# [在安卓中的应用](http://blog.csdn.net/qq_35064774/article/details/53065400)

1. 为什么写这篇文章

RxJava这些年越来越流行，而上月末(2016.10.29)发布了2.0正式版，但网上大部分关于RxJava的教程都是1.x的。关于2.0的教程基本是介绍1.x和2.x的区别，对于RxJava的老用户来说，自然看看和1.x的区别就大致会用了，但是对于新手来说，就不得不先学1.x。这样来说，学习成本就提高了，本身RxJava就不容易上手。

为了让年轻的司机可以直接从2.0开始学习，我就写了这篇文章。RxJava的老用户可以直接看我这篇文章 [RxJava 2.0有什么不同(译)](http://blog.csdn.net/qq_35064774/article/details/53045298)。 

由于本人文笔拙略，于是仿照着 [Grokking RxJava](http://blog.danlew.net/2014/09/15/grokking-rxjava-part-1/) 来写，望 Dan Lew 大大不要介意。

2. 基础

RxJava 2.0 最核心的是Publisher和Subscriber。Publisher可以发出一系列的事件，而Subscriber负责和处理这些事件。

平常用得最多的Publisher是Flowable，它支持背压，教程刚开始不适合介绍太多概念，有兴趣的可以看一下 [RxJava 2.0中backpressure(背压)概念的理解](http://blog.csdn.net/jdsjlzx/article/details/52717636)。

要使用RxJava 2，你需要先引入相应的jar包。

compile 'io.reactivex.rxjava2:rxjava:2.0.0'

compile 'org.reactivestreams:reactive-streams:1.0.0'

注意，和1.x中不一样，2.0有一个依赖包。

3. Hello RxJava 2

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

// create a flowable

Flowable<String> flowable = Flowable.create(new FlowableOnSubscribe<String>() {

@Override

public void subscribe(FlowableEmitter<String> e) throws Exception {

e.onNext("hello RxJava 2");

e.onComplete();

}

}, BackpressureStrategy.BUFFER);

上述代码仅仅是发射了一个字符串"hello RxJava 2"。   
下面我们还需要创建一个Subscriber。

// create

Subscriber subscriber = new Subscriber<String>() {

@Override

public void onSubscribe(Subscription s) {

System.out.println("onSubscribe");

s.request(Long.MAX\_VALUE);

}

@Override

public void onNext(String s) {

System.out.println(s);

}

@Override

public void onError(Throwable t) {

}

@Override

public void onComplete() {

System.out.println("onComplete");

}

};

需要注意的是，在onSubscribe中，我们需要调用request去请求资源，参数就是要请求的数量，一般如果不限制请求数量，可以写成Long.MAX\_VALUE。如果你不调用request，Subscriber的onNext和onComplete方法将不会被调用。

onNext方法里面传入的参数就是Flowable中发射出来的。

为了让”发射器”和”接收器”工作起来，我们还需要把他们组装在一起。

flowable.subscribe(subscriber);

一旦 flowable.subscribe 被执行，就会分别打印 hello RxJava 2 和 onComplete。

4. 更简洁的代码

上面一大串代码仅仅就达到了打印两个字符串的效果，你可能会想:”RxJava只不过是把事情变复杂了”。   
或许是这样的，但RxJava也提供了很多便利的方法来做这种简单的事情。

Flowable<String> flowable = Flowable.just("hello RxJava 2")；

我们可以直接调用Flowable.just创建一个发射字符串的”发射器”。

而对于 Subscriber 来说，我们目前仅仅关心onNext方法。所以可以简写成下面这样。

Consumer consumer = new Consumer<String>() {

@Override

public void accept(String s) throws Exception {

System.out.println(s);

}

};

当然这只是一个 Consumer，但 subscribe 方法提供了重载，让我们可以只传入一个Consumer。

所以订阅代码是这样的。

flowable.subscribe(consumer);

* 1
* 1

如果省去单独定义变量，最终可以写成下面这样。

Flowable.just("hello RxJava 2")

.subscribe(new Consumer<String>() {

@Override

public void accept(String s) throws Exception {

System.out.println(s);

}

});

5. 变换

让我们做一些更有意思的事情把！   
比如我想在hello RxJava 2后面加上我的签名，你可能会去修改Flowable传入的参数:

Flowable.just("hello RxJava 2 -ittianyu")

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

这当然是可以的，但是这样做，就导致所有的接收者都会受到影响。我只想针对某个订阅者做修改，那么你可能会写出这样的代码:

Flowable.just("hello RxJava 2")

