# **RxJava操作符系列四**

# Merge

该操作符可以将多个Observables的输出合并，就好像它们是一个单个的Observable一样，他可能让我们让合并的Observables发射的数据交错（顺序发生变化），在此过程中任何一个原始Observable的onError通知都会被立即传递给观察者，而且会终止合并后的Observable。

Observable observable=Observable.just(1,2);

Observable observable1=Observable.just(6,7);

Observable.merge(observable, observable1)

.observeOn(AndroidSchedulers.mainThread())

.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 s) {

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

}

});

输出日志信息

onNext: 1

onNext: 2

onNext: 6

onNext: 7

onCompleted:

如果我们此时在创建一个Obserable如下

Observable observable2 = Observable.create(new Observable.OnSubscribe<Integer>() {

@Override

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

try {

Thread.sleep(500);

subscriber.onNext(200);

subscriber.onCompleted();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

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

然后将merge(observable, observable1)更改为merge(observable2，observable, observable1)  
再次执行，发现虽然observable2是第一个参数，但是输出却是在最后一个输出。

# MergeDelayError

对于merge操作符的任何一个的Observable发射了onError通知终止了，merge操作符生成的Observable也会立即以onError通知终止。如果你想让它继续发射数据，在最后才报告错误，可以使用mergeDelayError。我们先使用merge将observable2的创建代码更改下面代码，依然执行merge(observable2，observable, observable1)。

Observable observable2 = Observable.create(new Observable.OnSubscribe<Integer>() {

@Override

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

try {

subscriber.onNext(100);

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

subscriber.onCompleted();

} catch (InterruptedException e) {

e.printStackTrace();

subscriber.onError(new Throwable("error11"));

}

}

}).subscribeOn(Schedulers.newThread()).observeOn(AndroidSchedulers.mainThread());

输出日志信息

onError: java.lang.Throwable: error

由于observable2 的异常，导致observable和 observable1被中断。  
那么如果在这种情况下依然发送数据该怎么办呢，MergeDelayError就可以达到这样的效果，看如下实现代码。**最后提醒下MergeDelayError的使用有个坑，就是subscribeOn和observeOn的调用问题，如果先mergeDelayError之后再用subscribeOn和observeOn指定调度器发现该操作符并不起作用。需要在单独创建Observable时使用**，如下示例代码

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

@Override

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

try {

subscriber.onNext(100);

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

subscriber.onCompleted();

} catch (InterruptedException e) {

e.printStackTrace();

subscriber.onError(new Throwable("error11"));

}

}

}).subscribeOn(Schedulers.newThread()).observeOn(AndroidSchedulers.mainThread());

Observable observable3=Observable.just(6,7,8,9).subscribeOn(Schedulers.newThread()).observeOn(AndroidSchedulers.mainThread());

Observable.merge(observable,observable1)

.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 s) {

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

}

});

输出日志信息

onNext: 6

onNext: 7

onNext: 8

onNext: 9

onError: java.lang.Throwable: error

# Concat

该操作符和merge操作符相似，不同之处就是该操作符按顺序一个接着一个发射多个Observables的发射物。保证顺序性。

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

@Override

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

try {

subscriber.onNext(100);

Thread.sleep(500);

subscriber.onNext(200);

subscriber.onCompleted();

} catch (InterruptedException e) {

e.printStackTrace();

subscriber.onError(new Throwable("error11"));

}

}

}).subscribeOn(Schedulers.newThread()).observeOn(AndroidSchedulers.mainThread());

Observable<Integer> observableB = Observable.range(7, 2);

Observable.concat(observableA, observableB).subscribe(new Subscriber<Integer>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Integer integer) {

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

}

});

输出日志信息

onNext: concat100

onNext: concat200

onNext: concat7

onNext: concat8

onCompleted: concat

这样就保证了顺序性，如果用merge的话，100和200应该在7和8后输出。

# Zip

该操作符返回一个Obversable，它使用这个函数按顺序结合两个或多个Observables发射的数据项，然后它发射这个函数返回的结果。它按照严格的顺序应用这个函数。它只发射与发射数据项最少的那个Observable一样多的数据，假如两个Observable数据分布为4项，5项，则最终合并是4项。如下示例代码

List<String> names = new ArrayList<>();

List<Integer> ages = new ArrayList<>();

for (int i = 0; i < 5; i++) {

names.add("张三" + i);

ages.add(20 + i);

}

ages.add(15);

Observable observable1 = Observable.from(names).subscribeOn(Schedulers.io());

Observable observable2 = Observable.from(ages).subscribeOn(Schedulers.io());

//Func2第三个参数是返回值类型

Observable.zip(observable1, observable2, new Func2<String, Integer, String>() {

@Override

public String call(String name, Integer age) {

return name + ": " + age;

