Rxjava_操作符6_辅助操作符

Delay

顾名思义，Delay操作符就是让发射数据的时机延后一段时间，这样所有的数据都会依次延后一段时间发射。
（和time的不同：延时创建和延时发射）

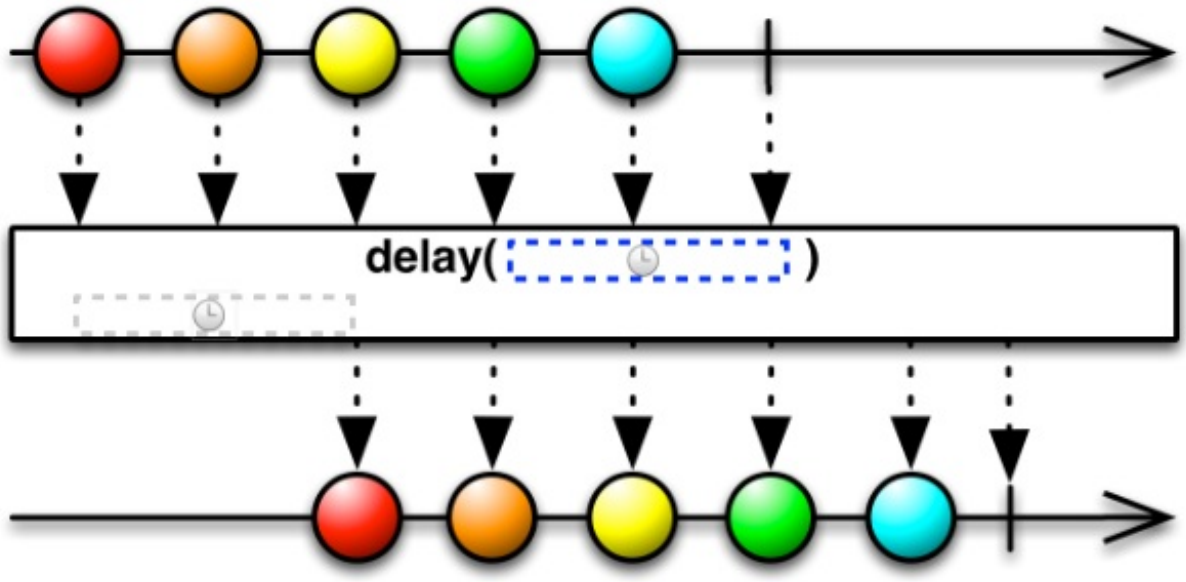
在Rxjava中将其实现为Delay和DelaySubscription。

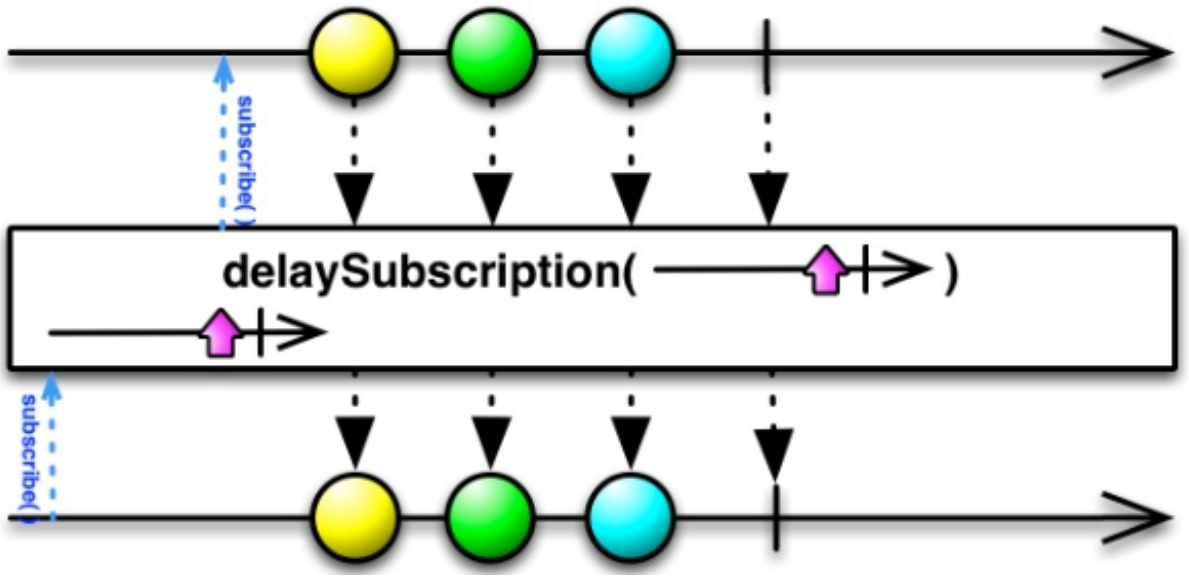
delay

`delay(Func1);` func1返回一个observable，delay监视这个observable，当这个observable中止的时候，delay发射数据
`delay(long,TimeUnit);` 延迟一段时间

