {
                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(Long.MAX_VALUE);
                    }

                    @Override
                    public void onNext(Long aLong) {
                        Log.d(TAG, "onNext: " + aLong);
                        try {
                            Thread.sleep(1000);  //延时1秒
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
```

interval操作符发送Long型的事件, 从0开始, 每隔指定的时间就把数字加1并发送出来, 在这个例子里, 我们让它每隔1毫秒就发送一次事件, 在下游延时1秒去接收处理, 不用猜也知道结果是什么:

```
zlc.season.rxjava2demo D/TAG: onSubscribe
zlc.season.rxjava2demo W/TAG: onError:
io.reactivex.exceptions.MissingBackpressureException: Can't deliver value 128 due to la
ck of requests
    at io.reactivex.internal.operators.flowable.FlowableInterval$IntervalSubscriber.run
(FlowableInterval.java:87)
    at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:428)
    at java.util.concurrent.FutureTask.runAndReset(FutureTask.java:278)
    at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(Schedul
edThreadPoolExecutor.java:273)
    at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1133)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:607)
    at java.lang.Thread.run(Thread.java:761)
```

一运行就抛出了`MissingBackpressureException`异常, 提醒我们发太多了, 那么怎么办呢, 这个又不是我们自己创建的FLowable啊...

别慌, 虽然不是我们自己创建的, 但是RxJava给我们提供了其他的方法:

- onBackpressureBuffer()
- onBackpressureDrop()
- onBackpressureLatest()

熟悉吗? 这跟我们上面学的策略是一样的, 用法也简单, 拿刚才的例子现学现用:

```
Flowable.interval(1, TimeUnit.MICROSECONDS)
                .onBackpressureDrop()  //加上背压策略
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Long>() {
                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(Long.MAX_VALUE);
                    }

                    @Override
                    public void onNext(Long aLong) {
                        Log.d(TAG, "onNext: " + aLong);
                        try {
                            Thread.sleep(1000);
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
```

其余的我就不一一列举了.

好了, 今天的教程就到这里吧, 这一节我们学习了如何使用内置的BackpressureStrategy来解决上下游事件速率不均衡的问题. 这些策略其实之前我们将Observable的时候也提到过, 其实大差不差, 只要理解了为什么会上游发事件太快, 下游处理太慢这一点, 你就好处理了, FLowable无非就是给你封装好了, 确实对初学者友好一点, 但是很多初学者往往只知道How, 却不知道Why, 最重要的其实是知道why, 而不是How.

(其余的教程大多数到这里就结束了, 但是, 你以为FLowable就这么点东西吗, 骚年, Too young too simple, sometimes naive! 这仅仅是开始, 真正牛逼的还没来呢. 敬请关注下一节, 下节见！)

前言

好久不见朋友们，最近一段时间在忙工作上的事情，没来得及写文章，这两天正好有点时间，赶紧写下了这篇教程，免得大家说我太监了。

正题

先来回顾一下上上节，我们讲Flowable的时候，说它采用了 `响应式拉` 的方式，我们还举了个 `叶问打小日本` 的例子，再来回顾一下吧，我们说把 `上游` 看成 `小日本`，把 `下游` 当作 `叶问`，当调用 `Subscription.request(1)` 时，`叶问` 就说 `我要打一个!` 然后 `小日本` 就拿出 `一个鬼子` 给叶问，让他打，等叶问打死这个鬼子之后，再次调用 `request(10)`，叶问就又说 `我要打十个!` 然后小日本又派出 `十个鬼子` 给叶问，然后就在边上看热闹，看叶问能不能打死十个鬼子，等叶问打死十个鬼子后再继续要鬼子接着打。

但是不知道大家有没有发现，在我们前两节中的例子中，我们口中声称的 `响应式拉` 并没有完全体现出来，比如这个例子：

```
Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> emitter) throws Exception {
                Log.d(TAG, "emit 1");
                emitter.onNext(1);
                Log.d(TAG, "emit 2");
                emitter.onNext(2);
                Log.d(TAG, "emit 3");
                emitter.onNext(3);
                Log.d(TAG, "emit complete");
                emitter.onComplete();
            }
        }, BackpressureStrategy.ERROR).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(1);
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                        mSubscription.request(1);
                    }
```

```
                @Override
                public void onError(Throwable t) {
                    Log.w(TAG, "onError: ", t);
                }