}

}).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 o) {

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

}

});

输出日志信息

onNext: 张三0: 20

onNext: 张三1: 21

onNext: 张三2: 22

onNext: 张三3: 23

onNext: 张三4: 24

onCompleted:

# StartWith

该操作符作用是在一个Observable在发射数据之前先发射一个指定的数据序列。如下

Observable.range(1,3).startWith(11,12).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: 11

onNext: 12

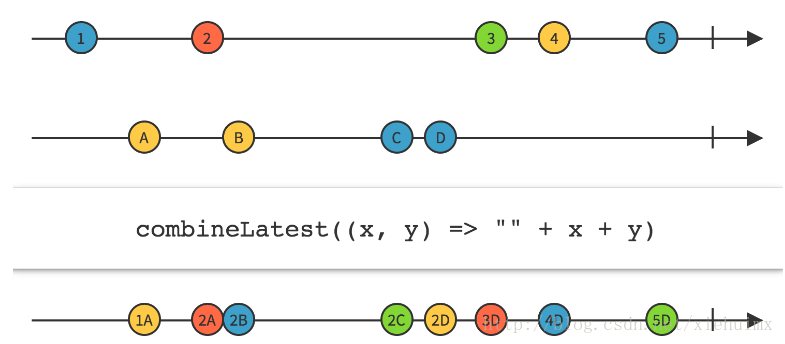
onNext: 1

onNext: 2

onNext: 3

onCompleted:

# CombineLatest



这里写图片描述

操作符行为类似于zip，但是只有当原始的Observable中的每一个都发射了一条数据时zip才发射数据。CombineLatest则在原始的Observable中任意一个发射了数据时发射一条数据。当原始Observables的任何一个发射了一条数据时，CombineLatest使用一个函数结合它们最近发射的数据，然后发射这个函数的返回值。通过上面的图应该更容易理解。  
示例代码

Observable<Integer> observableA = Observable.range(1, 4);

Observable<Integer> observableB = Observable.range(7, 5);

Observable.combineLatest(observableA, observableB, new Func2<Integer, Integer, String>() {

@Override

public String call(Integer integer, Integer integer2) {

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

return "observableA:" + integer + " observableB:" + integer2;

}

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

@Override

public void call(String s) {

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

}

});

输出日志信息

call: combineLatestcall: combineLatestobservableA:4 observableB:10call: combineLatestcall: combineLatestobservableA:4 observableB:11call: combineLatestcall: combineLatestobservableA:4 observableB:12call: combineLatestcall: combineLatestobservableA:4 observableB:13call: combineLatestcall: combineLatestobservableA:4 observableB:14

# Join

该操作符只要在另一个Observable发射的数据定义的时间窗口内，这个Observable发射了一条数据，就结合两个Observable发射的数据.例如A作为基础窗口，当A发射了数据1,2,3,4,5时，B发射了一个数据a.则此时合并数据（1，a）,（2，a）,（3，a），（4，a），（5，a），此时将窗口清楚并重新打开一个窗口循环此种操作直到数据输出完毕。  
示例代码

Observable<Integer> observableA = Observable.range(1, 2).subscribeOn(Schedulers.newThread());

Observable<Integer> observableB = Observable.range(7, 3).subscribeOn(Schedulers.newThread());

observableA.join(observableB, new Func1<Integer, Observable<Integer>>() {

@Override

public Observable<Integer> call(Integer integer) {

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

return Observable.just(integer).delay(1,TimeUnit.SECONDS);

}

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

@Override

public Observable<Integer> call(Integer integer) {

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

return Observable.just(integer).delay(1,TimeUnit.SECONDS);

}

}, new Func2<Integer, Integer, Integer>() {

@Override

public Integer call(Integer integer, Integer integer2) {

Log.e(TAG, "call:AjoinB A: " + integer + " B:" + integer2 + Thread.currentThread().getName());

return integer+integer2;

}

}).observeOn(AndroidSchedulers.mainThread())

.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, "onNext: "+integer);

}

});