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

这样的方式仍然不让人满意，因为我希望订阅者做的事越少越好，因为一般来说，订阅者都是在主线程中执行的。这个时候我们就可以利用操作符在数据传递的途中进行变换。

6. 操作符

操作符是为了解决 Flowable 对象变换问题而设计的，操作符可以在传递的途中对数据进行修改。   
RxJava提供了很多实用的操作符。比如 map 操作符，可以把一个事件转换成另一个事件。

Flowable.just("map")

.map(new Function<String, String>() {

@Override

public String apply(String s) throws Exception {

return s + " -ittianyu";

}

})

.subscribe(new Consumer<String>() {

@Override

public void accept(String s) throws Exception {

System.out.println(s);

}

});

上面代码中， map 是把传递过来的结果末尾加上了签名，然后在传递给了订阅者。   
是不是觉得神奇？   
map的作用就变换 Flowable 然后返回一个指定类型的 Flowable 对象。

7. map 操作符进阶

map 操作符更神奇的地方是，你可以返回任意类型的 Flowable，也就是说你可以使用 map 操作符发射一个新的数据类型的 Flowable 对象。

比如上面的例子，订阅者想要得到字符串的hashcode。

Flowable.just("map1")

.map(new Function<String, Integer>() {

@Override

public Integer apply(String s) throws Exception {

return s.hashCode();

}

})

.map(new Function<Integer, String>() {

@Override

public String apply(Integer integer) throws Exception {

return integer.toString();

}

})

.subscribe(new Consumer<String>() {

@Override

public void accept(String s) throws Exception {

System.out.println(s);

}

});

这里用了两个map，一个是把字符串转成hashcode，另一个是把hashcode 转成字符串。

8. 总结

1. 你可以在 Publisher 中查询数据库或者从网络上获取数据，然后在 Subscriber 中显示。
2. Publisher 不只有一种，事实上 Flowable 和 Processor 所有的子类都属于 Publisher。
3. 在数据发射途中，你可以利用操作符对数据进行变换。

# 操作符

# 1. 前言

在上一篇中，我介绍了RxJava 2.0的一些基础知识，同时也介绍了map()操作符。这篇blog将介绍许多RxJava中的操作符，RxJava的强大性就来自于它所定义的操作符。   
首先先看一个例子： 

# 2. 准备工作

假设我的 Flowable 发射的是一个列表，接收者要把列表内容依次输出。根据上一篇blog的内容，你可以会写出这样的代码:

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.just(list)

.subscribe(new Consumer<List<Integer>>() {

@Override

public void accept(List<Integer> list) throws Exception {

for (Integer integer : list)

System.out.println(integer);

}

});

这样的代码当然是不能容忍的，因为上面的代码使我们丧失了变化数据流的能力。一旦我们想要更改列表中的每一个数据，只能在订阅者中做。

当然我们可以使用map来中间处理，但是这样做也需要遍历整个list。

万幸，RxJava 2.0 提供了fromIterable方法，可以接收一个 Iterable 容器作为输入，每次发射一个元素。

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.fromIterable(list)

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

我们把fromX用到这个例子中来。

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.just(list)

.subscribe(nums -> {

Observable.fromIterable(nums)

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

});

虽然去掉了 for 循环，但是代码依然看起来很乱。嵌套了两层，它会破坏某些我们现在还没有讲到的RxJava的特性。

# 3. 改进

救星来了,他就是 flatMap()。   
Flowable.flatMap 可以把一个 Flowable 转换成另一个 Flowable :

List<Integer> list = new ArrayList<>();

list.add(10);

list.add(1);

list.add(5);

Flowable.just(list)

.flatMap(new Function<List<Integer>, Publisher<Integer>>() {

@Override

public Publisher<Integer> apply(List<Integer> integers) throws Exception {

return Flowable.fromIterable(integers);

}

})

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

System.out.println(integer);

}

});

和 map 不同之处在于 flatMap 返回的是一个 Flowable 对象。这正是我们想要的，我们可以把从List发射出来的一个一个的元素发射出去。

# 4. 更多操作符

目前为止，我们已经接触了两个操作符，RxJava中还有更多的操作符。

1. 如果我们想要订阅者只能收到大于5的数据，那么你可以这样做:
2. Flowable.fromArray(1, 20, 5, 0, -1, 8)
3. .filter(new Predicate<Integer>() {
4. @Override
5. public boolean test(Integer integer) throws Exception {
6. return integer.intValue() > 5;
7. }
8. })
9. .subscribe(new Consumer<Integer>() {
10. @Override
11. public void accept(Integer integer) throws Exception {
12. System.out.println(integer);
13. }