                @Override
                public void onComplete() {
                    Log.d(TAG, "onComplete");
                }
            });
```

虽然我们在下游中是每次处理掉了一个事件之后才调用request(1)去请求下一个事件，也就是说叶问的确是在打死了一个鬼子之后才继续打下一个鬼子，可是上游呢？上游真的是每次当下游请求一个才拿出一个吗？从上上篇文章中我们知道并不是这样的，上游仍然是一开始就发送了所有的事件，也就是说小日本并没有等叶问打死一个才拿出一个，而是一开始就拿出了所有的鬼子，这些鬼子从一开始就在这儿排队等着被打死。

有个故事是这么说的：

> 楚人有卖盾与矛者，先誉其盾之坚，曰：“吾盾之坚，物莫能陷也。”俄而又誉其矛之利，曰：“吾矛之利，万物莫不陷也。”市人诘之曰："以子之矛陷子之盾，何如？”其人弗能应也。众皆笑之。

没错，我们前后所说的就是自相矛盾了，这说明了什么呢，说明我们的实现并不是一个完整的实现，那么，究竟怎样的实现才是完整的呢？

我们先自己来想一想，在下游中调用Subscription.request(n)就可以告诉上游，下游能够处理多少个事件，那么上游要根据下游的处理能力正确的去发送事件，那么上游是不是应该知道下游的处理能力是多少啊，对吧，不然，一个巴掌拍不响啊，这种事情得你情我愿才行。

那么上游从哪里得知下游的处理能力呢？我们来看看上游最重要的部分，肯定就是`FlowableEmitter`了啊，我们就是通过它来发送事件的啊，来看看它的源码吧(别紧张，它的代码灰常简单)：

```
public interface FlowableEmitter<T> extends Emitter<T> {
    void setDisposable(Disposable s);
    void setCancellable(Cancellable c);

    /**
     * The current outstanding request amount.
     * <p>This method is thread-safe.
     * @return the current outstanding request amount
     */
    long requested();

    boolean isCancelled();
```

```
    FlowableEmitter<T> serialize();
}
```

FlowableEmitter是个接口，继承Emitter，Emitter里面就是我们的onNext(),onComplete()和onError()三个方法。我们看到FlowableEmitter中有这么一个方法：

```
long requested();
```

方法注释的意思就是`当前外部请求的数量`，哇哦，这好像就是我们要找的答案呢. 我们还是实际验证一下吧.

先来看`同步`的情况吧：

```
public static void demo1() {
       Flowable
               .create(new FlowableOnSubscribe<Integer>() {
                   @Override
                   public void subscribe(FlowableEmitter<Integer> emitter) throws Excep
tion {
                       Log.d(TAG, "current requested: " + emitter.requested());
                   }
               }, BackpressureStrategy.ERROR)
               .subscribe(new Subscriber<Integer>() {

                   @Override
                   public void onSubscribe(Subscription s) {
                       Log.d(TAG, "onSubscribe");
                       mSubscription = s;
                   }

                   @Override
                   public void onNext(Integer integer) {
                       Log.d(TAG, "onNext: " + integer);
                   }

                   @Override
                   public void onError(Throwable t) {
                       Log.w(TAG, "onError: ", t);
                   }

                   @Override
                   public void onComplete() {
                       Log.d(TAG, "onComplete");
                   }
               });
   }
```

这个例子中，我们在上游中打印出当前的request数量，下游什么也不做。

我们先猜测一下结果，下游没有调用request()，说明当前下游的处理能力为0，那么上游得到的requested也应该是0，是不是呢？

来看看运行结果：

```
D/TAG: onSubscribe
D/TAG: current requested: 0
```

哈哈，结果果然是0，说明我们的结论基本上是对的。

那下游要是调用了request()呢，来看看：

```
public static void demo1() {
       Flowable
               .create(new FlowableOnSubscribe<Integer>() {
                   @Override
                   public void subscribe(FlowableEmitter<Integer> emitter) throws Excep
tion {
                       Log.d(TAG, "current requested: " + emitter.requested());
                   }
               }, BackpressureStrategy.ERROR)
               .subscribe(new Subscriber<Integer>() {

                   @Override
                   public void onSubscribe(Subscription s) {
                       Log.d(TAG, "onSubscribe");
                       mSubscription = s;
                       s.request(10); //我要打十个！
                   }

