# **RxJava操作符系列五**

# Delay

该操作符让原始Observable在发射每项数据之前都暂停一段指定的时间。它接受一个定义时长的参数（包括long型数据和单位）。每当原始Observable发射一项数据，delay就启动一个定时器，当定时器过了给定的时间段时，delay返回的Observable发射相同的数据项。他默认是在computation调度器上执行，当然也有重载方法可以指定调度器，若发射数据后有更新UI操作需将调度器指定AndroidSchedulers.mainThread()。（注意重载方法delay(Fun1)，delay(Fun0,Fun1)是默认不在任何特定的调度器上执行）  
示例代码

Observable.create(new Observable.OnSubscribe<Integer>() {

@Override

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

Log.e(TAG, "call: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date()));

subscriber.onNext(1);

subscriber.onNext(2);

subscriber.onNext(3);

subscriber.onNext(4);

subscriber.onCompleted();

}

}).delay(2,TimeUnit.SECONDS)

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date()));

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date())+e.toString());

}

@Override

public void onNext(Integer integer) {

tv1.append("\n"+new SimpleDateFormat("yyyy/MM/ddHH:MM:ss").format(new Date())+" "+integer);

Log.e(TAG, "onNext: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date())+integer);

}

});

输出日志信息

call: 2016/12/17 20:12:07

onNext: 2016/12/17 20:12:091

onNext: 2016/12/17 20:12:092

onNext: 2016/12/17 20:12:093

onNext: 2016/12/17 20:12:094

onCompleted: 2016/12/17 20:12:09

为了让你看到延迟效果，我把call和onNext（）回调的时间也打印出来，发送最终数据是延迟两秒发送的。

# delaySubscription

该操作符也是delay的一种实现，它和dealy的区别是dealy是延迟数据的发送，而此操作符是延迟数据的注册，指定延迟时间的重载方法是执行在computation调度器的。为了方便观察延迟注册效果，创建Observable变量。如下示例代码

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

@Override

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

Log.e(TAG, "call: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date()));

subscriber.onNext(1);

subscriber.onNext(2);

subscriber.onNext(3);

subscriber.onNext(4);

subscriber.onCompleted();

}

});

Log.e(TAG, "call11: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date()));

observable.delaySubscription(2,TimeUnit.SECONDS)

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

Log.e(TAG, "onCompleted: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date()));

}

@Override

public void onError(Throwable e) {

Log.e(TAG, "onError: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date())+e.toString());

}

@Override

public void onNext(Integer integer) {

tv1.append("\n"+new SimpleDateFormat("yyyy/MM/ddHH:MM:ss").format(new Date())+" "+integer);

Log.e(TAG, "onNext: "+new SimpleDateFormat("yyyy/MM/dd HH:MM:ss").format(new Date())+integer);

}

});

输出日志信息

call11: 2016/12/17 20:12:43

call: 2016/12/17 20:12:45

onNext: 2016/12/17 20:12:451

onNext: 2016/12/17 20:12:452

onNext: 2016/12/17 20:12:453

onNext: 2016/12/17 20:12:454

onCompleted: 2016/12/17 20:12:45

# Do

对于do系列操作符理解比较容易，他相当于给Observable执行周期的关键节点添加回调。当Observable执行到这个阶段的时候，这些回调就会被触发。在Rxjava do系列操作符有多个，如doOnNext，doOnSubscribe，doOnUnsubscribe，doOnCompleted，doOnError，doOnTerminate和doOnEach。  
当Observable每发送一个数据时，doOnNext会被首先调用，然后再onNext。若发射中途出现异常doOnError会被调用，然后onError。若数据正常发送完毕doOnCompleted会被触发，然后执行onCompleted。当订阅或者解除订阅doOnSubscribe，doOnUnsubscribe会被执行。  
示例代码

Observable.just(1, 2, 3)

.doOnNext(new Action1<Integer>() {

@Override

public void call(Integer integer) {

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

}

})

.doOnError(new Action1<Throwable>() {

@Override

public void call(Throwable throwable) {

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

}

})

.doOnCompleted(new Action0() {

@Override

public void call() {

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

}

})

.doOnSubscribe(new Action0() {

@Override

public void call() {

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

}

})

.doOnUnsubscribe(new Action0() {

@Override

public void call() {

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

}

})

.doOnTerminate(new Action0() {

@Override

public void call() {

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

}

})