DelaySubscription

和Delay的不同之处，不同之处在于Delay是延时数据的发射，而DelaySubscription是延时订阅Subscriber。





下面我们使用Delay和DelaySubscribtion操作符来延迟两个Observable数据的发射

```
private Observable<Long> delayObserver() {
        return createObserver(2).delay(2000, TimeUnit.MILLISECONDS);
    }

    private Observable<Long> delaySubscriptionObserver() {
        return createObserver(2).delaySubscription(2000, TimeUnit.MILLISECONDS)
;
    }

    private Observable<Long> createObserver(int index) {
        return Observable.create(new Observable.OnSubscribe<Long>() {
            @Override
            public void call(Subscriber<? super Long> subscriber) {
                log("subscrib:" + getCurrentTime());
                for (int i = 1; i <= index; i++) {
                    subscriber.onNext(getCurrentTime());
                    try {
                        Thread.sleep(1000);
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            }
        }).subscribeOn(Schedulers.newThread());
    }

    private long getCurrentTime() {
        return System.currentTimeMillis()/1000;
    }
```

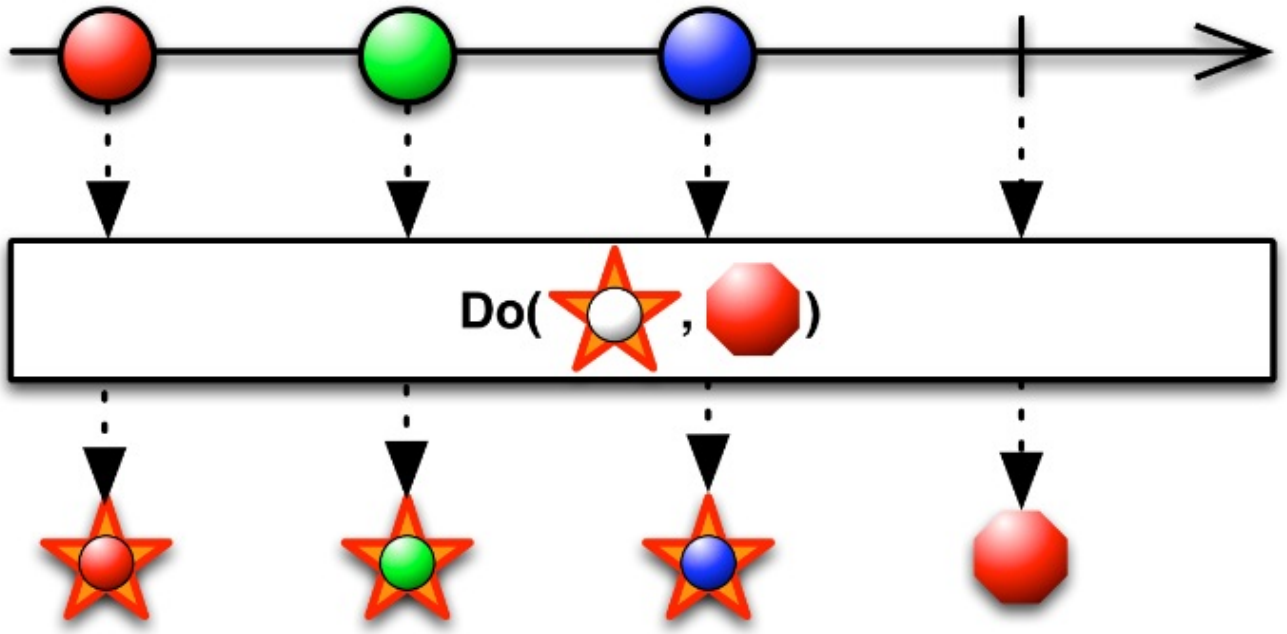
分别注册

```
mLButton.setText("delay");
        mLButton.setOnClickListener(e -> {
            log("start subscrib:" + getCurrentTime());
            delayObserver().subscribe(i -> log("delay:" + (getCurrentTime() - i
)));
        });
        mRButton.setText("delaySubscription");
        mRButton.setOnClickListener(e -> {
            log("start subscrib:" + getCurrentTime());
            delaySubscriptionObserver().subscribe(i -> log("delaySubscription:"
+ i));
        });
```