                   @Override
                   public void onNext(Integer integer) {
                       Log.d(TAG, "onNext: " + integer);
                   }

                   @Override
                   public void onError(Throwable t) {
                       Log.w(TAG, "onError: ", t);
                   }

                   @Override
                   public void onComplete() {
                       Log.d(TAG, "onComplete");
                   }
               });
   }
```

这次在下游中调用了request(10)，告诉上游我要打十个，看看运行结果：

```
D/TAG: onSubscribe
D/TAG: current requested: 10
```

果然！上游的requested的确是根据下游的请求来决定的，那要是下游多次请求呢？比如这样：

```
public static void demo1() {
        Flowable
                .create(new FlowableOnSubscribe<Integer>() {
                    @Override
                    public void subscribe(FlowableEmitter<Integer> emitter) throws Exce
ption {
                        Log.d(TAG, "current requested: " + emitter.requested());
                    }
                }, BackpressureStrategy.ERROR)
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(10);  //我要打十个!
                        s.request(100); //再给我一百个!
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

下游先调用了request(10), 然后又调用了request(100)，来看看运行结果：

```
D/TAG: onSubscribe
```

```
D/TAG: current requested: 110
```

看来多次调用也没问题，做了 `加法` 。

诶加法？对哦，只是做加法，那什么时候做 `减法` 呢？

当然是发送事件啦！

来看个例子吧：

```
public static void demo2() {
        Flowable
                .create(new FlowableOnSubscribe<Integer>() {
                    @Override
                    public void subscribe(final FlowableEmitter<Integer> emitter) throws
 Exception {
                        Log.d(TAG, "before emit, requested = " + emitter.requested());

                        Log.d(TAG, "emit 1");
                        emitter.onNext(1);
                        Log.d(TAG, "after emit 1, requested = " + emitter.requested());

                        Log.d(TAG, "emit 2");
                        emitter.onNext(2);
                        Log.d(TAG, "after emit 2, requested = " + emitter.requested());

                        Log.d(TAG, "emit 3");
                        emitter.onNext(3);
                        Log.d(TAG, "after emit 3, requested = " + emitter.requested());

                        Log.d(TAG, "emit complete");
                        emitter.onComplete();

                        Log.d(TAG, "after emit complete, requested = " + emitter.request
ed());
                    }
                }, BackpressureStrategy.ERROR)
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(10);  //request 10
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
```

```
                }

                @Override
                public void onError(Throwable t) {
                    Log.w(TAG, "onError: ", t);
                }

                @Override
                public void onComplete() {
                    Log.d(TAG, "onComplete");
                }
            });
    }
```

代码很简单，来看看运行结果：

```
D/TAG: onSubscribe
D/TAG: before emit, requested = 10
D/TAG: emit 1
D/TAG: onNext: 1
D/TAG: after emit 1, requested = 9
D/TAG: emit 2
D/TAG: onNext: 2
D/TAG: after emit 2, requested = 8
D/TAG: emit 3
D/TAG: onNext: 3
D/TAG: after emit 3, requested = 7
D/TAG: emit complete
D/TAG: onComplete
D/TAG: after emit complete, requested = 7
```

大家应该能看出端倪了吧，下游调用request(n) 告诉上游它的处理能力，上游每发送一个 `next事件` 之后，requested就减一，`注意是next事件，complete和error事件不会消耗requested`，当减到0时，则代表下游没有处理能力了，这个时候你如果继续发送事件，会发生什么后果呢？当然是 `MissingBackpressureException` 啦，试一试：

```
public static void demo2() {
        Flowable
                .create(new FlowableOnSubscribe<Integer>() {
                    @Override
                    public void subscribe(final FlowableEmitter<Integer> emitter) throws
 Exception {
                        Log.d(TAG, "before emit, requested = " + emitter.requested());

                        Log.d(TAG, "emit 1");
                        emitter.onNext(1);
```