.doAfterTerminate(new Action0() {

@Override

public void call() {

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

}

})

.subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Integer integer) {

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

}

});

输出日志信息

12-17 23:13:56.151 29946-29946/com.example.xh E/RxJava: doOnSubscribe: 12-17 23:13:56.151 29946-29946/com.example.xh E/RxJava: doOnNext: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: onNext1: 112-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: doOnNext: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: onNext1: 212-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: doOnNext: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: onNext1: 312-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: doOnCompleted: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: doOnTerminate: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: onCompleted1: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: doOnUnsubscribe: 12-17 23:13:56.155 29946-29946/com.example.xh E/RxJava: doAfterTerminate:

对于doOnEach操作符，他接收的是一个Observable参数，相当于doOnNext，doOnError，doOnCompleted综合体，如下示例代码

Observable.just(1,2,3)

.doOnEach(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);

}

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

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Integer integer) {

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

}

});

输出日志信息

onNext: 1

onNext1: 1

onNext: 2

onNext1: 2

onNext: 3

onNext1: 3

onCompleted:

onCompleted1:

# SubscribeOn/ObserveOn

该操作符指定Observable在一个特定的调度器上发送通知给观察者 (调用观察者的onNext, onCompleted, onError方法)，当遇到一个异常时ObserveOn会立即向前传递这个onError终止通知，它不会等待慢速消费的Observable接受任何之前它已经收到但还没有发射的数据项。这可能意味着onError通知会跳到（并吞掉）原始Observable发射的数据项前面。  
SubscribeOn操作符的作用类似，但它是用于指定Observable本身在特定的调度器上执行，它同样会在那个调度器上给观察者发通知。改操作符只能指定一次，如果指定多次则以第一次为准。而observeOn可以指定多次，每次指定会在observeOn下一句代码处生效。  
示例代码

stringBuffer = new StringBuffer();

Observable.create(new Observable.OnSubscribe<Drawable>() {

@Override

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

//不能执行耗时操作，及更新ui

stringBuffer.append("\n" + "开始发送事件" + Thread.currentThread().getName() + "\n");

Drawable drawable = getResources().getDrawable(R.mipmap.dir);

subscriber.onNext(drawable);

subscriber.onCompleted();

}

})

//指定创建Observable在io中

.subscribeOn(Schedulers.io())

//由于map中做耗时操作，通过Observable指定发射数据在新的线程

.observeOn(Schedulers.newThread())

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

@Override

public ImageView call(Drawable drawable) {

ImageView imageView = new ImageView(getActivity());

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

imageView.setLayoutParams(params);

imageView.setImageDrawable(drawable);

return imageView;

}

})

//操作UI，需要指定在主线程

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Action1<ImageView>() {

@Override

public void call(ImageView imageView) {

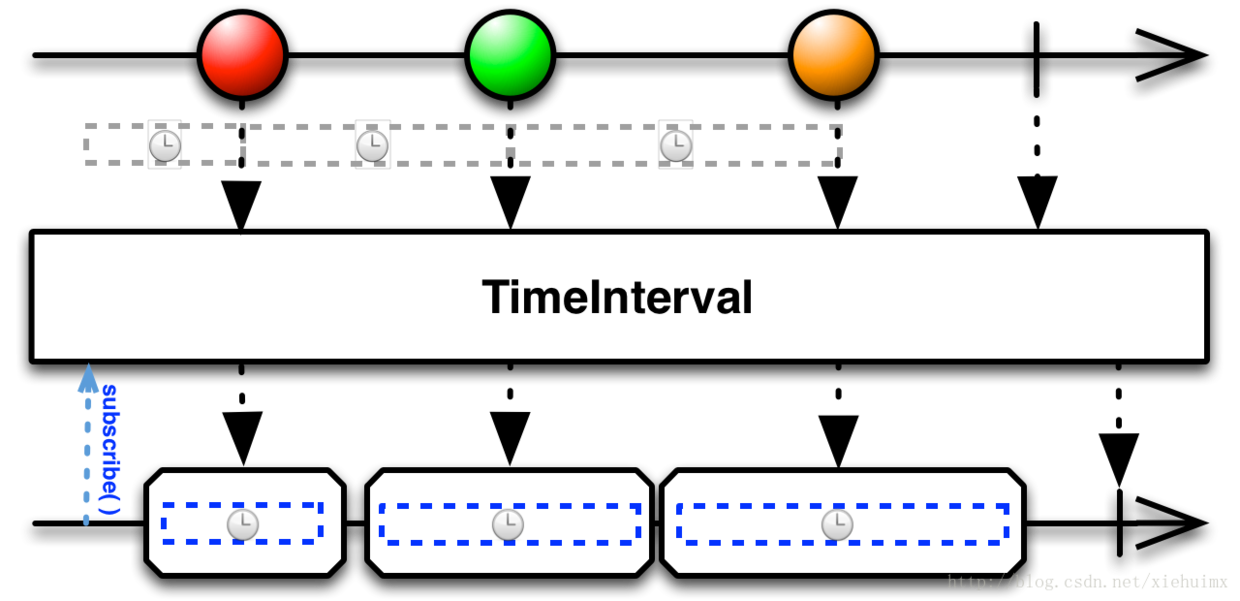
tv.append(stringBuffer.toString() + "接收信息事件" + Thread.currentThread().getName());