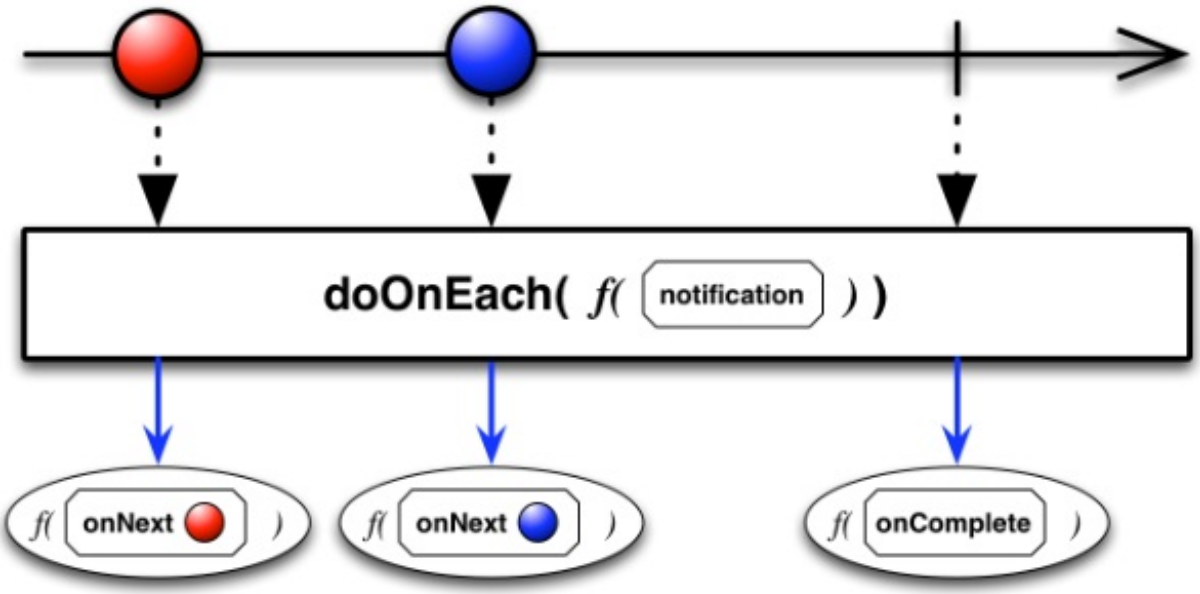
do

注册一个动作作为原始Observable生命周期事件的一种占位符
Do操作符就是给Observable的生命周期的各个阶段加上一系列的回调监听，当Observable执行到这个阶段的时候，这些回调就会被触发。在Rxjava实现了很多的doxxx操作符。

