# **RxJava操作符系列三**

# Filter

该操作符接收一个Func1参数，我们可以在其中通过运用你自己的判断条件去判断我们要过滤的数据，当数据通过判断条件后返回true表示发射该项数据，否则就不发射，这样就过滤出了我们想要的数据。如下，我们过滤出不能被2整除的数

Integer[] ints = {1, 2, 3, 4, 5, 6, 7, 8, 9};

Observable observable = Observable.from(ints).filter(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer % 2 != 0;//返回true，就不会过滤掉，会发射数据，过滤掉返回false的值

}

});

Action1 action1 = new Action1<Integer>() {

@Override

public void call(Integer i) {

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

}

};

observable.subscribe(action1);

输出日志信息

call: 1call: 3call: 5call: 7call: 9

# ofType

该操作符是filter操作符的一个特殊形式。它过滤一个Observable只返回指定类型的数据，例如当数据源有字符串和int型数据时，我们想要过滤出字符串就可以使用这个操作符，如下示例代码

Observable.just(0, "one", 6, 4, "two", 8, "three", 1, "four", 0)

.ofType(String.class)

.subscribe(new Subscriber<String>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(String string) {

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

}

});

输出日志信息

onNext:ofType one

onNext:ofType two

onNext:ofType three

onNext:ofType four

onCompleted:ofType

当然除了过滤基本类型的数据，也可以过滤自定义类型数据。

# First

如果我们只对Observable发射的第一项数据，或者满足某个条件的第一项数据感兴趣，则可以使用First操作符。

Observable.just(10, 11, 12, 13).first().subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

上面日志只打印一个值10，当然我们也可以给first传一个参数Fun1，指定一个条件如下

Observable.just(10, 11, 12, 13).first(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer > 12;

}

}).subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

此时输出的信息就是满足integer > 12的第一项数据13。

# firstOrDefault

该操作符是first操作符的变形。主要是在没有发射任何数据时发射一个你在参数中指定的默认值。如下，它有有两个重载方法。

Observable.just(11,12,13).firstOrDefault(10).subscribe(new Action1<Object>() {

@Override

public void call(Object o) {

Log.e(TAG, o.toString());

}

});

如果写成上面的代码，这个执行会和first效果一样。因为没有发射数据的时候才用到默认值，那么我们将上面代码更改如下，使用empty创建一个不发射任何数据但是正常终止的Observable。

Observable.empty().firstOrDefault(10).subscribe(new Action1<Object>() {

@Override

public void call(Object o) {

Log.e(TAG, o.toString());

}

});

发现此时输出了数据10.该操作符还提供了两个参数的重载方法firstOrDefault(T defaultValue, Func1<? super T, Boolean> predicate)。我们可以增加一个条件。如下示例

Observable.just(10，13,16).firstOrDefault(15, new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer>20;

}

}).subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

此时数据源10,13,16都不满足大于20,则此时将输出默认值15，如果我们将数据源数据增加一个值22.那么此时将不再输出默认值，而是输出22。

# takeFirst

该操作符与first操作符的区别就是如果原始Observable没有发射任何满足条件的数据，first会抛出一个NoSuchElementException直接执行onError()，而takeFist会返回一个空的Observable（不调用onNext()但是会调用onCompleted）  
如下面下面示例代码

Observable.just(10,11).filter(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer>20;

}

}).first().subscribe(new Subscriber<Object>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Object o) {

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

}

});

执行后输出的信息如下

onError: java.util.NoSuchElementException: Sequence contains no elements

若此时用takeFirst

Observable.just(10,11).takeFirst(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

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

return integer>30;

}

}).subscribe(new Subscriber<Object>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Object o) {

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

}

});

发现此时不会出现异常，而是执行了onCompleted（）。

# single

如果原始Observable在完成之前不是正好发射一次数据，它会抛出一个NoSuchElementException，白话可以理解为发送数据是一项的话输出此项的值，若是多个数据则抛出异常执行onError()方法。  
如下代码

Observable.just(10, 11, 12, 13).single().subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError"+e.toString());

}

@Override

public void onNext(Integer integer) {

Log.e(TAG, integer);

}

});

输出信息

onError: java.util.NoSuchElementException: Sequence contains no elements

如果将上述代码做下简单更改

Observable.just(10, 11, 12, 13).filter(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer > 12;

}

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

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted");

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError"+e.toString());

}

@Override

public void onNext(Integer integer) {

Log.e(TAG, integer);

}

});

此时会输出数据13,因为此时通过filter后就只有一条数据。single也有singleOrDefault(T)和singleOrDefault(T,Func1)两个变体，具体可以自己代码测试区别。

# Last

该操作符与first意义相反，若我们只对Observable发射的最后一项数据，或者满足某个条件的最后一项数据感兴趣时使用该操作符。  
示例代码

Observable.just(10, 11, 12, 13).last().subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

执行后输出13.它有一个重载方法可以指定条件，获取满足条件的最后一项数据的。将上面代码修改如下

Observable.just(10, 11, 12, 13).last(new Func1<Integer, Boolean>() {

@Override

public Boolean call(Integer integer) {

return integer < 12;