输出日志信息

call: A1 RxNewThreadScheduler-4

call: A2 RxNewThreadScheduler-4

call: B7 RxNewThreadScheduler-5

call:AjoinB A: 1 B:7RxNewThreadScheduler-5

call:AjoinB A: 2 B:7RxNewThreadScheduler-5

call: B8 RxNewThreadScheduler-5

call:AjoinB A: 1 B:8RxNewThreadScheduler-5

call:AjoinB A: 2 B:8RxNewThreadScheduler-5

call: B9 RxNewThreadScheduler-5

onNext: 8

onNext: 9

onNext: 9

call:AjoinB A: 1 B:9RxNewThreadScheduler-5

call:AjoinB A: 2 B:9RxNewThreadScheduler-5

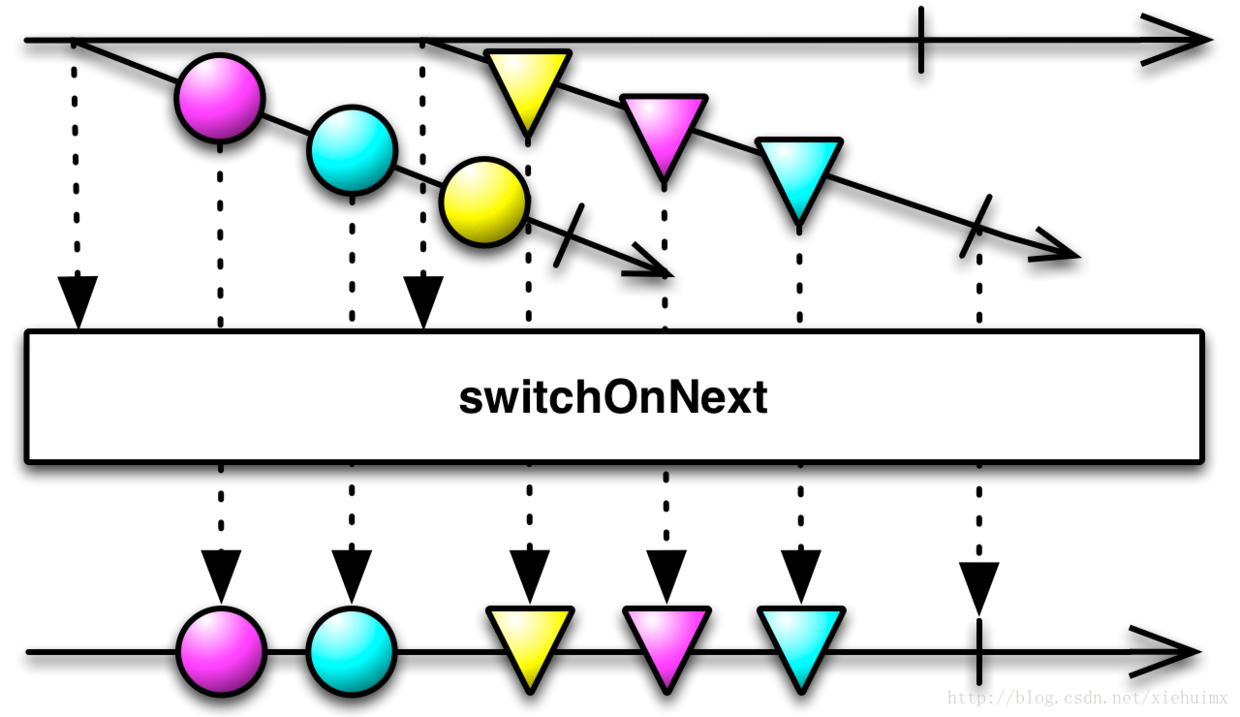
onNext: 10

onNext: 10

onNext: 11

onCompleted:

# switchOnNext



这里写图片描述

该操作符将一个发射多个Observables的Observable转换成另一个单独的Observable，后者发射那些Observables最近发射的数据项。当有新的Observable开始订阅时，会取消之前的订阅，并将数据丢弃。如下示例

Observable<Observable<Long>> observable = Observable.interval(0, 500, TimeUnit.MILLISECONDS).map(new Func1<Long, Observable<Long>>() {

@Override

public Observable<Long> call(Long aLong) {

//每隔200毫秒产生一组数据（0,10,20,30,40)

Log.e(TAG, "call1: "+aLong);

return Observable.interval(0, 200, TimeUnit.MILLISECONDS).map(new Func1<Long, Long>() {

@Override

public Long call(Long aLong) {

Log.e(TAG, "call2: "+aLong );

return aLong \* 10;

}

}).take(5);

}

}).take(2);

Observable.switchOnNext(observable).subscribe(new Subscriber<Long>() {

@Override

public void onCompleted() {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onNext(Long aLong) {

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

}

});

输出日志信息

call1: 0

call2: 0

onNext: SwitchOnNext 0

call2: 1

onNext: SwitchOnNext 10

call2: 2

onNext: SwitchOnNext 20

call1: 1

call2: 0

onNext: SwitchOnNext 0

call2: 1

onNext: SwitchOnNext 10

call2: 2

onNext: SwitchOnNext 20

call2: 3

onNext: SwitchOnNext 30

call2: 4

onNext: SwitchOnNext 40

onCompleted: SwitchOnNext