

RxJava/RxAndroid 使用实例实践



作者 sheepm (/u/cd2a290d5feb) [+ 关注](#)

2017.02.16 22:36 字数 3693 阅读 1160 评论 4 喜欢 25

(/u/cd2a290d5feb)

原文地址

RxAndroid Tutorial (<https://www.raywenderlich.com/141980/rxandroid-tutorial>)

响应式编程（Reactive programming）不是一种API，而是一种新的非常有用的范式，而RxJava (<https://github.com/ReactiveX/RxJava>)就是一套基于此思想的框架，在Android开发中我们通过这个框架就能探索响应式的世界，同时结合另一个库，RxAndroid (<https://github.com/ReactiveX/RxAndroid>)，这是一个扩展库，更好的兼容了Android特性，比如主线程，UI事件等。

在这篇指南中，你将会学习到以下这些内容：

- 什么是响应式编程
- 什么是observable
- 如何将异步事件比如按钮点击或者EditText字符变化转换成observables
- observable变换
- observable 过滤拦截
- 如何指定链式中的代码执行线程
- 如何合并多个observables

前言

从 the starter project for this tutorial (<https://koenig-media.raywenderlich.com/uploads/2016/11/CheeseFinder-starter-2.zip>) 可以下载这篇文章中项目的所有代码，可以直接在Android Studio中打开。

大部分的代码都在 `CheeseActivity.java` 这个类里面，继承于 `BaseSearchActivity`；里面有一些基础方法：

`showProgressBar()`: 显示一个进度条

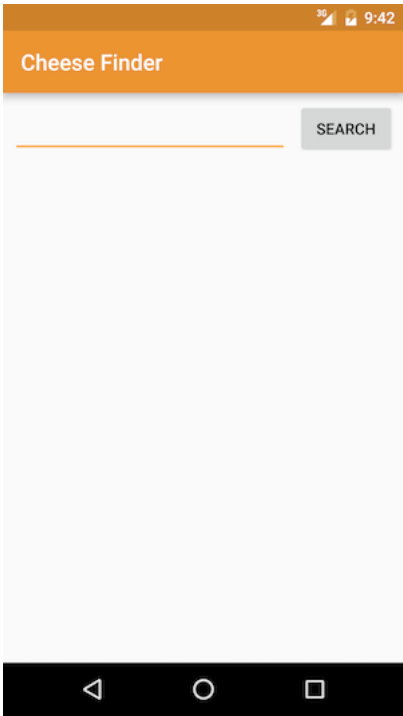
`hideProgressBar()`: 隐藏一个进度条

`showResult(List<String> result)`: 显示一个列表数据

`mCheeseSearchEngine`: `CheeseSearchEngine`类的一个对象，内部有一个search方法，接收一个数据查询并返回一个匹配的列表list。

直接运行的话，跑出来是这样子，就是一个查询的界面：





什么是响应式编程

在创建第一个observable之前，先看一下响应式编程的理论 :]

一般的程序是这样的，表达式只会计算一次，然后把赋值给变量

```
int a = 2;
int b = 3;
int c = a * b; // c is 6

a = 10;
// c is still 6
```

在a重新赋值后，前面的c并不会变化，而响应式编程会对值的变化做出响应。
有时候很有可能你已经做过一些响应式编程，但是并没有意识到这一点。
比如Excel中的表格，我们可以在表格里填上一些值，同时将某个格子的值设为一个表达式，就像下面这样

	<i>fx</i>	=B1*B2	
		A	B
1	a:		2
2	b:		3
3	c:		6
4			

设置这个表格里 B1区域的值为2，B2区域的值为3，B3是一个表达式，B3 = B1* B2，当其中一个值改变的时候，这个观察者B3也会变化，如图把B1改成10，B3就会自动计算成30。

	<i>fx</i>	10	
		A	B
1	a:		10
2	b:		3
3	c:		30
4			

RxJava Observable

RxJava使用的是**观察者模式**，其中有两个关键的接口：Observable 和 Observer，当Observable（被观察的对象）状态改变，所有subscribed（订阅）的Observer（观察者）会收到一个通知。



在Observable的接口中有一个方法 `subscribe()`，这样Observer 可以调用来进行订阅。同样，在Observer 接口中有三个方法，会被Observable 回调：

- `onNext(T value)` 提供了一个 T 类型的item给Observer
- `onComplete()` 在Observable发送items结束后通知Observer
- `onError(Throwable e)` 当Observable发生错误时通知Observer