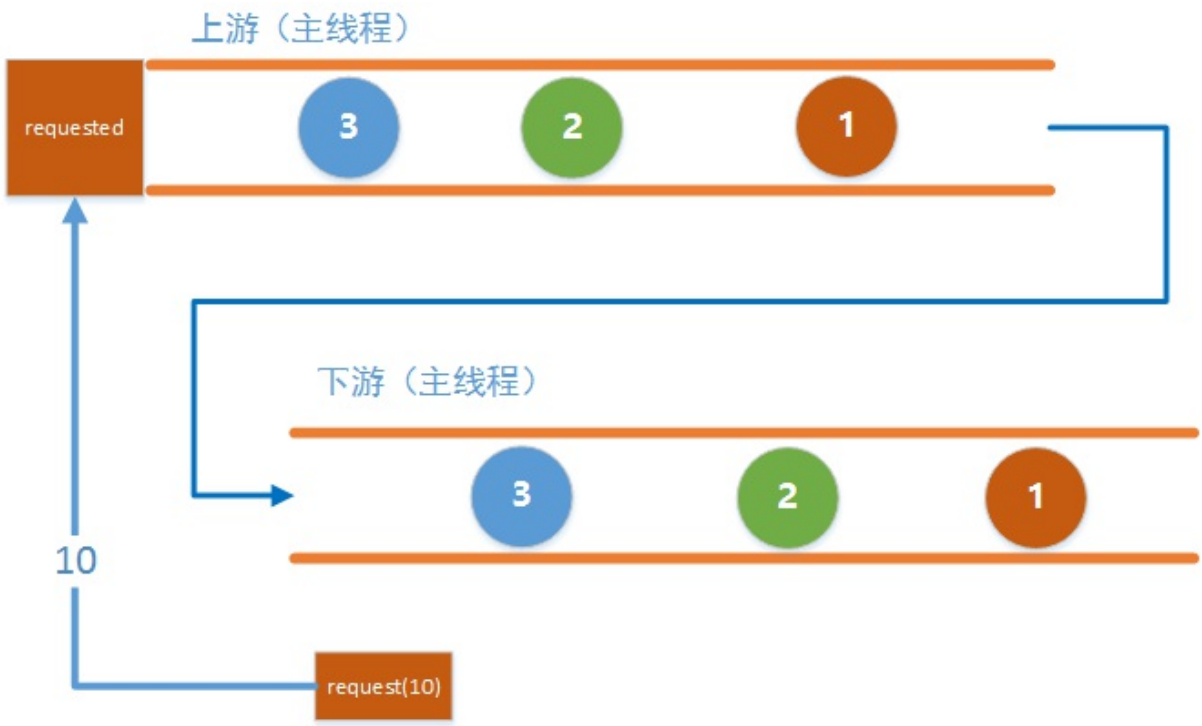
```
                        Log.d(TAG, "after emit 1, requested = " + emitter.requested());

                        Log.d(TAG, "emit 2");
                        emitter.onNext(2);
                        Log.d(TAG, "after emit 2, requested = " + emitter.requested());

                        Log.d(TAG, "emit 3");
                        emitter.onNext(3);
                        Log.d(TAG, "after emit 3, requested = " + emitter.requested());

                        Log.d(TAG, "emit complete");
                        emitter.onComplete();

                        Log.d(TAG, "after emit complete, requested = " + emitter.reques
ted());
                    }
                }, BackpressureStrategy.ERROR)
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(2);   //request 2
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

还是这个例子，只不过这次只request(2), 看看运行结果：