你可以注册回调，当Observable的某个事件发生时，Rx会在与Observable链关联的正常通知集合中调用它。Rx实现了多种操作符用于达到这个目的。



doOnEach

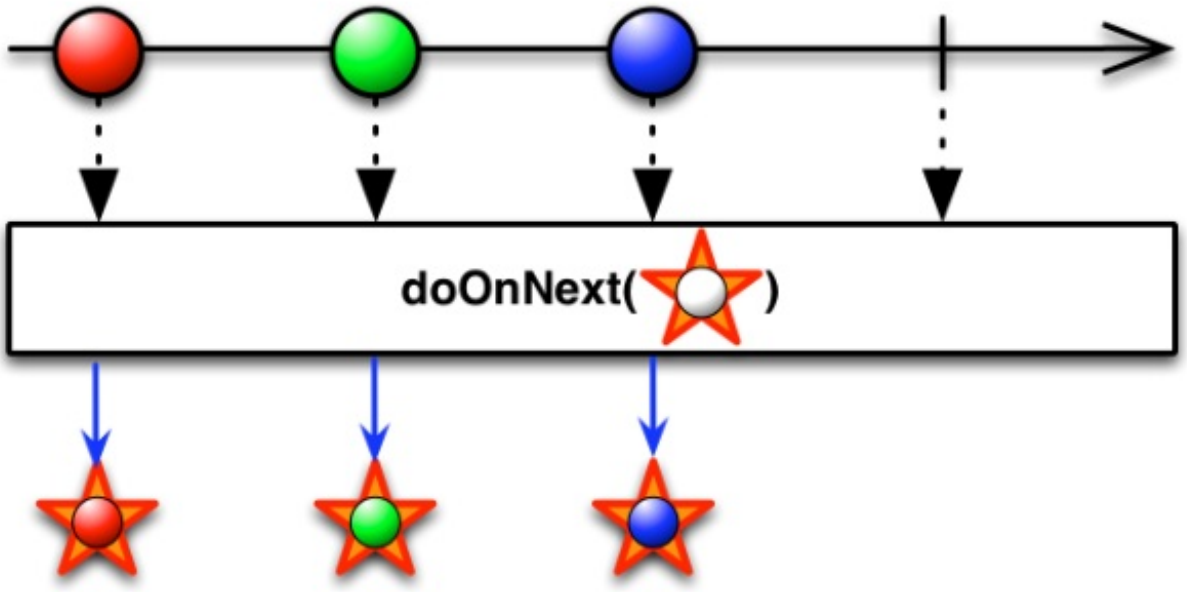


doOnEach操作符让你可以注册一个回调，它产生的Observable每发射一项数据就会调用它一次，不仅包括onNext还包括onError和onCompleted。。

你可以以Action的形式传递参数给它，这个Action接受一个onNext的变体Notification作为它的唯一参数

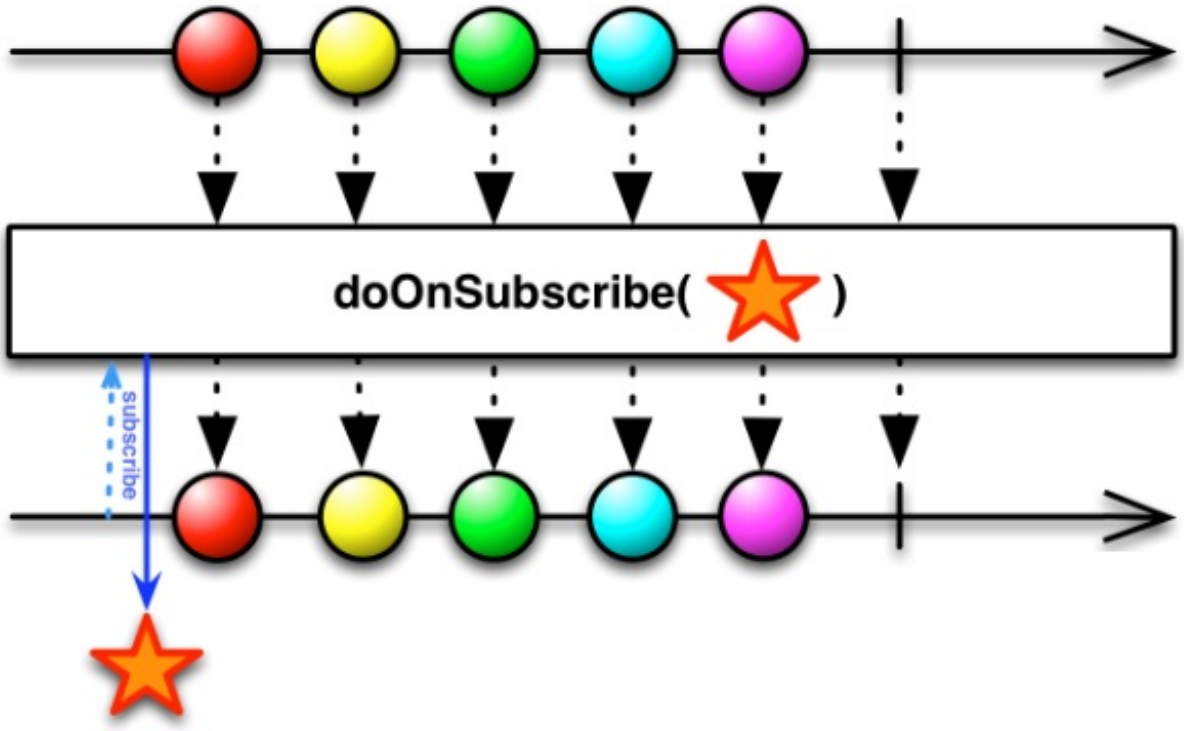
你也可以传递一个Observable给doOnEach，这个Observable的onNext会被调用，就好像它订阅了原始的Observable一样。

