# RxJava Notes

## 总概念解释：

## subscriber 观察者 （订阅者）

1.subscriber下的方法主要是onStart 以及unsubscribe方法 以及基本的observer的方法

## observable 被观察者

1.observable 有大量的方法 首先是一些构造方法： create just from

2.映射方法map flatmap等 用于将事件转化分发。

## 3 onSubscriber 事件排列分发者

在创建observable时会创建一个onSubscriber对象用于 将观察者的事件进行排列

RxJava 是把java 语言的一系列异步任务操作 当做一个事件流进行。 通过观察者Observable的各种调用 最终实现 异步任务 ，隐含了多线程的操作以及线程的交互，

### 观察者模式----

A 观察 B B 有任何变化 A会立刻做出对应的响应。 代码中使用register 或者subsribe 方法 订阅 一个状态，让A订阅B的一个操作， 在B进行操作的时候A立马执行对应的操作 盗一张图：



可以了解为可观察者（被观察者） -- 发出事件（执行方法） 观察者 观察到事件 --- 执行对应的操作。

### RxJava的基本 组成：

Observable (可观察者，即被观察者)、 Observer (观察者)、 subscribe (订阅)、事件（一般是执行方法）。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。

### RxJava 事件：

onNext 即被变成则事件第一回调

onCompleted 整个事件队列结束： RxJava 把一系列的操作都当做一个个的事件 形成一个队列 一步步执行onNext 当完结时（没有next的时候） 需要触发onCompleted作为结束一个事件流的标志

onError 当事件流中发生异常的时候 调用 用于处理异常 事件流同时会停止。

Notice： onError 和onCompleted 在一个事件流之中必须存在一个 他们不可能同时存在 。是互斥的、

# 观察者模块

## Interface Observer{

## }

包含了3个方法也就是事件流的方法next completed error

## Abstract class Subscriber{

## }

Subscriber Observer 的一个抽象实现 ：对observer进行了一定的拓展；在observer的基础上添加了几个方法：

1.onStart ：在事件之前在observable 发送事件之间调用 一般是做预处理操作

Notice： start 只能在subscribe发生的线程调用无法指定线程，因此不可以随便调用ui的方法 指定线程 需要 doOnSubscribe

2.unsubscribe 用于取消订阅 ：方法调用之后订阅者将不再接收处理事件， 可以使用isUnsubscribe判断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。

# Observable 被观察者

Observable 被观察者

调用静态方法create来创建一个observable 在create 中需要传入一个Observable.onSubscribe 它相当于一个计划表，在onSubscriber的call中调用subscriber（观察者）的方法。来为被观察者 添加事件触发规则； 这样 被观察者就可以在被订阅的时候回调观察者的方法。

快速构造 被观察者 事件流的方法

1. just（T ...）； 传入不定长数字 =============依次调用next；
2. from（T[]） 传入数组 ====================依次调用next；

订阅观察者的方法

observable.subscribe(observer);

// 或者：

observable.subscribe(subscriber);

完成构造 观察者与被观察者就可以 订阅关联两个对象

public Subscription subscribe(Subscriber subscriber) {

subscriber.onStart(); -------------------------------------事件开始前

onSubscribe.call(subscriber);---------------------------开始事件流

return subscriber;----------------------返回Subscription便于后期调用unsubscribe方法

}

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 。

# 异步线程控制 –Scheduler

默认状态下 subscribe方法在哪个线程调用就会在当前线程下产生事件并且在当前线程下有观察者消费事件。当切换线程 多线程交互的时候就用到了scheduler

Scheduler 可以指定代码运行在什么线程内。

Schedulers 类 用于派发，新建线程

1.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。

2.Schedulers.newThread(): 总是启用新线程，并在新线程执行操作。

3.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。

4.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。

## 派发

Observable的方法 subscribeOn（） 以及observeOn方法就可以调用 传入对应的scheduler 也就是分配到对应的线程下 执行

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。

# 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.map（）.

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

@Override

public Bitmap call(String filePath) { // 参数类型 String

return getBitmapFromPath(filePath); // 返回类型 Bitmap

}

})

Observable create 时传入了T1 的事件流 map 方法传入func 在func中接受 T1 并将 T1 转化为T2 并且组建为新的事件流 的Type 执行接下来的操作。这个可以理解为事件直接映射映射。“倒入水， 产生果汁。”

## 2.flatMap（）

简单的说是一堆多映射： 传入一个T1 返回需要的所有T 2

A demo:

Observable.from(students)

.flatMap(new Func1<Student, Observable<Course>>() {

@Override

public Observable<Course> call(Student student) {

return Observable.from(student.getCourses());

}

})

.subscribe(subscriber);

flatMap 传入的func中的第二个泛型参数也就是call的返回类型也是一个observable

通过新建一个observable 并且让他发生时间所有新建的observable 都会被汇入一个总的observable 这个总的在同意发送事件个观察者

将原事件 平铺成新的事件统一的发送出去

## lift ==== map的实现并不建议自定义使用

code

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);

}

});

}

形容过于复杂 目前为截图 ：简单的说也就是 lift 首先会商城一个新的observable 新的observable接受事件 接着它传入一个operator operator 拦截了原型的subscriber 并且地调用它的call 产生一个新的subscriber 新的subscriber 会被原型的onSubscriber给调用 。 原先的onSubscriber 会将事件发给新的subscriber。新的subscriber将处理的结果发向原先的subscriber。