```
D/TAG: onSubscribe
D/TAG: before emit, requested = 2
```

```
 D/TAG: emit 1
 D/TAG: onNext: 1
 D/TAG: after emit 1, requested = 1
 D/TAG: emit 2
 D/TAG: onNext: 2
 D/TAG: after emit 2, requested = 0
 D/TAG: emit 3
 W/TAG: onError: io.reactivex.exceptions.MissingBackpressureException: create: could no
t emit value due to lack of requests
                at io.reactivex.internal.operators.flowable.FlowableCreate$ErrorAsyncE
mitter.onOverflow(FlowableCreate.java:411)
                at io.reactivex.internal.operators.flowable.FlowableCreate$NoOverflowB
aseAsyncEmitter.onNext(FlowableCreate.java:377)
                at zlc.season.rxjava2demo.demo.ChapterNine$4.subscribe(ChapterNine.jav
a:80)
                at io.reactivex.internal.operators.flowable.FlowableCreate.subscribeAc
tual(FlowableCreate.java:72)
                at io.reactivex.Flowable.subscribe(Flowable.java:12218)
                at zlc.season.rxjava2demo.demo.ChapterNine.demo2(ChapterNine.java:89)
                at zlc.season.rxjava2demo.MainActivity$2.onClick(MainActivity.java:36)
                at android.view.View.performClick(View.java:4780)
                at android.view.View$PerformClick.run(View.java:19866)
                at android.os.Handler.handleCallback(Handler.java:739)
                at android.os.Handler.dispatchMessage(Handler.java:95)
                at android.os.Looper.loop(Looper.java:135)
                at android.app.ActivityThread.main(ActivityThread.java:5254)
                at java.lang.reflect.Method.invoke(Native Method)
                at java.lang.reflect.Method.invoke(Method.java:372)
                at com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteIn
it.java:903)
                at com.android.internal.os.ZygoteInit.main(ZygoteInit.java:698)
 D/TAG: after emit 3, requested = 0
 D/TAG: emit complete
 D/TAG: after emit complete, requested = 0
```

到目前为止我们一直在说同步的订阅，现在同步说完了，我们先用一张图来总结一下同步的情况：



这张图的意思就是当上下游在同一个线程中的时候，在 `下游` 调用request(n)就会直接改变 `上游` 中的requested的值，多次调用便会叠加这个值，而上游每发送一个事件之后便会去减少这个值，当这个值减少至0的时候，继续发送事件便会抛异常了。

我们再来说说 `异步` 的情况，异步和同步会有区别吗？会有什么区别呢？带着这个疑问我们继续来探究。

同样的先来看一个基本的例子：

```
public static void demo3() {
        Flowable
                .create(new FlowableOnSubscribe<Integer>() {
                    @Override
                    public void subscribe(FlowableEmitter<Integer> emitter) throws Exce
ption {
                        Log.d(TAG, "current requested: " + emitter.requested());
                    }
                }, BackpressureStrategy.ERROR)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }
```

```
                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

这次是异步的情况，上游啥也不做，下游也啥也不做，来看看运行结果：

```
D/TAG: onSubscribe
D/TAG: current requested: 128
```

哈哈，又是128，看了我前几篇文章的朋友肯定很熟悉这个数字啊！这个数字为什么和我们之前所说的默认的水缸大小一样啊，莫非？

带着这个疑问我们继续来研究一下：

```
public static void demo3() {
        Flowable
                .create(new FlowableOnSubscribe<Integer>() {
                    @Override
                    public void subscribe(FlowableEmitter<Integer> emitter) throws Excep
tion {

                        Log.d(TAG, "current requested: " + emitter.requested());
                    }
                }, BackpressureStrategy.ERROR)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                        s.request(1000); //我要打1000个！！
                    }
```