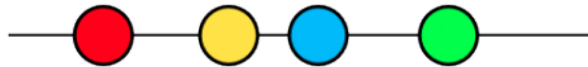
作为一个表现良好的Observable，发射0到多个数据时后面都会跟上一个completion 或是error的回调。

听起来有点复杂，但是一些例子可以很清晰的解释。

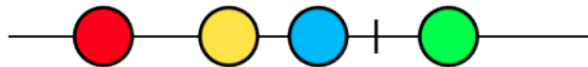
一个网络请求observable 通常只发射一个数据并且立刻completes。



每一个圆代表了从observable 发射出去的item数据，黑色的block代表了结束或是错误。一个鼠标的移动observable 将会不断的发送鼠标当前坐标，并且从不会结束。



在一个observable 已经结束后不能再发射新的item数据，下面这个就是一个不好的示范，违反了Observable 的准则



在已经发信号结束后依然发射了一个item。

怎么创建一个Observable

你可以直接通过 `observable.create()` 创建一个Observable

```
Observable<T> create(ObservableOnSubscribe<T> source)
```

看起来十分的简洁，但是这段代码是什么意思呢？这个“source” 又是什么？ 想要理解这个，只需要知道 `ObservableOnSubscribe` 是什么。这是一个接口，其中有一个方法：



```
public interface ObservableOnSubscribe<T> {  
    void subscribe(ObservableEmitter<T> emitter) throws Exception;  
}
```

这个你创建Observable 时的一个“source” 需要暴露一个 subscribe() 方法，从这里又引出来另一个 emitter（发射器），那么什么又是emitter？

RxJava中的 Emitter 接口和 Observer 比较相似，都有以下方法

```
public interface Emitter<T> {  
    void onNext(T value);  
    void onError(Throwable error);  
    void onComplete();  
}
```

ObservableEmitter 提供了一个方法用来取消订阅，用一个实际场景来形容一下。想象一个水龙头和水流，这个管道就相当于Observable，从里面能放出水，ObservableEmitter 就相当于水龙头，控制开关，而水龙头连接到管道就是 Observable.create()。

举个例子免得前面描述太过于抽象，先来看看第一个例子

观察按钮点击事件

在 CheeseActivity 类中有这么一段代码

```
// 1  
private Observable<String> createButtonClickObservable() {  
  
    // 2  
    return Observable.create(new ObservableOnSubscribe<String>() {  
  
        // 3  
        @Override  
        public void subscribe(final ObservableEmitter<String> emitter) throws Exception  
        // 4  
        {  
            mSearchButton.setOnClickListener(new View.OnClickListener() {  
                @Override  
                public void onClick(View view) {  
                    // 5  
                    emitter.onNext(mQueryEditText.getText().toString());  
                }  
            });  
  
            // 6  
            emitter.setCancellable(new Cancellable() {  
                @Override  
                public void cancel() throws Exception {  
                    // 7  
                    mSearchButton.setOnClickListener(null);  
                }  
            });  
        }  
    });  
}
```

上面这段代码做了以下几件事情

1. 定义了一个方法会返回一个Observable，泛型是String类型。
2. 通过 Observable.create() 创建了一个observable，并提供了一个 ObservableOnSubscribe。
3. 在参数的内部类中覆写了 subscribe() 方法。
4. 给搜索按钮mSearchButton添加了一个点击事件。
5. 当点击事件触发时，调用emitter 的onNext 方法，并传递了当前mQueryEditText的值。
6. 在Java中保持引用容易造成内存泄漏，在不再需要的时候及时移除listeners是一个好习惯，那么这里怎么移除呢？ObservableEmitter 有一个 setCancellable() 方法。通



过重写cancel()方法，然后当Observable 被处理的时候这个实现会被回调，比如已经结束或者是所有的观察者都解除了订阅。

7. 通过setOnClickListener(null) 来移除监听。

现在被观察者Observable 已经有了，还需要观察者来进行订阅，在此之前，我们先看看另一个接口， Consumer ，它可以十分简单的从emitter 接收到数据。

```
public interface Consumer<T> {  
    void accept(T t) throws Exception;  
}
```

如果仅是想要简单的订阅一下Observable，这个接口是很方便的。

Observable 的接口方法 subscribe() 可以接收很多类型的参数，你可以订阅一个全参数的版本，只要你实现其中所有的方法就可以。如果只是想要接收一下发射的数据，可以使用单一的 Consumer 的版本，这样只需要实现一个方法，而且也是 onNext 。