});

filter 是用于过滤数据的，返回false表示拦截此数据。

1. 如果我们只想要2个数据:
2. Flowable.fromArray(1, 2, 3, 4)
3. .take(2)
4. .subscribe(new Consumer<Integer>() {
5. @Override
6. public void accept(Integer integer) throws Exception {
7. System.out.println(integer);
8. }

});

take 用于指定订阅者最多收到多少数据。

1. 如果我们想在订阅者接收到数据前干点事情，比如记录日志:
2. Flowable.just(1, 2, 3)
3. .doOnNext(new Consumer<Integer>() {
4. @Override
5. public void accept(Integer integer) throws Exception {
6. System.out.println("保存:" + integer);
7. }
8. })
9. .subscribe(new Consumer<Integer>() {
10. @Override
11. public void accept(Integer integer) throws Exception {
12. System.out.println(integer);
13. }

});

doOnNext 允许我们在每次输出一个元素之前做一些额外的事情。

# 5. 总结

# 如果你是从第一篇一直跟着敲代码，坚持到敲完了这一篇。  我相信你应该开始对RxJava 2.0 有感觉了。  别紧张，教程还未终结。

# 响应式的好处

# 1. 前言

在第一篇中，我介绍了RxJava的基础知识。第二篇中，我向你展示了操作符的强大之处。这一篇，我将向你介绍响应式的优点。 

# 2. 错误处理

到目前为止，我都没怎么介绍 onComplete 和 onError 方法。这两个方法用来通知订阅者，数据发送完成或出现错误。

Flowable.create(new FlowableOnSubscribe<String>() {

@Override

public void subscribe(FlowableEmitter<String> e) throws Exception {

e.onNext("exception:" + (1 / 0));

e.onComplete();

}

}, BackpressureStrategy.BUFFER)

.subscribe(new Subscriber<String>() {

@Override

public void onSubscribe(Subscription s) {

s.request(1);

}

@Override

public void onNext(String s) {

System.out.println(s);

}

@Override

public void onError(Throwable t) {

t.printStackTrace();

System.out.println("onError");

}

@Override

public void onComplete() {

System.out.println("on complete");

}

});

上面的代码中，发射数据时，做了一个(1 / 0)的运算，但这明显是会抛出除零异常的。所以，上述代码最后会打印 onError。   
而如果改成(1 / 1)，则打印的是 exception:1 和 on complete。

这样的设计有以下几个优点:

1. 只要发生错误，onError()一定会被调用。   
   这极大的简化了错误处理。只需要在一个地方处理错误即可以。
2. 操作符不需要处理异常。   
   将异常处理交给订阅者来做，一旦有调用链中有一个抛出了异常，就会直接执行onError()方法，停止数据传送。
3. 你能够知道什么时候订阅者已经接收了全部的数据。

# 3. 调度器

假设你编写的 [**Android**](http://lib.csdn.net/base/android) App 需要从网络请求数据。网络请求是耗时的操作，因此你不得不在子线程中加载数据。那么问题来了！

在Android中写多线程不是一件容易的事，尤其是嵌套数据获取，比如要获取用的资料，其中有一项是头像，但得到的一般是头像的url地址，你还需要在资料获取成功后，在发送一次请求，这样就导致代码看起来很乱。

幸运的是我们有银弹。

使用RxJava你可以随意的切换线程。

Flowable.create(new FlowableOnSubscribe<String>() {

@Override

public void subscribe(FlowableEmitter<String> e) throws Exception {

e.onNext("将会在3秒后显示");

SystemClock.sleep(3000);

e.onNext("ittianyu");

e.onComplete();

}

}, BackpressureStrategy.BUFFER)

.subscribeOn(Schedulers.io())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Consumer<String>() {

@Override

public void accept(String s) throws Exception {

Toast.makeText(RxJava2Activity.this, s, Toast.LENGTH\_SHORT).show();

}

});

上述代码中，Flowable总共发射了两个数据，但中间延时了3秒，如果在主线程中延时，那将会导致UI卡顿，这是绝对不能容忍的。   
所以在订阅之前，我们使用 subscribeOn(Schedulers.io()) 指定了发送数据是在io线程(某个子线程)，然后调用 observeOn(AndroidSchedulers.mainThread()) 指定订阅者在主线程执行。

对了，要使用 AndroidSchedulers 还需要引入 RxAndroid:

compile 'io.reactivex.rxjava2:rxandroid:2.0.0'

# 4. 总结

这一篇向你介绍了一些关于响应式的优点，当然，这只是RxJava的冰山一角。你可以继续阅读下一篇教程，学习在**[安卓](http://lib.csdn.net/base/android" \o "Android知识库" \t "_blank)**中使用RxJava。