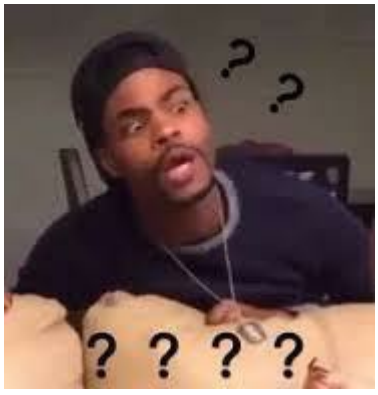
```
                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

这次我们在下游调用了request（1000）告诉上游我要打1000个，按照之前我们说的，这次的运行结果应该是1000，来看看运行结果：

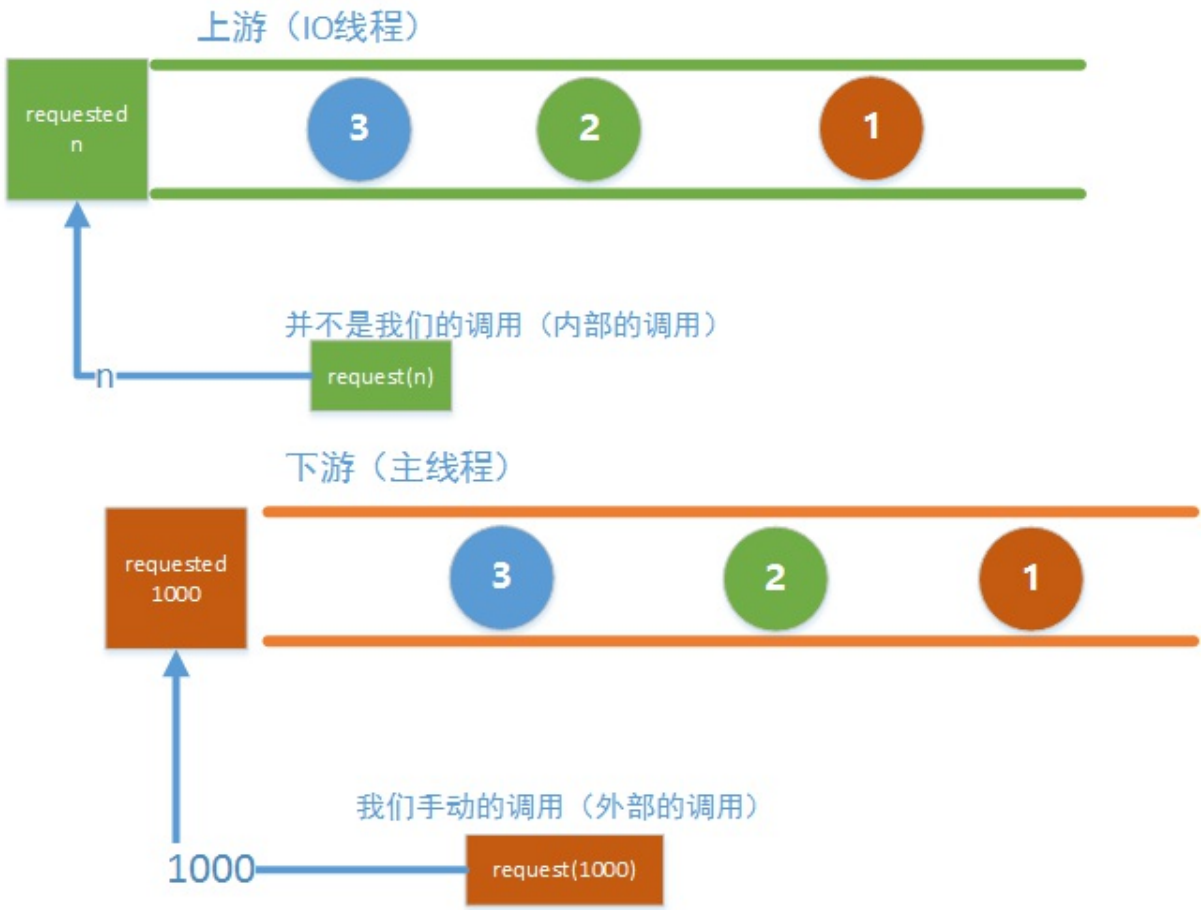
```
D/TAG: onSubscribe
D/TAG: current requested: 128
```

卧槽，你确定你没贴错代码？

是的，真相就是这样，就是128，蜜汁128。。。



为了答疑解惑，我就直接上图了：



可以看到，当上下游工作在不同的线程里时，每一个线程里都有一个requested，而我们调用request（1000）时，实际上改变的是下游主线程中的requested，而上游中的requested的值是由RxJava内部调用request(n)去设置的，这个调用会在合适的时候自动触发。

现在我们就能理解为什么没有调用request，上游中的值是128了，因为下游在`一开始就在内部调用了`request(128)去设置了上游中的值，因此即使下游没有调用request()，上游也能发送128个事件，这也可以解释之前我们为什么说Flowable中默认的水缸大小是128，其实就是这里设置的。

刚才同步的时候我们说了，上游每发送一个事件，requested的值便会减一，对于异步来说同样如此，那有人肯定有疑问了，一开始上游的requested的值是128，那这128个事件发送完了不就不能继续发送了吗？

刚刚说了，设置上游requested的值的这个内部调用会在`合适的时候`自动触发，那到底什么时候是合适的时候呢？是发完128个事件才去调用吗？还是发送了一半才去调用呢？

带着这个疑问我们来看下一段代码：

```
  public static void request() {
        mSubscription.request(96); //请求96个事件
    }

public static void demo4() {
```

```
        Flowable
                .create(new FlowableOnSubscribe<Integer>() {
                    @Override
                    public void subscribe(FlowableEmitter<Integer> emitter) throws Exce
ption {
                        Log.d(TAG, "First requested = " + emitter.requested());
                        boolean flag;
                        for (int i = 0; ; i++) {
                            flag = false;
                            while (emitter.requested() == 0) {
                                if (!flag) {
                                    Log.d(TAG, "Oh no! I can't emit value!");
                                    flag = true;
                                }
                            }
                            emitter.onNext(i);
                            Log.d(TAG, "emit " + i + " , requested = " + emitter.reques
ted());
                        }
                    }
                }, BackpressureStrategy.ERROR)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Subscriber<Integer>() {