我们可以直接在Activity的OnStart方法中来实现这个

```
@Override  
protected void onStart() {  
    super.onStart();  
    // 1  
    Observable<String> searchTextObservable = createButtonClickObservable();  
  
    searchTextObservable  
        // 2  
        .subscribe(new Consumer<String>() {  
            //3  
            @Override  
            public void accept(String query) throws Exception {  
                // 4  
                showResult(mCheeseSearchEngine.search(query));  
            }  
        });  
}
```

其中Consumer需要导的包是

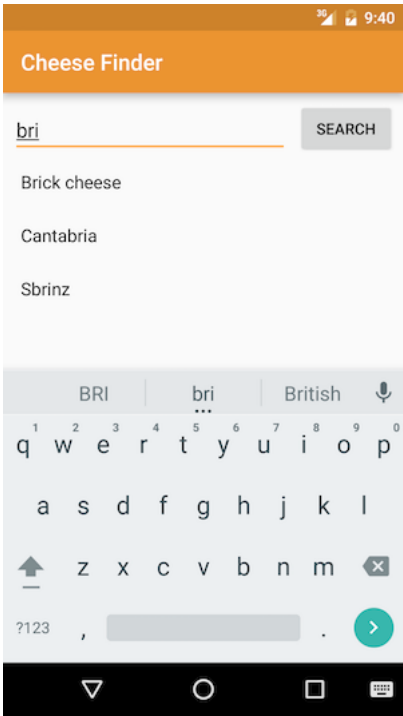
```
import io.reactivex.functions.Consumer;
```

依次解释一下上面每一步

1. 创建一个Observable 基于前面写的事件监听代码
2. 通过subscribe方法来订阅这个Observable ，并提供一个单一的 Consumer
3. 重写Consumer 方法，这会在按钮点击的时候接收到发射出来的EditText的值
4. 搜索并展示结果

这样一个简单的实现也写完了，运行一下APP，跑出来的结果就像下面这样





RxJava线程模型

虽然已经像模像样的写了一个小程序，但其实存在一些问题。当按钮按下去后这个UI线程实际上被阻塞住了
如果在控制台可能可以看到这样的提示

```
> 08-24 14:36:34.554 3500-3500/com.raywenderlich.cheesefinder I/Choreographer: Skipped frame 1. The application may be doing too much work on its main thread.
```

这是由于search 发生在主线程，如果是一个网络请求的话，Android会直接crash，抛出一个NetworkOnMainThreadException 的异常。如果不指定线程，那么RxJava的操作会一直在一个线程上。
通过 subscribeOn 和 observeOn 两个操作符能改变线程的执行状态。
subscribeOn 在操作链上最好只调用一次，如果多次调用，依然只有第一次生效
subscribeOn 用来指定 observable 在哪个线程上创建执行操作，如果想要通过 observables 发射事件给Android的View，那么需要保证订阅者在Android的UI线程上执行操作。
另一方面， observeOn 可以在链上调用多次，它主要是用来指定下一个操作在哪一个线程上执行，来个例子：

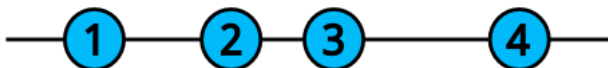
```
myObservable // observable will be subscribed on i/o thread
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .map(/* this will be called on main thread... */)
    .doOnNext(/* ...and everything below until next observeOn */)
    .observeOn(Schedulers.io())
    .subscribe(/* this will be called on i/o thread */);
```

主要用到三种schedulers：
Schedulers.io(): 适合I/O类型的操作，比如网络请求，磁盘操作。
Schedulers.computation(): 适合计算任务，比如事件循环或者回调处理。
AndroidSchedulers.mainThread() : 回调主线程，比如UI操作。

Map 操作符

map操作符通过运用一个方法把从一个observable 发射的数据再返回成另一个 observable给那些调用的。
比如你有一个observable称之为numbers，并且会发射一系列的值，如下所示

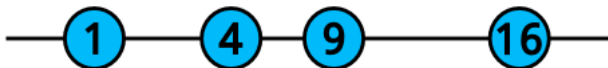




通过map操作符的apply方法

```
numbers.map(new Function<Integer, Integer>() {  
    @Override  
    public Integer apply(Integer number) throws Exception {  
        return number * number;  
    }  
})
```