doOnNext



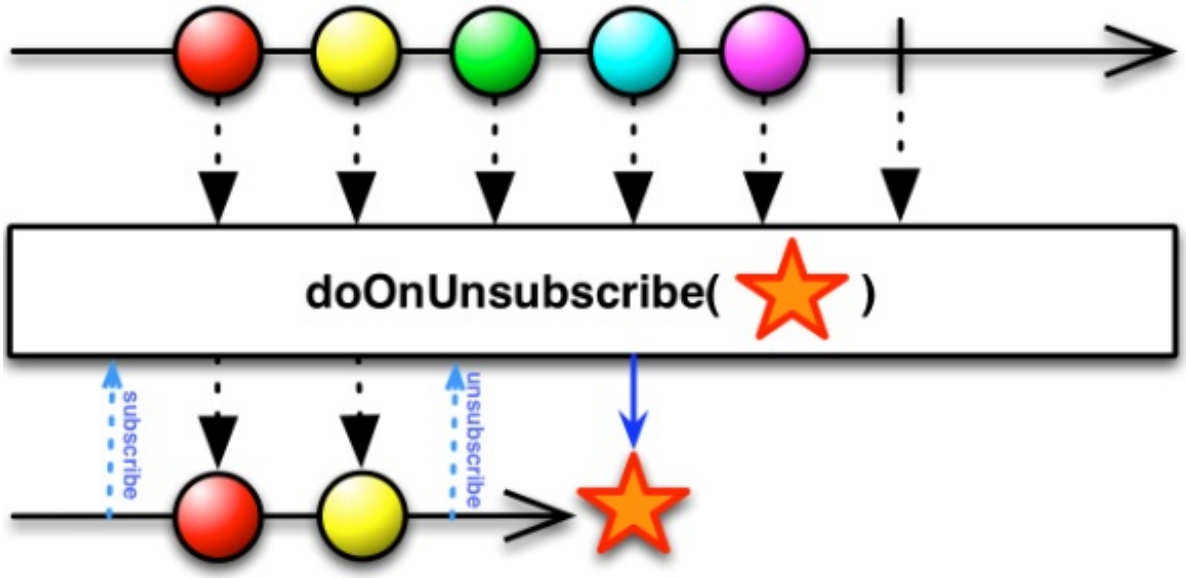
doOnNext操作符类似于doOnEach(Action1)，但是它的Action是接受一个接受发射的数据项作为参数。
DoOnNext则只有onNext的时候才会被触发。

doOnSubscribe



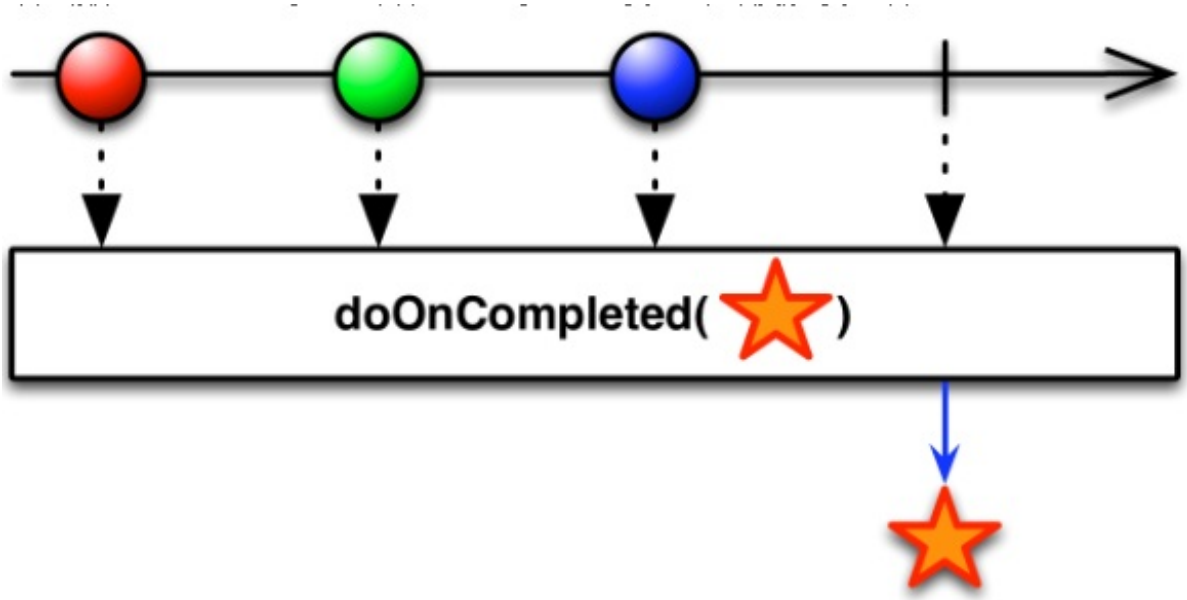
doOnSubscribe操作符注册一个动作，当观察者订阅它生成的Observable它就会被调用。