                    @Override
                    public void onSubscribe(Subscription s) {
                        Log.d(TAG, "onSubscribe");
                        mSubscription = s;
                    }

                    @Override
                    public void onNext(Integer integer) {
                        Log.d(TAG, "onNext: " + integer);
                    }

                    @Override
                    public void onError(Throwable t) {
                        Log.w(TAG, "onError: ", t);
                    }

                    @Override
                    public void onComplete() {
                        Log.d(TAG, "onComplete");
                    }
                });
    }
```

这次的上游稍微复杂了一点点，首先仍然是个无限循环发事件，但是是有条件的，只有当上游的requested != 0的时候才会发事件，然后我们调用request（96）去消费96个事件（为什么是96而不是其他的数字先不要管），来看看运行结果吧：

```
D/TAG: onSubscribe
D/TAG: First requested = 128
D/TAG: emit 0 , requested = 127
D/TAG: emit 1 , requested = 126
D/TAG: emit 2 , requested = 125
  ...
D/TAG: emit 124 , requested = 3
D/TAG: emit 125 , requested = 2
D/TAG: emit 126 , requested = 1
D/TAG: emit 127 , requested = 0
D/TAG: Oh no! I can't emit value!
```

首先运行之后上游便会发送完128个事件，之后便不做任何事情，从打印的结果中我们也可以看出这一点。

然后我们调用request(96)，这会让下游去消费96个事件，来看看运行结果吧：

```
D/TAG: onNext: 0
D/TAG: onNext: 1
  ...
D/TAG: onNext: 92
D/TAG: onNext: 93
D/TAG: onNext: 94
D/TAG: onNext: 95
D/TAG: emit 128 , requested = 95
D/TAG: emit 129 , requested = 94
D/TAG: emit 130 , requested = 93
D/TAG: emit 131 , requested = 92
  ...
D/TAG: emit 219 , requested = 4
D/TAG: emit 220 , requested = 3
D/TAG: emit 221 , requested = 2
D/TAG: emit 222 , requested = 1
D/TAG: emit 223 , requested = 0
D/TAG: Oh no! I can't emit value!
```

可以看到，当下游消费掉第96个事件之后，上游又开始发事件了，而且可以看到当前上游的requested的值是96(打印出来的95是已经发送了一个事件减一之后的值)，最终发出了第223个事件之后又进入了等待区，而223-127 正好等于 96。

这是不是说明当下游每消费96个事件便会自动触发内部的request()去设置上游的requested的值啊！没错，就是这样，而这个新的值就是96。

朋友们可以手动试试请求95个事件，上游是不会继续发送事件的。

至于这个96是怎么得出来的（肯定不是我猜的蒙的啊），感兴趣的朋友可以自行阅读源码寻找答案，对于初学者而言应该没什么必要，管它内部怎么实现的呢对吧。

好了今天的教程就到这里了！通过本节的学习，大家应该知道如何正确的去实现一个完整的响应式拉取了，在`某一些场景`下，可以在发送事件前先判断当前的requested的值是否大于0，若等于0则说明下游处理不过来了，则需要等待，例如下面这个例子。

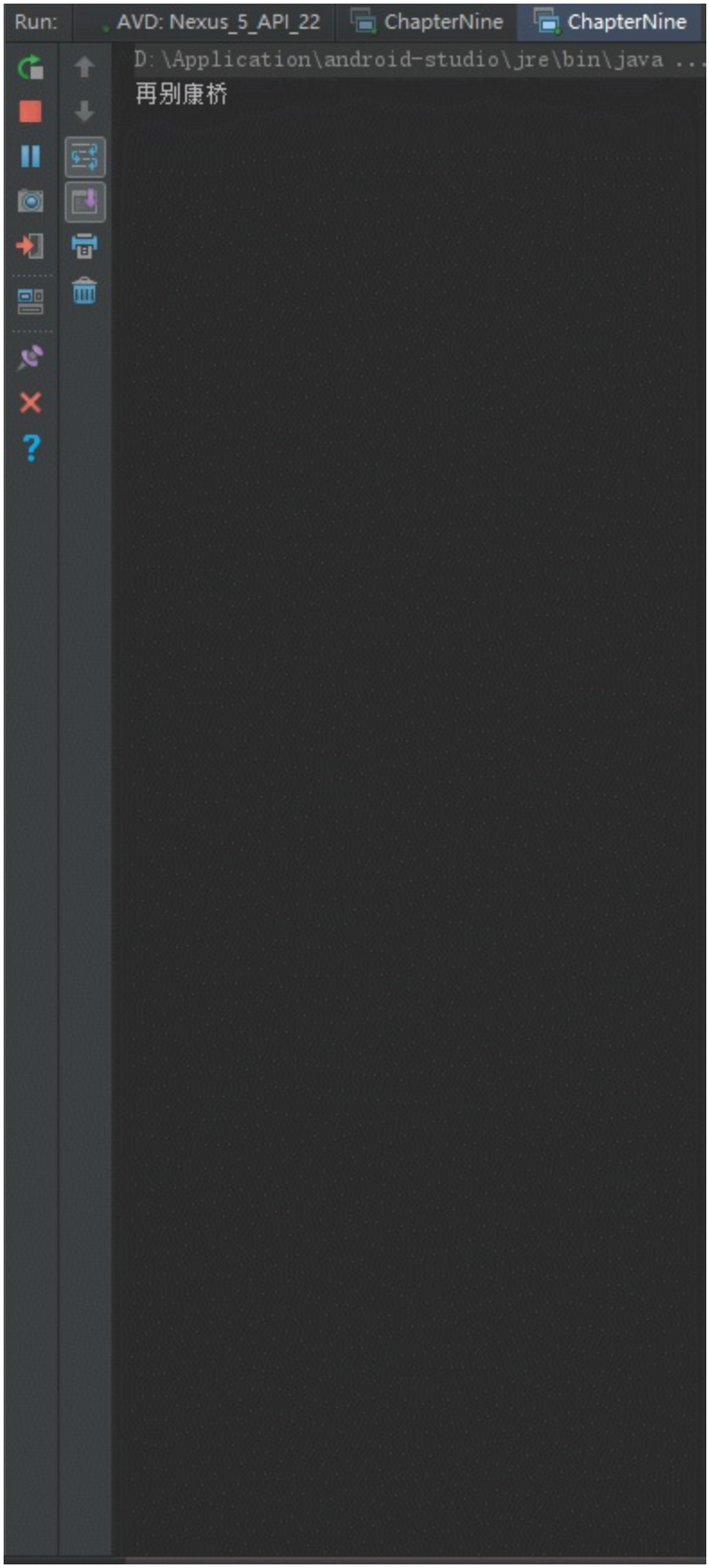
实践

这个例子是读取一个文本文件，需要一行一行读取，然后处理并输出，如果文本文件很大的时候，比如几十M的时候，全部先读入内存肯定不是明智的做法，因此我们可以一边读取一边处理，实现的代码如下：