然后结果就像下面这样



再来个实例，我们用这个操作符能够把前面的代码拆分一下

```
@Override  
protected void onStart() {  
    super.onStart();  
    Observable<String> searchTextObservable = createButtonClickObservable();  
  
    searchTextObservable  
        // 1  
        .observeOn(Schedulers.io())  
        // 2  
        .map(new Function<String, List<String>>() {  
            @Override  
            public List<String> apply(String query) {  
                return mCheeseSearchEngine.search(query);  
            }  
        })  
        // 3  
        .observeOn(AndroidSchedulers.mainThread())  
        .subscribe(new Consumer<List<String>>() {  
            @Override  
            public void accept(List<String> result) {  
                showResult(result);  
            }  
        }));  
}
```

简述一下代码，首先，指定下一次操作在I/O线程上，然后通过给的String，执行search返回一个结果列表，

再将线程从I/O上变更为主线程，showResult，展示返回的数据。

通过doOnNext显示进度条

为了用户体验，我们需要一个进度条

这里可以引入 doOnNext 操作符，doOnNext 有一个 Consumer，并且在每次observable发射数据的时候都会被调用，再改一下前面的代码



```

@Override
protected void onStart() {
    super.onStart();
    Observable<String> searchTextObservable = createButtonClickObservable();

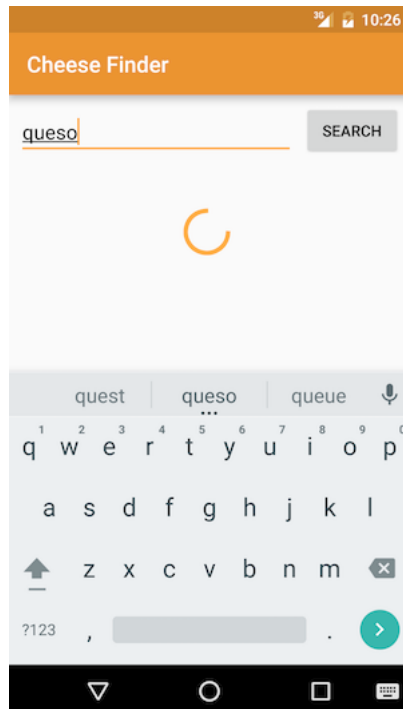
    searchTextObservable
        // 1
        .observeOn(AndroidSchedulers.mainThread())
        // 2
        .doOnNext(new Consumer<String>() {
            @Override
            public void accept(String s) {
                showProgressBar();
            }
        })
        .observeOn(Schedulers.io())
        .map(new Function<String, List<String>>() {
            @Override
            public List<String> apply(String query) {
                return mCheeseSearchEngine.search(query);
            }
        })
        .observeOn(AndroidSchedulers.mainThread())
        .subscribe(new Consumer<List<String>>() {
            @Override
            public void accept(List<String> result) {
                // 3
                hideProgressBar();
                showResult(result);
            }
        }));
}

```

每次在点击按钮的时候就能收到一个事件

首先把线程切换到主线程，然后在 `doOnNext` 里面来显示进度条，再把线程切换到子线程，来进行请求数据，最后在切换回来关闭进度条，展示数据。RxJava非常适合这种需求，代码也很清晰。

把这个例子跑起来的效果就像下面这样，点击的时候就显示进度条：



观察EditText变化

除了通过点击按钮来搜索，更好的方式就是根据EditText的text内容变化自动的搜索。

首先，就需要对EditText的内容变化进行订阅观察，先看代码实例：




```
//1
private Observable<String> createTextChangeObservable() {
    //2
    Observable<String> textChangeObservable = Observable.create(new ObservableOnSubscribe<String>() {
        @Override
        public void subscribe(final ObservableEmitter<String> emitter) throws Exception {
            //3
            final TextWatcher watcher = new TextWatcher() {
                @Override
                public void beforeTextChanged(CharSequence s, int start, int count, int after) {}

                @Override
                public void afterTextChanged(Editable s) {}

                //4
                @Override
                public void onTextChanged(CharSequence s, int start, int before, int count) {
                    emitter.onNext(s.toString());
                }
            };
        }
    });

    //5
    mQueryEditText.addTextChangedListener(watcher);

    //6
    emitter.setCancellable(new Cancellable() {
        @Override
        public void cancel() throws Exception {
            mQueryEditText.removeTextChangedListener(watcher);
        }
    });
}

// 7
return textChangeObservable;
}
```

