一个Observable可以发出零个或者多个事件，知道结束或者出错。每发出一个事件，就会调用它的Subscriber的onNext方法，最后调用Subscriber.onCompleted()或者Subscriber.onError()结束。

Observable<String> myObservable = Observable.create(

    new Observable.OnSubscribe<String>() {

        @Override

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

            sub.onNext("Hello, world!");

            sub.onCompleted();

        }

    }

);

这里定义的Observable对象仅仅发出一个Hello World字符串，然后就结束了。接着我们创建一个Subscriber来处理Observable对象发出的字符串。

Subscriber<String> mySubscriber = new Subscriber<String>() {

    @Override

    public void onNext(String s) { System.out.println(s); }

    @Override

    public void onCompleted() { }

    @Override

    public void onError(Throwable e) { }

};

这里subscriber仅仅就是打印observable发出的字符串。

通过subscribe函数就可以将我们定义的myObservable对象和mySubscriber对象关联起来，这样就完成了subscriber对observable的订阅。

myObservable.subscribe(mySubscriber);

Observable.just用来创建只发出一个事件就结束的Observable对象，上面创建Observable对象的代码可以简化为一行：

Observable<String> myObservable = Observable.just("Hello, world!");

接下来看看如何简化Subscriber，上面的例子中，我们其实并不关心OnComplete和OnError，我们只需要在onNext的时候做一些处理，这时候就可以使用Action1类。

Action1<String> onNextAction = new Action1<String>() {

    @Override

    public void call(String s) {

        System.out.println(s);

    }

};

subscribe方法有一个重载版本，接受三个Action1类型的参数，分别对应OnNext，OnComplete， OnError函数。

myObservable.subscribe(onNextAction, onErrorAction, onCompleteAction);

这里我们并不关心onError和onComplete，所以只需要第一个参数就可以：

myObservable.subscribe(onNextAction);

上面的代码最终可以写成这样：

Observable.just("Hello, world!")

    .subscribe(new Action1<String>() {

        @Override

        public void call(String s) {

              System.out.println(s);

        }

});

使用java8的lambda可以使代码更简洁：

Observable.just("Hello, world!").subscribe(s -> System.out.println(s));

操作符用于在Observable和最终的Subscriber之间修改Observable发出的事件。

map操作符，用来把把一个事件转换为另一个事件：

Observable.just("Hello, world!")

  .map(new Func1<String, String>() {

      @Override

      public String call(String s) {

          return s + " -Dan";

      }

  })

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

使用lambda可以简化为：

Observable.just("Hello, world!")

    .map(s -> s + " -Dan")

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

map操作符更有趣的一点是它不必返回Observable对象返回的类型，你可以使用map操作符返回一个发出新的数据类型的observable对象。  
比如上面的例子中，subscriber并不关心返回的字符串，而是想要字符串的hash值：

Observable.just("Hello, world!")

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

        @Override

        public Integer call(String s) {

            return s.hashCode();

        }

    })

    .subscribe(i -> System.out.println(Integer.toString(i)));

使用lambda可以进一步简化代码：

Observable.just("Hello, world!")

    .map(s -> s.hashCode())

    .subscribe(i -> System.out.println(Integer.toString(i)));

前面说过，Subscriber做的事情越少越好，我们再增加一个map操作符：

Observable.just("Hello, world!")

    .map(s -> s.hashCode())

    .map(i -> Integer.toString(i))

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

Observable和Subscriber可以做任何事情  
Observable可以是一个数据库查询，Subscriber用来显示查询结果；

Observable可以是屏幕上的点击事件，Subscriber用来响应点击事件；

Observable可以是一个网络请求，Subscriber用来显示请求结果。  
  
Observable和Subscriber是独立于中间的变换过程的。  
在Observable和Subscriber中间可以增减任何数量的map。整个系统是高度可组合的，操作数据是一个很简单的过程。

假设我有这样一个方法，这个方法根据输入的字符串返回一个网站的url列表：

Observable<List<String>> query(String text);

现在我希望构建一个健壮系统，它可以查询字符串并且显示结果。

根据上面的内容，我们可能会写出下面的代码：

query("Hello, world!")

    .subscribe(urls -> {

        for (String url : urls) {

            System.out.println(url);

        }

    });

这种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。一旦我们想要更改每一个URL，只能在Subscriber中来做。

Observable.from()方法，它接收一个集合作为输入，然后每次输出一个元素给subscriber：

Observable.from("url1", "url2", "url3")

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

我们来把这个方法使用到刚才的场景：

query("Hello, world!")