```
public static void main(String[] args) {
        practice1();
        try {
            Thread.sleep(10000000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public static void practice1() {
        Flowable
                .create(new FlowableOnSubscribe<String>() {
                    @Override
                    public void subscribe(FlowableEmitter<String> emitter) throws Except
ion {
                        try {
                            FileReader reader = new FileReader("test.txt");
                            BufferedReader br = new BufferedReader(reader);

                            String str;

                            while ((str = br.readLine()) != null && !emitter.isCancelled
()) {
                                while (emitter.requested() == 0) {
                                    if (emitter.isCancelled()) {
                                        break;
                                    }
                                }
                                emitter.onNext(str);
                            }

                            br.close();
```

```
                        reader.close();

                        emitter.onComplete();
                    } catch (Exception e) {
                        emitter.onError(e);
                    }
                }
            }, BackpressureStrategy.ERROR)
            .subscribeOn(Schedulers.io())
            .observeOn(Schedulers.newThread())
            .subscribe(new Subscriber<String>() {

                @Override
                public void onSubscribe(Subscription s) {
                    mSubscription = s;
                    s.request(1);
                }

                @Override
                public void onNext(String string) {
                    System.out.println(string);
                    try {
                        Thread.sleep(2000);
                        mSubscription.request(1);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }

                @Override
                public void onError(Throwable t) {
                    System.out.println(t);
                }

                @Override
                public void onComplete() {
                }
            });
    }
```

运行的结果便是：



好了，本次的教程就到这里了，谢谢大家捧场！下节见，敬请期待！（PS： 我这么用心的写文章， 你们也不给个赞吗？）