分析一下上面这几步代码：

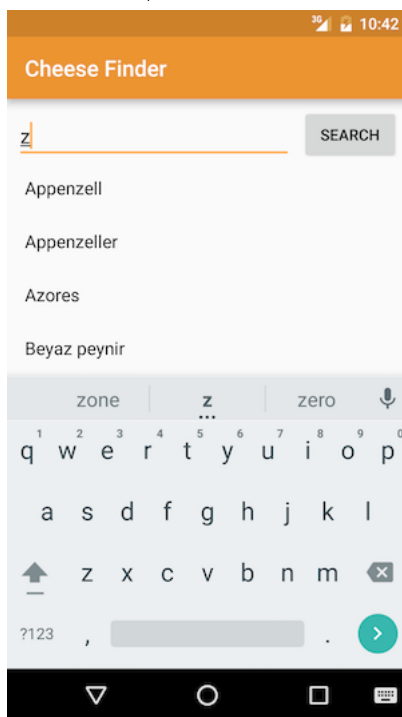
1. 定义一个方法返回一个EditText变化的observable
2. 通过 `observable.create` 创建一个`textChangeObservable`，传入一个`ObservableOnSubscribe` 对象
3. 在`subscribe` 方法中，创建一个`TextWatcher`，这是用来监听值变化的
4. 这里不用管 `beforeTextChanged()` 和 `afterTextChanged()`，在`onTextChanged` 里面，把这个数据通过`emitter.onNext` 发射出去，这样订阅的观察者就能接收到
5. 通过`addTextChangedListener`将Edittext绑定上这个watcher监听
6. 最后在emitter的`setCancellable`中去移除这个监听，防止内存泄漏

实现了这个Observable后就可以把前面的给替换掉

```
Observable<String> searchTextObservable = createTextChangeObservable();
```

再跑一次程序，就可以边输入边搜索了





内容长度拦截过滤

现在可能有一个需求是在输入长度比较短的时候不进行搜索，达到一定字符后才搜索，RxJava引入了一个 `filter` 操作符。

`filter`只会通过那些满足条件的item，`filter`通过一个 `Predicate`，这个接口内部有一个 `test` 方法用来决定是否满足条件，最后会返回一个boolean 值。

这里，`Predicate` 拿到的是一个输入字符String，如果长度大于或等于2，就返回true，表示满足条件。

```
return textChangeObservable
    .filter(new Predicate<String>() {
        @Override
        public boolean test(String query) throws Exception {
            return query.length() >= 2;
        }
    });
```

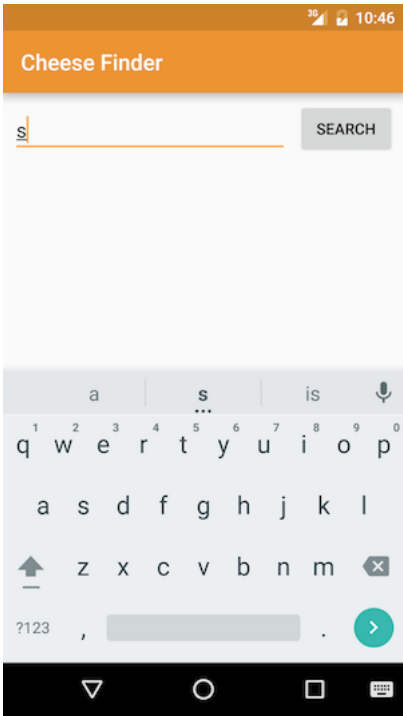
注意Predicate需要导的包是：

```
import io.reactivex.functions.Predicate;
```

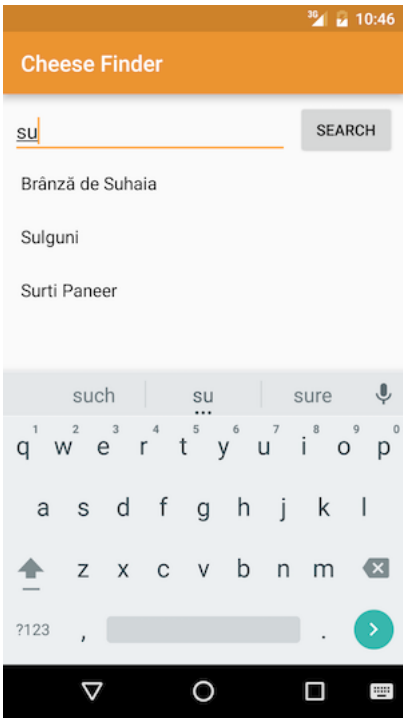
再前面创建Observable的代码后面加一个 `filter` 后，当query的长度不足2时，那这个值就不会被发射出去，然后订阅的就收不到这个消息。