    .subscribe(urls -> {

        Observable.from(urls)

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

    });

虽然去掉了for each循环，但是代码依然看起来很乱。多个嵌套的subscription不仅看起来很丑，难以修改，更严重的是它会破坏某些我们现在还没有讲到的RxJava的特性。

Observable.flatMap()接收一个Observable的输出作为输入，同时输出另外一个Observable。

query("Hello, world!")

    .flatMap(new Func1<List<String>, Observable<String>>() {

        @Override

        public Observable<String> call(List<String> urls) {

            return Observable.from(urls);

        }

    })

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

使用lambda可以大大简化代码长度：

query("Hello, world!")

    .flatMap(urls -> Observable.from(urls))

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

理解flatMap的关键点在于，flatMap输出的新的Observable正是我们在Subscriber想要接收的。

现在Subscriber不再收到List<String>，而是收到一些列单个的字符串，就像Observable.from()的输出一样。

flatMap()实在不能更赞了，它可以返回任何它想返回的Observable对象。  
比如下面的方法：

// 返回网站的标题，如果404了就返回null

Observable<String> getTitle(String URL);

接着前面的例子，现在我不想打印URL了，而是要打印收到的每个网站的标题。问题来了，我的方法每次只能传入一个URL，并且返回值不是一个String，而是一个输出String的Observabl对象。使用flatMap()可以简单的解决这个问题。

query("Hello, world!")

    .flatMap(urls -> Observable.from(urls))

    .flatMap(new Func1<String, Observable<String>>() {

        @Override

        public Observable<String> call(String url) {

            return getTitle(url);

        }

    })

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

使用lambda：

query("Hello, world!")

    .flatMap(urls -> Observable.from(urls))

    .flatMap(url -> getTitle(url))

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

我们可以将任意多个API调用链接起来。大家应该都应该知道同步所有的API调用，然后将所有API调用的回调结果组合成需要展示的数据是一件多么蛋疼的事情。这里我们成功的避免了callback hell（多层嵌套的回调，导致代码难以阅读维护）。现在所有的逻辑都包装成了这种简单的响应式调用。

**丰富的操作符**

如果url不存在，getTitle()返回null。我们不想输出"null"，那么我们可以从返回的title列表中过滤掉null值！

query("Hello, world!")

    .flatMap(urls -> Observable.from(urls))

    .flatMap(url -> getTitle(url))

    .filter(title -> title != null)

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

filter()输出和输入相同的元素，并且会过滤掉那些不满足检查条件的。

如果我们只想要最多5个结果：

query("Hello, world!")

    .flatMap(urls -> Observable.from(urls))

    .flatMap(url -> getTitle(url))

    .filter(title -> title != null)

    .take(5)

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

take()输出最多指定数量的结果。

如果我们想在打印之前，把每个标题保存到磁盘：

query("Hello, world!")

    .flatMap(urls -> Observable.from(urls))

    .flatMap(url -> getTitle(url))

    .filter(title -> title != null)

    .take(5)

    .doOnNext(title -> saveTitle(title))

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

doOnNext()允许我们在每次输出一个元素之前做一些额外的事情，比如这里的保存标题。

操作符可以让你对数据流做任何操作。

将一系列的操作符链接起来就可以完成复杂的逻辑。代码被分解成一系列可以组合的片段。这就是响应式函数编程的魅力。

另外，RxJava也使我们处理数据的方式变得更简单。在最后一个例子里，我们调用了两个API，对API返回的数据进行了处理，然后保存到磁盘。但是我们的Subscriber并不知道这些，它只是认为自己在接收一个Observable<String>对象。良好的封装性也带来了编码的便利！

**[响应式的好处](http://blog.csdn.net/lzyzsd/article/details/44891933)**

**错误处理**

到目前为止，我们都没怎么介绍onComplete()和onError()函数。这两个函数用来通知订阅者，被观察的对象将停止发送数据以及为什么停止（成功的完成或者出错了）。

下面的代码展示了怎么使用这两个函数：

Observable.just("Hello, world!")

.map(s -> potentialException(s))

.map(s -> anotherPotentialException(s))

.subscribe(new Subscriber<String>() {

@Override

Public void onNext(String s) { System.out.println(s); }

@Override

public void onCompleted() { System.out.println("Completed!"); }

@Override

public void onError(Throwable e) { System.out.println("Ouch!"); }

});

代码中的potentialException() 和 anotherPotentialException()有可能会抛出异常。每一个Observerable对象在终结的时候都会调用onCompleted()或者onError()方法，所以Demo中会打印”Completed!”或者”Ouch!”。

这种模式有以下几个优点：

1.只要有异常发生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，Observerable的操作符调用链中一旦有一个抛出了异常，就会直接执行onError()方法。