doOnUnsubscribe



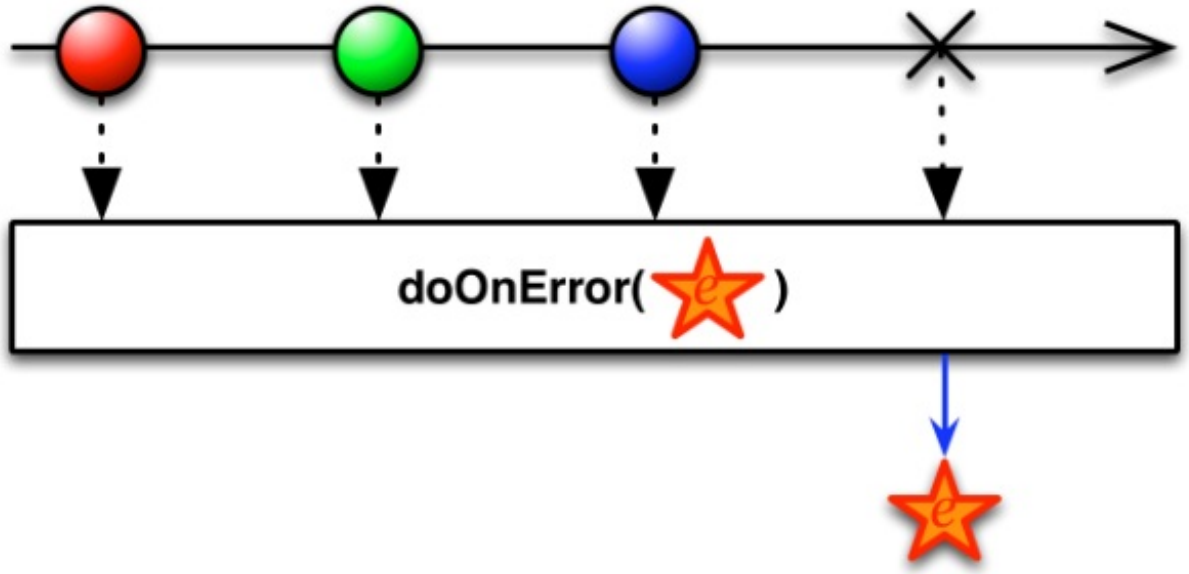
doOnUnsubscribe操作符注册一个动作，当观察者取消订阅它生成的Observable它就会被调用。

doOnCompleted



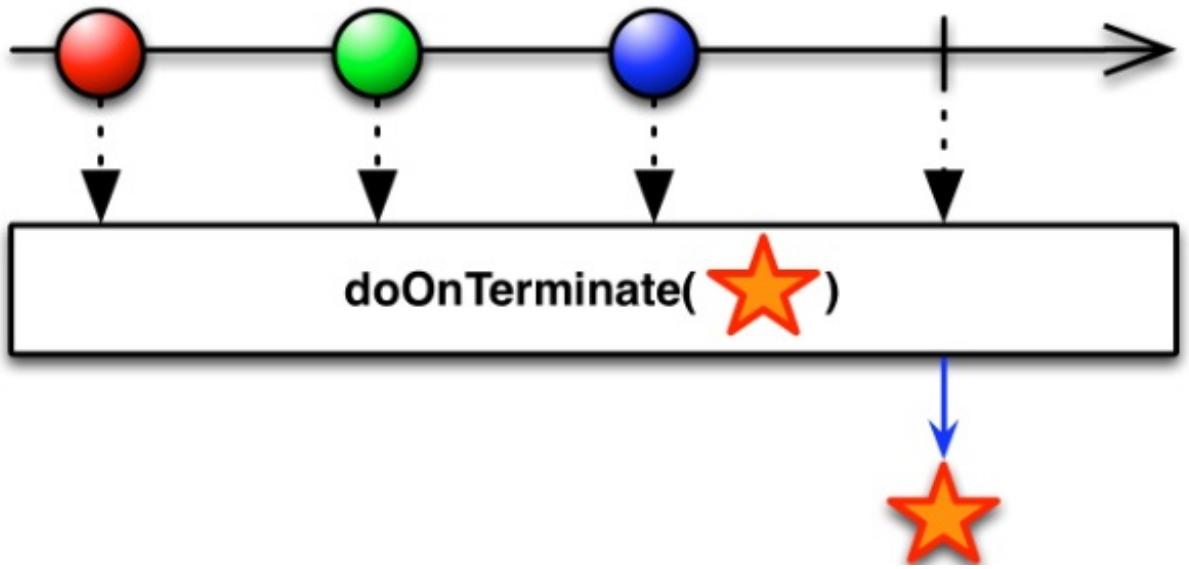
doOnCompleted 操作符注册一个动作，当它产生的Observable正常终止调用onCompleted时会被调用。