跑起来就像这样，只输一个数，返回false，不会触发搜索。





再输一个字符就通过了filter的过滤。



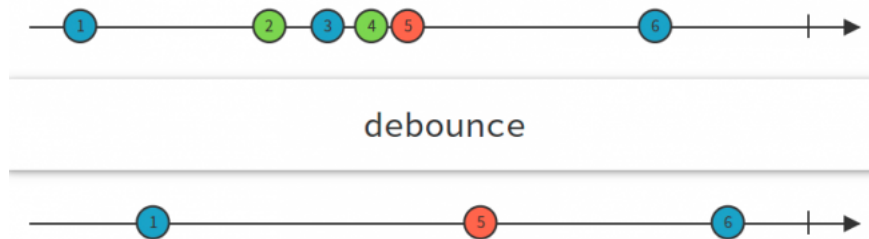
Debounce 操作符

有时我们对于EditText内容频繁变化的场景并不想每次变化都去新发送一个请求，所以，这里又引入了一个新的操作符 `debounce`，意思就是防抖动，这个和filter比较类似，也是一种拦截的策略。

这个操作符是根据item被发射的时间来进行过滤。每次在一个item被发射后，`debounce` 会等待一段指定长度的时间，然后才去发射下一个item。

如果在这段时间内都没有一个item发生，那么上一个最后的item会被发射出去，这样能保证起码有一个item能被发射成功。



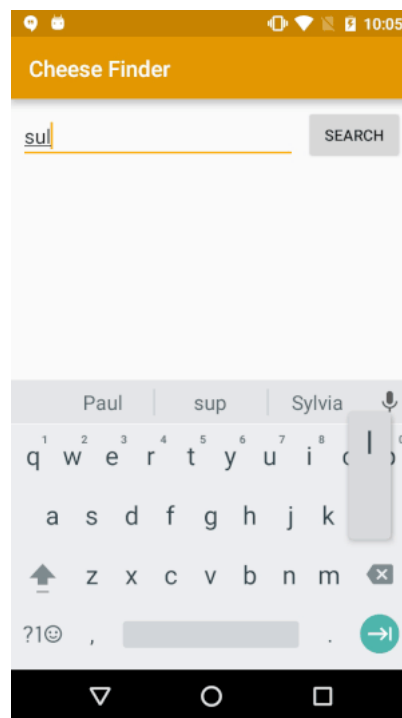


从图里看到，2，3，4，5触发的时间非常的接近，所以这一段时间内前三个都被过滤了，只留下了5。

在前面的 `createTextChangeObservable()` 中，我们再添加一个 `debounce` 操作符在 `filter` 的后面

```
return textChangeObservable
    .filter(new Predicate<String>() {
        @Override
        public boolean test(String query) throws Exception {
            return query.length() >= 2;
        }
    }).debounce(1000, TimeUnit.MILLISECONDS); // add this line
```

再跑一下APP，可以看到中间阶段直接省略了，最后搜索了一下结果值



Merge 操作符

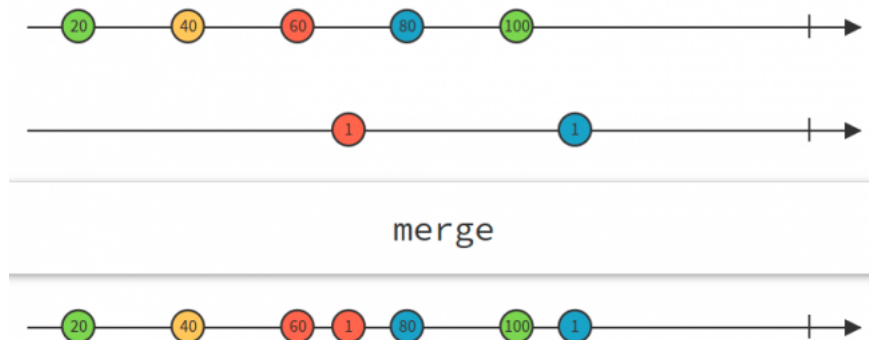
一开始我们实现了一个observable 是监听点击按钮的事件，然后又实现了一个 observable 是监听EditText的内容变化，那么怎么把这两个合二为一呢。