# [在安卓中的应用](http://blog.csdn.net/qq_35064774/article/details/53065400)

# 1. 前言

在第1 2 3篇中，我介绍了RxJava 2.0的基本用法。这一篇将介绍在**[安卓](http://lib.csdn.net/base/android" \o "Android知识库" \t "_blank)**中的基本用法。   
附上本系列教程的链接。   
[从零开始的RxJava2.0教程(一)基础](http://blog.csdn.net/qq_35064774/article/details/53057332)   
[从零开始的RxJava2.0教程(二)操作符](http://blog.csdn.net/qq_35064774/article/details/53057359)   
[从零开始的RxJava2.0教程(三)响应式的好处](http://blog.csdn.net/qq_35064774/article/details/53065360)   
[从零开始的RxJava2.0教程(四)在安卓中的应用](http://blog.csdn.net/qq_35064774/article/details/53065400)

# 2. RxAndroid

RxAndroid是RxJava针对**[Android](http://lib.csdn.net/base/android" \o "Android知识库" \t "_blank)**平台的扩展库。

曾几何时，RxAndroid确实提供了很多的实用的方法，但后来社区上很多人对这库的结构有意见，然后作者便进行了重构，现在只保留了AndroidSchedulers， 现在基本RxAndroid只有一个功能，那就是AndroidSchedulers.mainThread 方法用于指定主线程。

以前那些类被划分到了其他一些Rx库，比如 RxBinding, RxLifecycle。这里我并不打算介绍，因为对于新手来说，为时太早，如果你有兴趣，可以自己查阅相关资料。

# 3. 配合 Retrofit 2

目前来说，Android的网络库基本被 Retrofit + OkHttp 一统天下了。因为它确实很优秀。

当然，RxJava的影响力也不容小觑，Retrofit 也提供了对RxJava 的支持，截止到目前(2016.11.06)为止，官方似乎还没有支持RxJava 2.0。不过你不用担心，已经有大神帮我们做好了。

要使用 Retrofit 需要引入对应的库。

compile 'com.squareup.okhttp3:okhttp:3.4.1'

compile 'com.squareup.okio:okio:1.10.0'

compile 'com.squareup.retrofit2:retrofit:2.1.0'

compile 'com.jakewharton.retrofit:retrofit2-rxjava2-adapter:1.0.0'

retrofit2 依赖于 okhttp3，而 okhttp3 又依赖于okio。最后一项便是大神写好的 retrofit2-rxjava2-adapter 适配器啦。

假设我们现在需要去获取百度页面的html代码。

public interface BaiDuService {

@GET("/")

Flowable<ResponseBody> getText();

}

定义好请求接口。注意，返回值是 Flowable 类型。

Retrofit retrofit = new Retrofit.Builder()

.baseUrl("http://www.baidu.com/")

.addCallAdapterFactory(RxJava2CallAdapterFactory.create())// 添加RxJava2的适配器支持

.build();

BaiDuService service = retrofit.create(BaiDuService.class);

service.getText()

.subscribeOn(Schedulers.newThread())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<ResponseBody>() {

@Override

public void onSubscribe(Subscription s) {

s.request(Long.MAX\_VALUE);

}

@Override

public void onNext(ResponseBody s) {

Toast.makeText(RxJava2Activity.this, "获取成功", Toast.LENGTH\_SHORT).show();

try {

System.out.println(s.string());

} catch (IOException e) {

e.printStackTrace();

}

}

@Override

public void onError(Throwable t) {

t.printStackTrace();

Toast.makeText(RxJava2Activity.this, "获取失败，请检查网络是否畅通", Toast.LENGTH\_SHORT).show();

}

@Override

public void onComplete() {

System.out.println("任务结束");

}

});

在创建 Retrofit 的时候添加对 RxJava 2 的适配器，这样，请求就可以直接返回 Flowable。 然后就可以进行 RxJava 2 的操作了。

# 4. 从RxJava 2中获取数据

前面我介绍了可以通过 fromX 的方法把数组、列表等数据发射出去。那么有没有办法直接把发射的数据获取出来而不是通过订阅者来输出呢？

List<String> ids = Flowable.range(1, 100)

.map(new Function<Integer, String>() {

@Override

public String apply(Integer integer) throws Exception {

return "id:" + integer;

}

})

.toList().blockingGet();

System.out.println(ids);

比如这里，我发射了1到100总计100个数据，我们可以通过 blockingX 方法来拿到这些数据。

# 5. 结束

本系列教程到此结束，但我介绍的仅仅只是 RxJava 的冰山一角，如果你有更高的需求，请查阅其他资料。

感谢您的阅读。