layout.addView(imageView);

}

});

# TimeInterval



这里写图片描述

这个操作符通过这张图能更好的理解，这个操作符将原始Observable转换为另一个Obserervable，后者发射一个标志替换前者的数据项，这个标志表示前者的两个连续发射物之间流逝的时间长度。新的Observable的第一个发射物表示的是在观察者订阅原始Observable到原始Observable发射它的第一项数据之间流逝的时间长度。不存在与原始Observable发射最后一项数据和发射onCompleted通知之间时长对应的发射物。

Observable.interval(1,TimeUnit.SECONDS)

.filter(new Func1<Long, Boolean>() {

@Override

public Boolean call(Long aLong) {

return aLong<5;

}

})

.timeInterval()

.observeOn(AndroidSchedulers.mainThread())

.subscribe(new Subscriber<TimeInterval<Long>>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(TimeInterval<Long> longTimeInterval) {

Log.e(TAG, "onNext: value:"+longTimeInterval.getValue()+"getIntervalInMilliseconds"+longTimeInterval.getIntervalInMilliseconds());

}

});

输出日志信息

onNext: value:0getIntervalInMilliseconds1002

onNext: value:1getIntervalInMilliseconds999

onNext: value:2getIntervalInMilliseconds999

onNext: value:3getIntervalInMilliseconds1000

onNext: value:4getIntervalInMilliseconds1001

通过日志发现，返回的TimeInterval类型数据，包含时间间隔和值。

# Timestamp

该操作符和TimeInterval一样最终发射的都是TimeInterval类型数据。但是不同的是，改操作符发射数据每一项包含数据的原始发射时间（TimeInterval是时间间隔）  
示例代码

Observable.just(1,2,3,4).timestamp().subscribe(new Action1<Timestamped<Integer>>() {

@Override

public void call(Timestamped<Integer> integerTimestamped) {

SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd-HH:mm:ss");

Log.e(TAG, "value: " + integerTimestamped.getValue() + " time: "+sdf.format(new Date(integerTimestamped.getTimestampMillis())) );

}

});

输出日志信息

value: 1 time: 2016-12-17-23:33:47

value: 2 time: 2016-12-17-23:33:47

value: 3 time: 2016-12-17-23:33:47

value: 4 time: 2016-12-17-23:33:47

# Timeout

如果原始Observable过了指定的一段时间没有发射任何数据，Timeout操作符会以一个onError通知终止这个Observable。

Observable.create(new Observable.OnSubscribe<Integer>() {

@Override

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

try {

subscriber.onNext(1);

Thread.sleep(100);

subscriber.onNext(2);

Thread.sleep(200);

subscriber.onNext(3);

Thread.sleep(300);

subscriber.onNext(4);

Thread.sleep(400);

subscriber.onNext(5);

subscriber.onCompleted();

} catch (InterruptedException e) {

subscriber.onError(new Throwable("Error"));

e.printStackTrace();

}

}

})

//此timeout方法默认在computation调度器上执行.