doOnError



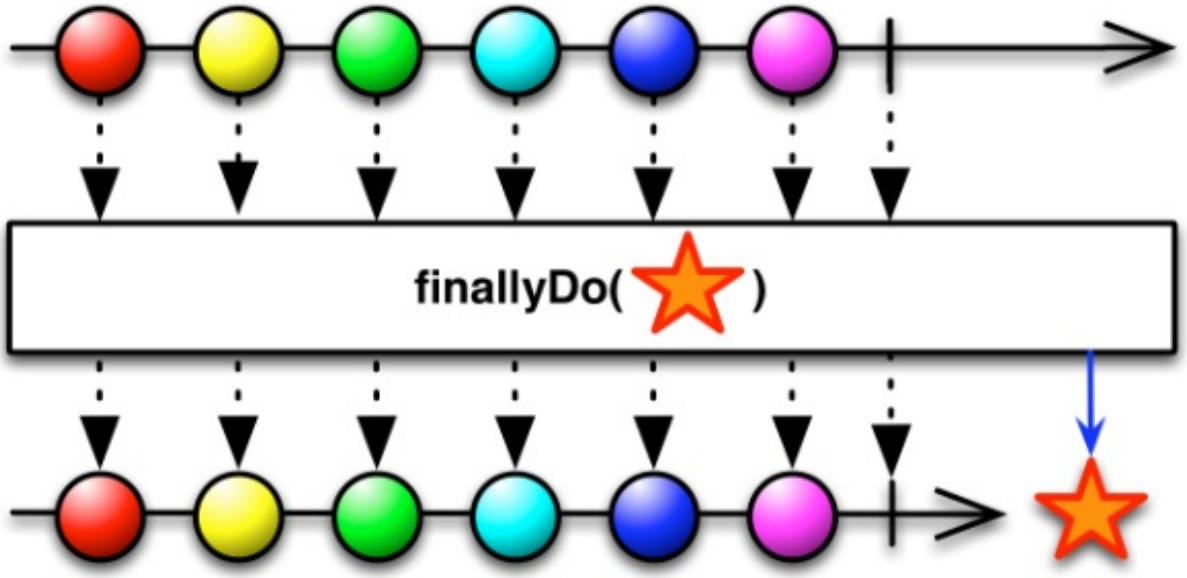
doOnError 操作符注册一个动作，当它产生的Observable异常终止调用onError时会被调用。

doOnTerminate



doOnTerminate 操作符注册一个动作，当它产生的Observable终止之前会被调用，无论是正常还是异常终止。

finallyDo



finallyDo 操作符注册一个动作，当它产生的Observable终止之后会被调用，无论是正常还是异常终止。
1.正常执行，无onError通知

```
Observable.just(new Student(1, "create - 1", 20), new Student(1, "create - 1",
20))
        .subscribeOn(Schedulers.io())
        .doOnEach(new Action1<Notification<? super Student>>() {
            @Override
            public void call(Notification<? super Student> notification) {
                Log.i(TAG, "doOnEach");
            }
        })

        .doOnNext(new Action1<Student>() {
```

```
            @Override
            public void call(Student student) {
                Log.i(TAG, "doOnNext");
            }
        })
        .doOnSubscribe(new Action0() {
            @Override
            public void call() {
                Log.i(TAG, "doOnSubscribe");
            }
        })
        .doOnUnsubscribe(new Action0() {
            @Override
            public void call() {
                Log.i(TAG, "doOnUnsubscribe");
            }
        })
        .doOnError(new Action1<Throwable>() {
            @Override
            public void call(Throwable throwable) {
                Log.i(TAG, "doOnError");
            }
        })

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

            @Override
            public void onStart() {
                super.onStart();
                Log.i(TAG, "onStart");
            }

            @Override
            public void onCompleted() {
                Log.i(TAG, "onCompleted");

            }

            @Override
            public void onError(Throwable e) {
                Log.i(TAG, "onError");
            }

            @Override
            public void onNext(Student student) {
                Log.i(TAG, "onNext");
                Log.i(TAG, "onNext - student:" + student.toString());
```

```
                mAdaStudent.addData(student);
                //unsubscribe();
            }
        });
```

