**RxJava操作符系列二**

**Map**

该操作符是对原始Observable发射的每一项数据运用一个函数，然后返回一个发射这些结果的Observable。

例如我们有一个整形数组的数据，当大于5时输出为true，则代码实现

Integer[] integers = {0, 9, 6, 4, 8};

Observable.from(integers).map(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

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

return (integer > 5);

}})

.subscribe(new Subscriber<Boolean>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Boolean aBoolean) {

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

}});

日志输出信息

call: 0

onNext: false

call: 9

onNext: true

call: 6

onNext: true

call: 4

onNext: false

call: 8

onNext: true

onCompleted:

对于map,他可以将将数据源变换为你想要的类型，比如，你想获取有一个Student对象（里面age，name属性）那么我们可以通过map只获取name。接下来。我们再举个例子，我们根据一个图片路径获取图片并将图片设置到ImageView,然后将ImageView加的我们的布局中。

String path = Environment.getExternalStorageDirectory()+ File.separator+"aaa.jpg";

Observable.just(path)

.subscribeOn(Schedulers.io())

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

@Override

public Bitmap call(String s) {

Bitmap bitmap = BitmapFactory.decodeFile(s);

Log.e(TAG, "call: Bitmap"+bitmap);

return bitmap;

}})

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

@Override

public ImageView call(Bitmap bitmap) {

Log.e(TAG, "call: ImageView");

ImageView imageView = new ImageView(getActivity());

LinearLayout.LayoutParams params = new LinearLayout.LayoutParams(LinearLayout.LayoutParams.WRAP\_CONTENT, LinearLayout.LayoutParams.WRAP\_CONTENT);

imageView.setLayoutParams(params);

imageView.setImageBitmap(bitmap);

return imageView;

}})

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<ImageView>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(ImageView imageView) {

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

layout.addView(imageView);

}});

**Cast**

该操作符就是做一些强制类型转换操作的。例如，当我们在页面跳转时数据对象往往是序列化的，当我们在新的页面收到数据后就要强制转换为我们想要的类型。cast操作符也可以实现这样的功能。如下

Observable.just(serializable).cast(FileInfo.class).subscribe(new Subscriber<FileInfo>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(FileInfo fileInfo) {

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

tv1.append("\n"+fileInfo.toString());

}});

不过在该操作符实际用途并没有那么的广泛，很少用到，当然这个操作符也可以达到java 中instanceof相同的作用，用于类型检查，当不是该类型就会执行onError()方法。

**FlatMap**

该操作符与map操作符的区别是它将一个发射数据的Observable变换为多个Observables，然后将它们发射的数据合并后放进一个单独的Observable.

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

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

@Override

public Observable<String> call(final Integer integer) {

return Observable.create(new Observable.OnSubscribe<String>() {

@Override

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

Log.e(TAG, "call: FlatMap " + Thread.currentThread().getName());

try {

Thread.sleep(200);

subscriber.onNext(integer + 100 + " FlatMap");

subscriber.onCompleted();

} catch (InterruptedException e) {

e.printStackTrace();

subscriber.onError(e);

}

}})

.subscribeOn(Schedulers.newThread());}})

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<String>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(String s) {

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

}});

打印日志信息

call: FlatMap RxNewThreadScheduler-2

call: FlatMap RxNewThreadScheduler-3

call: FlatMap RxNewThreadScheduler-4

onNext: FlatMap 101 FlatMap

onNext: FlatMap 102 FlatMap

onNext: FlatMap 103 FlatMap

onCompleted: FlatMap

ConcatMap

该操作符是类似于最简单版本的flatMap，但是它按次序连接而不是合并那些生成的Observables，然后产生自己的数据序列.将上述flatMap代码更改如下

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

Observable.from(integers).concatMap(new Func1<Integer, Observable<String>>() {

@Override

public Observable<String> call(final Integer integer) {

return Observable.create(new Observable.OnSubscribe<String>() {

@Override

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

Log.e(TAG, "call:2 ConcatMap " + Thread.currentThread().getName());

try {

Thread.sleep(200);

subscriber.onNext(integer + 100 + " ConcatMap");

subscriber.onCompleted();

} catch (InterruptedException e) {

e.printStackTrace();

subscriber.onError(e);

}

}

}).subscribeOn(Schedulers.newThread());

}

}).observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<String>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(String s) {

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

}

});

输出日志信息

call:2 ConcatMap RxNewThreadScheduler-5

onNext: ConcatMap 101 ConcatMap

call:2 ConcatMap RxNewThreadScheduler-6

onNext: ConcatMap 102 ConcatMap

call:2 ConcatMap RxNewThreadScheduler-7

onNext: ConcatMap 103 ConcatMap

onCompleted: ConcatMap

通过该操作符和flatMap输出的日志信息，很容易看出flatMap并没有保证数据源的顺序性，但是ConcatMap操作符保证了数据源的顺序性。在应用中，如果你对数据的顺序性有要求的话，就需要使用ConcatMap。若没有要求，二者皆可使用。

SwitchMap

当原始Observable发射一个新的数据（Observable）时，它将取消订阅并停止监视产生执之前那个数据的Observable，只监视当前这一个.

