1.概念：

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）

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。

(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。

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。

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

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

just(T...): 将传入的参数依次发送出来。

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。

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();

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

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：

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。

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。

将传入的 Subscriber 作为 Subscription 返回。这是为了方便 unsubscribe().

线程控制 —— Scheduler (一)

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。

注意：如果不是build 模式，会失效，都会在主线程执行

2) Scheduler 的原理 (一)

变换

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

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);

}

});

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

Func 和 Action 的区别

Func1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

flatMap(): 这是一个很有用但非常难理解的变换

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。

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);

2) 变换的原理：lift()

// 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);

}

});

}