RxJava提供了很多的操作符来联合observables，但是其中最有用和简单的就是

`merge`。

`merge` 可以将两个或更多的observable 联合起来，合成一个单一的observable。





这里我们把前面两个observable 绑定起来

```
Observable<String> buttonClickStream = createButtonClickObservable();
Observable<String> textChangeStream = createTextChangeObservable();

Observable<String> searchTextObservable = Observable.merge(textChangeStream, buttonC
```

现在的效果就是前面的两种效果的结合体，无论是自动搜索还是手动搜索都是可以触发的。

RxJava和Activity/Fragment生命周期

前面我们实现过 `setCancellable` 方法，这个方法会在解除订阅的时候回调。

`Observable.subscribe()` 会返回一个 `Disposable`，`Disposable` 是一个接口，其中有两个方法：

```
public interface Disposable {
    void dispose(); // ends a subscription
    boolean isDisposed(); // returns true if resource is disposed (unsubscribed)
}
```

我们先在 `CheeseActivity` 中定义一个 `Disposable`

```
private Disposable mDisposable;
```

在 `onStart()` 中，把 `subscribe()` 的返回值赋给 `mDisposable`

```
mDisposable = searchTextObservable // change this line
    .observeOn(AndroidSchedulers.mainThread())
    .doOnNext(new Consumer<String>() {
        @Override
        public void accept(String s) {
            showProgressBar();
        }
    })
    .observeOn(Schedulers.io())
    .map(new Function<String, List<String>>() {
        @Override
        public List<String> apply(String query) {
            return mCheeseSearchEngine.search(query);
        }
    })
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new Consumer<List<String>>() {
        @Override
        public void accept(List<String> result) {
            hideProgressBar();
            showResult(result);
        }
    });
```

最后我们就能在 `onStop()` 中去解除这个订阅，代码如下：



```
@Override
protected void onStop() {
    super.onStop();
    if (!mDisposable.isDisposed()) {
        mDisposable.dispose();
    }
}
```

这样就解除了订阅。

后记

你可以下载这篇文章中的代码程序，下载地址 (<https://koenig-media.raywenderlich.com/uploads/2016/12/CheeseFinder-final.zip>)
当然这篇文章只是讲到了RxJava世界的一小点，比如，JakeWharton大神的库 RxBinding (<https://github.com/JakeWharton/RxBinding>)，这个库里面包括大量的 Android View的API，你可以通过调用 RxView.clicks(viewVariable) 来创建一个点击事件 observable。
除此之外，学习更多有关RxJava的知识，可以看 官方文档 (<http://reactivex.io/documentation/operators.html>)。

RxJava (/nb/9849550)

举报文章 © 著作权归作者所有

 **sheepm (/u/cd2a290d5feb)**
写了 46434 字，被 182 人关注，获得了 452 个喜欢
(/u/cd2a290d5feb)




+ 关注

翻译国外文章和源码解析的小地方

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持

喜欢 (/sign_in) | 25



更多分享


(<http://cwassets.jianshu.io/notes/images/9178864>






登录 (/sign_in) 发表评论



4条评论 只看作者

按喜欢排序 按时间正序 按时间倒序


 **alexfqyp (/u/dea23bf05d1e)**
2楼 · 2017.03.07 16:27
(/u/dea23bf05d1e)
不错的文章 讲的很详细 看别的文章都只了解用法 不知道原理

 赞  回复

 **古月西城 (/u/115aebbfcefe)**
3楼 · 2017.03.22 15:47
(/u/115aebbfcefe)
写得不错，终于有能看得懂的 RxAndroid 文章

 赞  回复





wenld_ (/u/99f514ea81b3)

4楼 · 2017.04.06 23:04

(/u/99f514ea81b3)

写的通俗易懂。膜拜

👍 赞

💬 回复



StephenCMZ (/u/998eddf8739d)

5楼 · 2017.04.14 14:17

(/u/998eddf8739d)

给你一个大写加粗的赞

👍 赞

💬 回复

被以下专题收入，发现更多相似内容

Android知识

Android开发

Android..

RxJava

Android..

Android UI

Android

RxJava系...

Android..

Android Dev

线程