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

Observable.from(integers).switchMap(new Func1<Integer, Observable<String>>() {

@Override

public Observable<String> call(Integer integer) {

Log.e(TAG, "call: SwitchMap" + Thread.currentThread().getName());

//如果不通过subscribeOn(Schedulers.newThread())在在子线程模拟并发操作，所有数据源依然会全部输出,也就是并发操作此操作符才有作用

//若在此通过Thread。sleep（）设置等待时间，则输出信息会不一样。相当于模拟并发程度

return Observable.just((integer + 100) + "SwitchMap").subscribeOn(Schedulers.newThread());

}

}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Subscriber<String>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(String s) {

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

}

});

输出日志信息

call: SwitchMapmaincall: SwitchMapmaincall: SwitchMapmain

onNext: SwitchMap 106SwitchMap

onCompleted: SwitchMap

当数据源较多时，并不一定是只输出最后一项数据，有可能输出几项数据，也可能是全部。

GroupBy

看到这个词你就应该想到了这个操作符的作用，就是你理解的含义，他将数据源按照你的约定进行分组。我们通过groupBy实行将1到10的数据进行就划分，代码如下

Observable.range(1, 10).groupBy(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer % 2 == 0;

}

}).subscribe(new Subscriber<GroupedObservable<Boolean, Integer>>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted:1 ");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError:1 ");

}

@Override

public void onNext(GroupedObservable<Boolean, Integer> booleanIntegerGroupedObservable) {

booleanIntegerGroupedObservable.toList().subscribe(new Subscriber<List<Integer>>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted:2 " );

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError:2 ");

}

@Override

public void onNext(List<Integer> integers) {

Log.e(TAG, "onNext:2 "+integers);

}

});

}

});

输出日志信息

onNext:2 [1, 3, 5, 7, 9]

onCompleted:2

onNext:2 [2, 4, 6, 8, 10]

onCompleted:2

onCompleted:1

在上面代码中booleanIntegerGroupedObservable变量有一个getKey（）方法,该方法返回的是分组的key，他的值就是groupBy方法call回调所用函数的值，在上面也就是integer % 2 == 0的值，及true和false。有几个分组也是有此值决定的。

Scan

操作符对原始Observable发射的第一项数据应用一个函数，然后将那个函数的结果作为自己的第一项数据发射。它将函数的结果同第二项数据一起填充给这个函数来产生它自己的第二项数据。它持续进行这个过程来产生剩余的数据序列。  
例如计算1+2+3+4的和

Observable.range(1,4).scan(new Func2<Integer, Integer, Integer>() {

@Override

public Integer call(Integer integer, Integer integer2) {

Log.e(TAG, "call: integer:"+integer+" integer2 "+integer2);

return integer+integer2;

}

}).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 );

}

});

输出日志信息

onNext: 1

call: integer:1 integer2 2

onNext: 3

call: integer:3 integer2 3

onNext: 6

call: integer:6 integer2 4

onNext: 10

onCompleted:

对于scan有一个重载方法，可以设置一个初始值，如上面代码，初始值设置为10，只需将scan加个参数scan（10，new Func2）。

Buffer

操作符将一个Observable变换为另一个，原来的Observable正常发射数据，变换产生的Observable发射这些数据的缓存集合，如果原来的Observable发射了一个onError通知，Buffer会立即传递这个通知，而不是首先发射缓存的数据，即使在这之前缓存中包含了原始Observable发射的数据。  
示例代码

Observable.range(10, 6).buffer(2).subscribe(new Subscriber<List<Integer>>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(List<Integer> integers) {

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

}

});

输出日志信息

onNext: [10, 11]

onNext: [12, 13]

onNext: [14, 15]

onCompleted:

上面一次性订阅两个数据，如果设置参数为6，就一次性订阅。buffer的另一重载方法buffer(count, skip)从原始Observable的第一项数据开始创建新的缓存（长度count），此后每当收到skip项数据，用count项数据填充缓存：开头的一项和后续的count-1项，它以列表(List)的形式发射缓存，取决于count和skip的值，这些缓存可能会有重叠部分（比如skip < count时），也可能会有间隙（比如skip > count时）。具体执行结果，你可以设置不同的skip和count观察输出日志，查看执行结果及流程。

Window

Window和Buffer类似，但不是发射来自原始Observable的数据包，它发射的是Observables，这些Observables中的每一个都发射原始Observable数据的一个子集，最后发射一个onCompleted通知。

Observable.range(10, 6).window(2).subscribe(new Subscriber<Observable<Integer>>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted1: ");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError1: ");

}

@Override

public void onNext(Observable<Integer> integerObservable) {

Log.e(TAG, "onNext1: ");

tv1.append("\n");

integerObservable.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted2: ");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError2: ");

}

@Override

public void onNext(Integer integer) {

Log.e(TAG, "onNext2: "+integer);

}

});

}

});

输出日志信息

onNext2: 10

onNext2: 11

onCompleted2:

onNext2: 12

onNext2: 13

onCompleted2:

onNext2: 14

onNext2: 15

onCompleted2:

onCompleted1:

window和buffer一样也有不同的重载方法。这两个操作符相对其他操作符不太容易理解，可以去[RxJava GitHub](https://github.com/ReactiveX/RxJava/wiki/Transforming-Observables" \t "_blank)理解，里面有图示解析。当然最好的理解方式就是通过更改变量的值，去观察输出的日志信息。