}

}).subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

此时最终输出数据就是11.该操作符和first一样也有几种变体，如lastOrDefault，TakeLast，具体效果可自己测试。

# Skip

该操作符是跳过之前的前几项数据，然后再发射数据。

Observable.range(1, 10).skip(6).subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

输出日志信息

call: 7call: 8call: 9call: 10

skip还有两个重载方法.skip(long time, TimeUnit unit)默认是在computation调度器上执行，如果要有更新UI操作需要通过observeOn方法指定为AndroidSchedulers.mainThread()，当然还有一个重载方法skip(long time, TimeUnit unit, Scheduler scheduler)可以指定调度器。注意的一点是这两个重载方法的第一个参数不是跳过的数据数量，指的是时间。

Observable.interval(500, TimeUnit.MILLISECONDS)

.skip(2, TimeUnit.SECONDS)

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<Long>() {

@Override

public void onCompleted() {

}

@Override

public void onError(Throwable e) {

}

@Override

public void onNext(Long aLong) {

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

if (aLong > 10) {

this.unsubscribe();

}

}

});

如上代码，通过interval每隔500毫秒产生一个数据，通过skip设定跳过时间为2秒。并且当数据大于10时解除订阅。

# skipLast

正好和skip 相反，忽略最后产生的n个数据项

Observable.range(1, 10).skipLast(6).subscribe(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

});

输出日志信息

call: 1call: 2call: 3call: 4

# Take

Take操作符可以修改Observable的行为，只返回前面的N项数据，然后发射完成通知，忽略剩余的数据。

Observable.range(1,8)

.take(4)

.subscribe(new Subscriber<Integer>() {

@Override

public void onNext(Integer item) {

Log.e(TAG, "Next: " + item);

}

@Override

public void onError(Throwable error) {

Log.e(TAG, "Error: " + error.getMessage());

}

@Override

public void onCompleted() {

Log.e(TAG, "complete.");

}

});

输出日志信息

Next: 1Next: 2Next: 3Next: 4

complete

take和skip一样也有其它两个重载方法take(long time, TimeUnit unit)，take(long time, TimeUnit unit, Scheduler scheduler)，默认在computation调度器上执行。  
take还有变体操作符TakeLast，takeLastBuffer具体执行效果可自行代码。

# Debounce

该操作符指的是过了一段指定的时间还没发射数据时才发射一个数据，听着可能有点绕。你可以理解对源Observable间隔期产生的结果进行过滤，如果在这个规定的间隔期内没有别的结果产生，则将这个结果提交给订阅者，否则忽略该结果，原理有点像光学防抖  
上代码

Integer[] ints = {1, 2, 3, 4, 5, 6, 7, 8, 9};

Observable<String> observable = Observable.from(ints).flatMap(new Func1<Integer, Observable<String>>() {

@Override

public Observable<String> call(Integer integer) {

try {

Thread.currentThread().sleep(200 \* integer);

} catch (InterruptedException e) {

e.printStackTrace();

}

return Observable.just(integer + "");

}

});

observable.subscribeOn(Schedulers.newThread())

.debounce(1, TimeUnit.SECONDS)

.observeOn(AndroidSchedulers.mainThread())

.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);

}

});

输出信息

onNext: 4

onNext: 5

onNext: 6

onNext: 7

onNext: 8

onNext: 9

onCompleted:

这个输出数据不一定一样，有可能从5开始。

# Distinct

这个比较好理解，它就是过滤掉重复的数据，只允许还没有发射过的数据项通过。  
示例代码

Observable.just(0, 0, 6, 4, 2, 8, 2, 1, 9, 0)

.distinct()

.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Integer integer) {

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

}

});

输出日志信息

onNext:Distinct 0

onNext:Distinct 6

onNext:Distinct 4

onNext:Distinct 2

onNext:Distinct 8

onNext:Distinct 1

onNext:Distinct 9

onCompleted:Distinct

# ElementAt

该操作符获取原始Observable发射的数据序列指定索引位置的数据项，然后当做自己的唯一数据发射。给它传递一个基于0的索引值，它会发射原始Observable数据序列对应索引位置的值，如果你传递给elementAt的值为4，那么它会发射第5项的数据。如下示例代码

Observable.just(0, 0, 6, 4, 2, 8, 2, 1, 9, 0)

.elementAt(4)

.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Integer integer) {

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

}

![detail13.jpg](http://upload-images.jianshu.io/upload\_images/2764996-b8763b15e8db7adf.jpg?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240)

});

输出日志信息

onNext:ElementAt 2

onCompleted:ElementAt

# IgnoreElements

操作符抑制原始Observable发射的所有数据，只允许它的终止通知（onError或onCompleted）通过，使用该操作符onNext()方法不会执行。

Observable.just(1, 2, 3).ignoreElements().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");

}

});

执行后只会输出onCompleted。这个操作符效果就如同empty（）方法创建一个空的Observable,只会执行onCompleted()方法，不同的是ignoreElements是对数据源的处理，而empty（）是创建Observable。