3.你能够知道什么时候订阅者已经接收了全部的数据。

知道什么时候任务结束能够帮助简化代码的流程。（虽然有可能Observable对象永远不会结束）

我觉得这种错误处理方式比传统的错误处理更简单。传统的错误处理中，通常是在每个回调中处理错误。这不仅导致了重复的代码，并且意味着每个回调都必须知道如何处理错误，你的回调代码将和调用者紧耦合在一起。

使用RxJava，Observable对象根本不需要知道如何处理错误！操作符也不需要处理错误状态，一旦发生错误，就会跳过当前和后续的操作符。所有的错误处理都交给订阅者来做。

**调度器**

假设你编写的**[Android](http://lib.csdn.net/base/android" \o "Android知识库" \t "http://blog.csdn.net/lzyzsd/article/details/_blank)** app需要从网络请求数据。网络请求需要花费较长的时间，因此你打算在另外一个线程中加载数据。那么问题来了！

编写多线程的Android应用程序是很难的，因为你必须确保代码在正确的线程中运行，否则的话可能会导致app崩溃。最常见的就是在非主线程更新UI。

使用RxJava，你可以使用subscribeOn()指定观察者代码运行的线程，使用observerOn()指定订阅者运行的线程：

myObservableServices.retrieveImage(url)

.subscribeOn(Schedulers.io())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(bitmap -> myImageView.setImageBitmap(bitmap));

是不是很简单？任何在我的Subscriber前面执行的代码都是在I/O线程中运行。最后，操作view的代码在主线程中运行.

最棒的是我可以把subscribeOn()和observerOn()添加到任何Observable对象上。这两个也是操作符！。我不需要关心Observable对象以及它上面有哪些操作符。仅仅运用这两个操作符就可以实现在不同的线程中调度。

如果使用AsyncTask或者其他类似的，我将不得不仔细设计我的代码，找出需要并发执行的部分。使用RxJava，我可以保持代码不变，仅仅在需要并发的时候调用这两个操作符就可以。

**订阅（Subscriptions）**

当调用Observable.subscribe()，会返回一个Subscription对象。这个对象代表了被观察者和订阅者之间的联系。

Subscription subscription = Observable.just("Hello, World!") .subscribe(s -> System.out.println(s));

你可以在后面使用这个Subscription对象来操作被观察者和订阅者之间的联系.

subscription.unsubscribe();

System.out.println("Unsubscribed=" + subscription.isUnsubscribed());

// Outputs "Unsubscribed=true"

RxJava的另外一个好处就是它处理unsubscribing的时候，会停止整个调用链。如果你使用了一串很复杂的操作符，调用unsubscribe将会在他当前执行的地方终止。不需要做任何额外的工作！

**[在Android中使用响应式编程](http://blog.csdn.net/lzyzsd/article/details/45033611)**

RxAndroid是RxJava的一个针对Android平台的扩展。它包含了一些能够简化Android开发的工具。

首先，AndroidSchedulers提供了针对Android的线程系统的调度器。需要在UI线程中运行某些代码？很简单，只需要使用AndroidSchedulers.mainThread():

retrofitService.getImage(url)

.subscribeOn(Schedulers.io())

.observeOn(AndroidSchedulers.mainThread())

.subscribe(bitmap -> myImageView.setImageBitmap(bitmap));

如果你已经创建了自己的Handler，你可以使用HandlerThreadScheduler1将一个调度器链接到你的handler上。

接着要介绍的就是AndroidObservable，它提供了跟多的功能来配合Android的生命周期。bindActivity()和bindFragment()方法默认使用AndroidSchedulers.mainThread()来执行观察者代码，这两个方法会在Activity或者Fragment结束的时候通知被观察者停止发出新的消息。

AndroidObservable.bindActivity(this, retrofitService.getImage(url))

.subscribeOn(Schedulers.io())

.subscribe(bitmap -> myImageView.setImageBitmap(bitmap);

我自己也很喜欢AndroidObservable.fromBroadcast()方法，它允许你创建一个类似BroadcastReceiver的Observable对象。

下面的例子展示了如何在网络变化的时候被通知到：

IntentFilter filter = new IntentFilter(ConnectivityManager.CONNECTIVITY\_ACTION);

AndroidObservable.fromBroadcast(context, filter)

.subscribe(intent -> handleConnectivityChange(intent));

最后要介绍的是ViewObservable,使用它可以给View添加了一些绑定。如果你想在每次点击view的时候都收到一个事件，可以使用ViewObservable.clicks()，或者你想监听TextView的内容变化，可以使用ViewObservable.text()。

ViewObservable.clicks(mCardNameEditText, false)