.timeout(250,TimeUnit.MILLISECONDS)

.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

onNext: 2

onNext: 3

onError:

由于发送数据3后sleep(300)超过设置的时间250ms，则执行onError。timeout还有重载方法可以在超时的时候切换到一个我们指定的备用的Observable，而不是发错误通知。它也默认在computation调度器上执行。如下示例代码

Observable.create(new Observable.OnSubscribe<Integer>() {

@Override

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

try {

subscriber.onNext(1);

Thread.sleep(100);

subscriber.onNext(2);

Thread.sleep(200);

subscriber.onNext(3);

Thread.sleep(300);

subscriber.onNext(4);

Thread.sleep(400);

subscriber.onNext(5);

subscriber.onCompleted();

} catch (InterruptedException e) {

subscriber.onError(new Throwable("Error"));

e.printStackTrace();

}

}

})

.timeout(250,TimeUnit.MILLISECONDS,Observable.just(10,11))

.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

onNext: 2

onNext: 3

onNext: 10

onNext: 11

onCompleted:

该操作符还有几个重载方法如 timeout(Func1)，timeout(Func1,Observable)， timeout(Func0,Func1)， timeout(Func0,Func1,Observable)这几个操作符默认在immediate调度器上执行，具体执行效果可自行观察代码。

# To

此系列操作符的作用是将Observable转换为另一个对象或数据结构。下面介绍几个常用的to操作符。

## toList

发射多项数据的Observable会为每一项数据调用onNext方法。你可以用toList操作符改变这个行为，让Observable将多项数据组合成一个List，然后调用一次onNext方法传递整个列表，如果原始Observable没有发射任何数据就调用了onCompleted，toList返回的Observable会在调用onCompleted之前发射一个空列表。如果原始Observable调用了onError，toList返回的Observable会立即调用它的观察者的onError方法。

Observable.just(1,2,3,4,5).toList().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);

}

});

如上代码，通过toList将单个数据最终以List<Integer>的形式输出。

## ToMap

该操作符收集原始Observable发射的所有数据项到一个Map（默认是HashMap）然后发射这个Map。我们可以提供一个用于生成Map的Key的函数，还可以提供一个函数转换数据项到Map存储的值（默认数据项本身就是值）。  
示例代码

Observable.just(1,2,3,4)

.toMap(new Func1<Integer, String>() {

@Override

public String call(Integer integer) {

//生成map的key值

return "key"+integer;

}

}).subscribe(new Subscriber<Map<String, Integer>>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Map<String, Integer> integerIntegerMap) {

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

}

});

输出日志信息

onNext: {key4=4, key3=3, key2=2, key1=1}

onCompleted:

该操作符有个两个参数的构造方法可以更改发射的数据的值，如下

Observable.just(1,2,3,4)

.toMap(new Func1<Integer, String>() {

@Override

public String call(Integer integer) {

return "key" + integer;

}

}, new Func1<Integer, Integer>() {

@Override

public Integer call(Integer integer) {

return integer+10;

}

})

.subscribe(new Subscriber<Map<String, Integer>>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Map<String, Integer> integerIntegerMap) {

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

}

输出日志信息

onNext: {key4=14, key3=13, key2=12, key1=11}

onCompleted:

发现此时指定了map的key，并且更改了发射的数据值。

## toMutimap

类似于toMap，不同的是，它生成的这个Map同时还是一个ArrayList（默认是这样，你可以传递一个可选的工厂方法修改这个行为）。toMap(Func1)是将原Observable发送的数据保存到一个MAP中，并在参数函数中，设定key。但toMultimap操作符在将数据保存到MAP前，先将数据保存到Collection，而toMap操作符将数据直接保存到MAP中，并没有再包裹一层Collection。

Observable.just(1,2,3,4)

.toMultimap(new Func1<Integer, String>() {

@Override

public String call(Integer integer) {

return "key"+integer;

}

}).subscribe(new Subscriber<Map<String, Collection<Integer>>>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Map<String, Collection<Integer>> integerCollectionMap) {

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

}

});

输出日志信息

onNext: {key4=[4], key3=[3], key2=[2], key1=[1]}

onCompleted:

通过上面信息，也看的两者区别。

## toSortedList

该操作符类似于toList，区别是它可以对数据进行自然排序。如下示例

Integer[] integers = {2, 3, 6, 4, 9,2, 8};

Observable.from(integers)

.toSortedList()

.flatMap(new Func1<List<Integer>, Observable<Integer>>() {

@Override

public Observable<Integer> call(List<Integer> integer) {

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

return Observable.from(integer);

}

}).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);

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

}

});

输出日志信息

call: [2, 2, 3, 4, 6, 8, 9]

onNext: 2

onNext: 2

onNext: 3

onNext: 4

onNext: 6

onNext: 8

onNext: 9

onCompleted: