**RxJava操作符系列一**

在RxJava中最重要的就是Observable（被观察者），subscribe（订阅），Observer（观察者）或者Subscriber（订阅者），Observable也就是数据（事件）源，Subscriber负责接收以及处理数据（事件）。当然要想实现两者通信，需要有一种机制那就是订阅。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。

例如张三（观察者）想看某款新闻软件的科技信息（被观察者），由于科技信息是每天推送或者不定时推送，如果张三一直盯着手机屏幕看并且刷新消息是不是又新的信息，显然不现实。这时候就可以通过张三 subscribe（订阅）科技信息，而实现当有新的科技信息时自动给张三推送消息，在这期间，张三并不需要一直盯着屏幕刷新闻。在我们平时的认知中实现订阅应该是张三.subscribe(科技新闻)，不过在RxJava代码中实现订阅应该写成科技新闻.subscribe(张三)。

在RxJava中，有三个事件回调方法，分别是onNext(),OnError(),onCompleted(),onNext()是最终输出及处理数据的回调，在发射数据过程中出现错误异常会回调OnError()方法，当不会再有新的 onNext() 发出时，需要触发 onCompleted() 方法作为标志。OnError()和onCompleted()是互斥的。下面举一个最简单的例子

Observable observable2 = Observable.just("也许当初忙着微笑和哭泣", "忙着追逐天空中的流星", "人理所当然的忘记", "是谁风里雨里一直默默守护在原地");

Subscriber subscriber = new Subscriber<String>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " )

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: ")

}

@Override

public void onNext(String s) {

Log.e(TAG, "onNext: "+s )

}

};

observable.subscribe(subscriber);

运行后打印信息为

onNext: 也许当初忙着微笑和哭泣

onNext: 忙着追逐天空中的流星

onNext: 人理所当然的忘记

onNext: 是谁风里雨里一直默默守护在原地

onCompleted:

**Create**

我们可以使用该操作符从零开始创建一个Observable，给这个操作符传递一个被观察者作为参数的函数，并调用观察者的onNext，onError和onCompleted方法。如下

//被观察者

Observable<String> observable = Observable.create(new Observable.OnSubscribe<String>() {

@Override

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

//可以多次调用subscriber.onNext("大家好")发射数据

subscriber.onNext("大家好");

subscriber.onNext("我开始学习RxJava");

subscriber.onCompleted();

}

});

发送数据需要在回调方法call中调用subscriber的onNext(),onNext(T)发送的参数需要和Observable.OnSubscribe<T>()中参数相同，在上面我们传入的是String类型。创建后Observale后，我们需要创建Subscriber(观察者)去处理observable发送的数据。如下

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

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, e.getMessage());

}

@Override

public void onNext(String s) {

Log.e(TAG, "onNext："+s);

}

};

数据成功发送后，会回调Subscriber的onNext()的方法，其中的参数就是接收到的数据。当onNext（）接收数据完毕后会执行onCompleted（）,如果中途有环节出现错误异常，会执行onError()。现在观察者和被观察者都创建完毕了，他们执行还需要一个前提就是订阅，如果不订阅，observable并不会发射数据，subscribe也不会接收数据，订阅代码如下

observable.subscribe(subscriber);

执行后输出信息

onNext：大家好

onNext：我开始学习RxJava

onCompleted

**from**

该操作符是将其它种类的对象和数据类型转换为Observable，如果当你发射的的数据是同一种类型，而不是混合使用Observables和其它类型的数据，会非常方便。如下创建Observable

Integer[] integers = {1,2, 3, 4};

Observable<Integer> observable=Observable.from(integers);

Subscriber<String> subscriber = new Subscriber<Integer>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, e.getMessage());

}

@Override

public void onNext(Integer i) {

Log.e(TAG, "onNext："+i);

}

};

observable.subscribe(subscriber);

输出信息为

onNext：1

onNext：2

onNext：3

onNext：4

onCompleted

from操作符可以转换Future、Iterable和数组。对于Iterable和数组，产生的Observable会发射Iterable或数组的每一项数据。对于Future，它会发射Future.get()方法返回的单个数据，并且还可以增加通过: from(Future,timeout, timeUnit)指定超时时间，如果执行的时候Future超时会回调onError（）方法。

**just**

just将单个数据转换为发射那个数据的Observable，Just类似于From，但是From会将数组或Iterable的数据取出然后逐个发射，而Just只是简单的原样发射，将数组或Iterable当做单个数据，如果你传递null给Just，它会返回一个发射null值的Observable。不要误认为它会返回一个空Observable（完全不发射任何数据的Observable）。对于just可以接收1到10个数据，返回一个按参数列表顺序发射这些数据的Observable。

Observable.just(1 2, 3, 4)

.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: ");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: ");

}

@Override

public void onNext(Integer integer) {

Log.e(TAG, "onNext: " + integer);

}

});

输出

onNext:1

onNext:2

onNext:3

onNext:4

onCompleted:

对于just参数类型可以是多种，如下，传入两个类型数据

Observable.just(0, "one", 6, "two", 8, "three")

.subscribe(new Subscriber<Serializable>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: ");

}

@Override

public void onNext(Serializable serializable) {

Log.e(TAG, "onNext: "+serializable.toString());

}

});

则输出信息

onNext:0

onNext:one

onNext:6

onNext:two

onNext:8

onNext:three

onCompleted:

**Empty/Never/Error**

Empty:创建一个不发射任何数据但是正常终止的Observable，此时会回调onCompleted()

Never:创建一个不发射数据也不终止的Observable

Error:创建一个不发射数据以一个错误终止的Observable

error操作符需要一个Throwable参数，你的Observable会以此终止。这些操作符默认不在任何特定的调度器上执行，但是empty和error有一个可选参数是Scheduler，如果你传递了Scheduler参数，它们会在你指定的调度器上发送通知。

**Range**

该操作符创建特定整数序列的Observable，它接受两个参数，一个是范围的起始值，一个是范围的数据的数目。如果你将第二个参数设为0，将导致Observable不发射任何数据（如果设置为负数，会抛异常）。

Observable.range(1,4)

.subscribe(new Subscriber<Integer>() {

public String TAG="RXJAVA";

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: ");

}

@Override

public void onNext(Integer integer) {

Log.e(TAG, "onNext: "+integer);

}

});

输出信息

onNext: 1

onNext: 2

onNext: 3

onNext: 4

onCompleted:

你可以在代码实战中，更改第二个参数为负数，或者0，以及将第一个参数更改为你想测试的任意值，去观察执行日志帮助理解。

**Timer**

Timer操作符创建一个在给定的时间段之后返回一个特殊值的Observable。它在延迟一段给定的时间后发射一个简单的数字0 。

Observable.timer(1, TimeUnit.SECONDS)

.subscribe(new Subscriber<Long>() {

public String TAG="RXJAVA";

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: ");

}

@Override

public void onNext(Long integer) {

Log.e(TAG, "onNext:1111111 "+integer);

}

});

对于该操作符默认在computation调度器上执行的，如果你想在onNext()回调方法更新UI,需要通过observeOn(AndroidSchedulers.mainThread())设置，否则会调用onError()方法。当然Time也提供的一个有三个参数的方法timer(long,TimeUnit,Scheduler)可以指定 Scheduler 。

**Interval**

该操作符按固定的时间间隔发射一个无限递增的整数序列，它接受一个表示时间间隔的参数和一个表示时间单位的参数，当然该操作符合Timer一样，是在computation调度器上执行的，若想更新UI需要指定Scheduler 为AndroidSchedulers.mainThread()。

Subscription subscription = Observable.interval(1, TimeUnit.SECONDS)

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Action1<Long>() {

@Override

public void call(Long aLong) {

tv.append(" " + aLong + " ");

}

});

通过上面代码就会每隔1秒在tv上追加一个数字，并且会永远执行。如果在某个时刻不想继续输出，就需要要解除订阅。

if (subscription != null && !subscription.isUnsubscribed()) {

subscription.unsubscribe();

}

**Repeat**

该操作符是重复的发射某个数据序列，并且可以自己设置重复的次数。当接收到onComplete()会触发重订阅再次重复发射数据,当重复发射数据次数到达后执行onCompleted()。

String[] strs = {"也许当初忙着微笑和哭泣", "忙着追逐天空中的流星"};

Observable.from(strs).repeat(2).subscribe(new Subscriber<String>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: ");

}

@Override

public void onNext(String s) {

Log.e(TAG, "onNext: "+s );

tv1.append("\n" + s);

}

});

输出

onNext: 也许当初忙着微笑和哭泣

onNext: 忙着追逐天空中的流星

onNext: 也许当初忙着微笑和哭泣

onNext: 忙着追逐天空中的流星

onCompleted:

**Defer**

直到有观察者订阅时才创建Observable，并且为每个观察者创建一个新的Observable，该操作符能保证订阅执行时数据源是最新的数据。如下正常代码

String test="旧数据";

Observable observable=Observable.just(test);

Subscriber subscriber=new Subscriber() {

public String TAG="RXJAVA";

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: " );

}

@Override

public void onNext(Object o) {

Log.e(TAG, "onNext: "+o );

}

};

test="新数据";

observable.subscribe(subscriber);

输出

onNext: 旧数据

onCompleted:

通过上面代码和输出日志发现，虽然在后面讲数据test更新为新数据，但是并没有生效，要想使用最新的数据就需要使用defer操作符。此时更改使用defer

test="旧数据";

Observable<String> observable=Observable.defer(new Func0<Observable<String>>() {

@Override

public Observable<String> call() {

return Observable.just(test);

}

});

Subscriber subscriber=new Subscriber() {

public String TAG="RXJAVA";

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: " );

}

@Override

public void onNext(Object o) {

Log.e(TAG, "onNext: "+o );

}

};

test="新数据";

observable.subscribe(subscriber);

输出信息

onNext: 新数据

onCompleted:

通过新的打印信息，发现输出值已经是最新的数据。