.subscribe(view -> handleClick(view));

**Retrofit**

大名鼎鼎的Retrofit库内置了对RxJava的支持。通常调用发可以通过使用一个Callback对象来获取异步的结果：

@GET("/user/{id}/photo")

void getUserPhoto(@Path("id") int id, Callback<Photo> cb);

使用RxJava，你可以直接返回一个Observable对象。

@GET("/user/{id}/photo")

Observable<Photo> getUserPhoto(@Path("id") int id);

现在你可以随意使用Observable对象了。你不仅可以获取数据，还可以进行变换。   
Retrofit对Observable的支持使得它可以很简单的将多个REST请求结合起来。比如我们有一个请求是获取照片的，还有一个请求是获取元数据的，我们就可以将这两个请求并发的发出，并且等待两个结果都返回之后再做处理：

Observable.zip( service.getUserPhoto(id), service.getPhotoMetadata(id), (photo, metadata) -> createPhotoWithData(photo, metadata)) .subscribe(photoWithData -> showPhoto(photoWithData));

**遗留代码，运行极慢的代码**

Retrofit可以返回Observable对象，但是如果你使用的别的库并不支持这样怎么办？或者说一个内部的内码，你想把他们转换成Observable的？有什么简单的办法没？

绝大多数时候Observable.just() 和 Observable.from() 能够帮助你从遗留代码中创建 Observable 对象:

private Object oldMethod() { ... }

public Observable<Object> newMethod() { return Observable.just(oldMethod()); }

上面的例子中如果oldMethod()足够快是没有什么问题的，但是如果很慢呢？调用oldMethod()将会阻塞住他所在的线程。   
为了解决这个问题，可以参考我一直使用的方法–使用defer()来包装缓慢的代码：

private Object slowBlockingMethod() { ... }

public Observable<Object> newMethod() { return Observable.defer(() -> Observable.just(slowBlockingMethod())); }

现在，newMethod()的调用不会阻塞了，除非你订阅返回的observable对象。

**生命周期**

我把最难的部分留在了最后。如何处理Activity的生命周期？主要就是两个问题：   
1.在configuration改变（比如转屏）之后继续之前的Subscription。

比如你使用Retrofit发出了一个REST请求，接着想在listview中展示结果。如果在网络请求的时候用户旋转了屏幕怎么办？你当然想继续刚才的请求，但是怎么搞？

2.Observable持有Context导致的内存泄露

这个问题是因为创建subscription的时候，以某种方式持有了context的引用，尤其是当你和view交互的时候，这太容易发生！如果Observable没有及时结束，内存占用就会越来越大。   
不幸的是，没有银弹来解决这两个问题，但是这里有一些指导方案你可以参考。

第一个问题的解决方案就是使用RxJava内置的缓存机制，这样你就可以对同一个Observable对象执行unsubscribe/resubscribe，却不用重复运行得到Observable的代码。cache() (或者 replay())会继续执行网络请求（甚至你调用了unsubscribe也不会停止）。这就是说你可以在Activity重新创建的时候从cache()的返回值中创建一个新的Observable对象。

Observable<Photo> request = service.getUserPhoto(id).cache();

Subscription sub = request.subscribe(photo -> handleUserPhoto(photo));

// ...When the Activity is being recreated...

sub.unsubscribe();

// ...Once the Activity is recreated...

request.subscribe(photo -> handleUserPhoto(photo));

注意，两次sub是使用的同一个缓存的请求。当然在哪里去存储请求的结果还是要你自己来做，和所有其他的生命周期相关的解决方案一样，必须在生命周期外的某个地方存储。（retained fragment或者单例等等）。

第二个问题的解决方案就是在生命周期的某个时刻取消订阅。一个很常见的模式就是使用CompositeSubscription来持有所有的Subscriptions，然后在onDestroy()或者onDestroyView()里取消所有的订阅。

private CompositeSubscription mCompositeSubscription = new CompositeSubscription();

private void doSomething() { mCompositeSubscription.add( AndroidObservable.bindActivity(this, Observable.just("Hello, World!")) .subscribe(s -> System.out.println(s))); }

@Override

protected void onDestroy() { super.onDestroy(); mCompositeSubscription.unsubscribe(); }

你可以在Activity/Fragment的基类里创建一个CompositeSubscription对象，在子类中使用它。

注意! 一旦你调用了 CompositeSubscription.unsubscribe()，这个CompositeSubscription对象就不可用了, 如果你还想使用CompositeSubscription，就必须在创建一个新的对象了。

两个问题的解决方案都需要添加额外的代码，如果谁有更好的方案，欢迎告诉我。