2.有onError通知

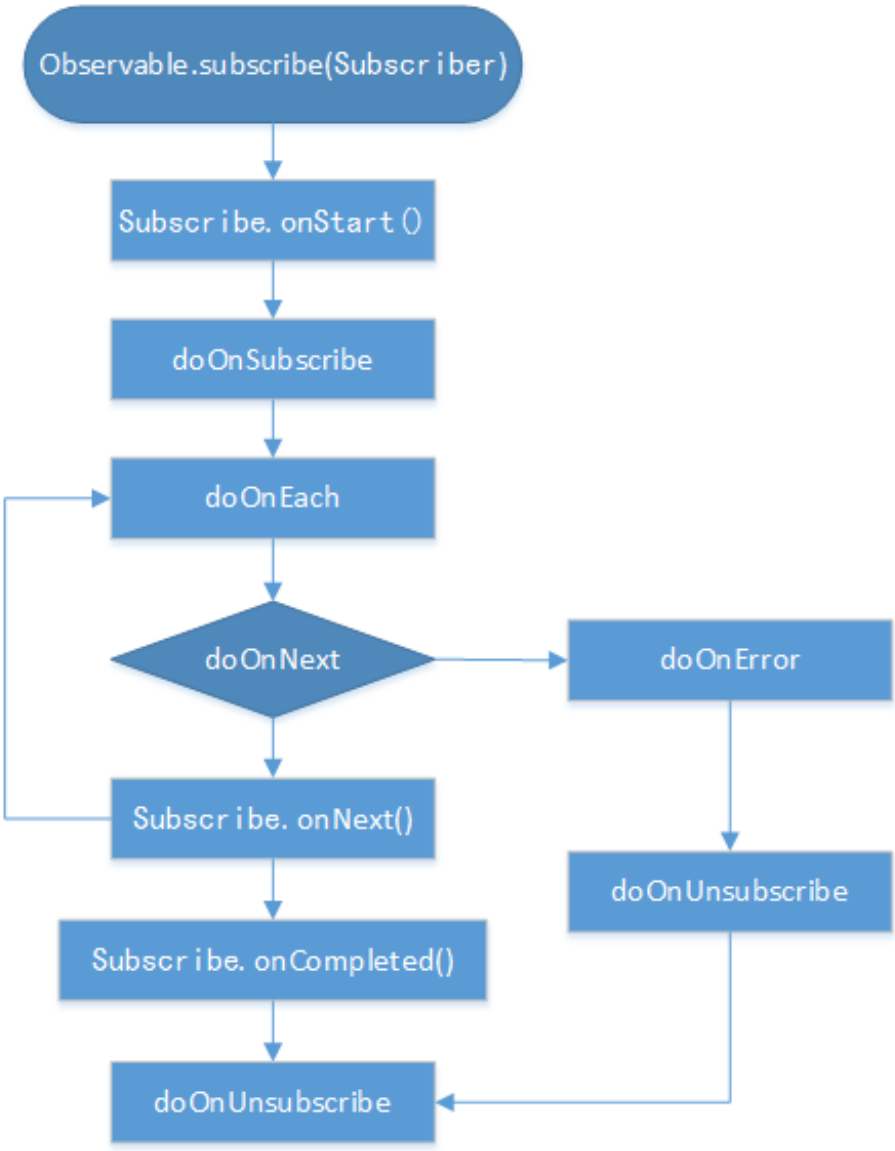
```
Observable.create(new Observable.OnSubscribe<Student>() {
    @Override
    public void call(Subscriber<? super Student> subscriber) {
        subscriber.onNext(new Student(1001, "do - 1", 20));
        subscriber.onError(new Throwable("do"));
    }
})***
```

1. doOnSubscribe

```
doOnEach
doOnNext
onNext
onNext - student:Student{id='1'name='create - 1', age=20}

doOnEach
doOnNext
doOnEach
onNext
onNext - student:Student{id='1'name='create - 1', age=20}

onCompleted
doOnUnsubscribe
```

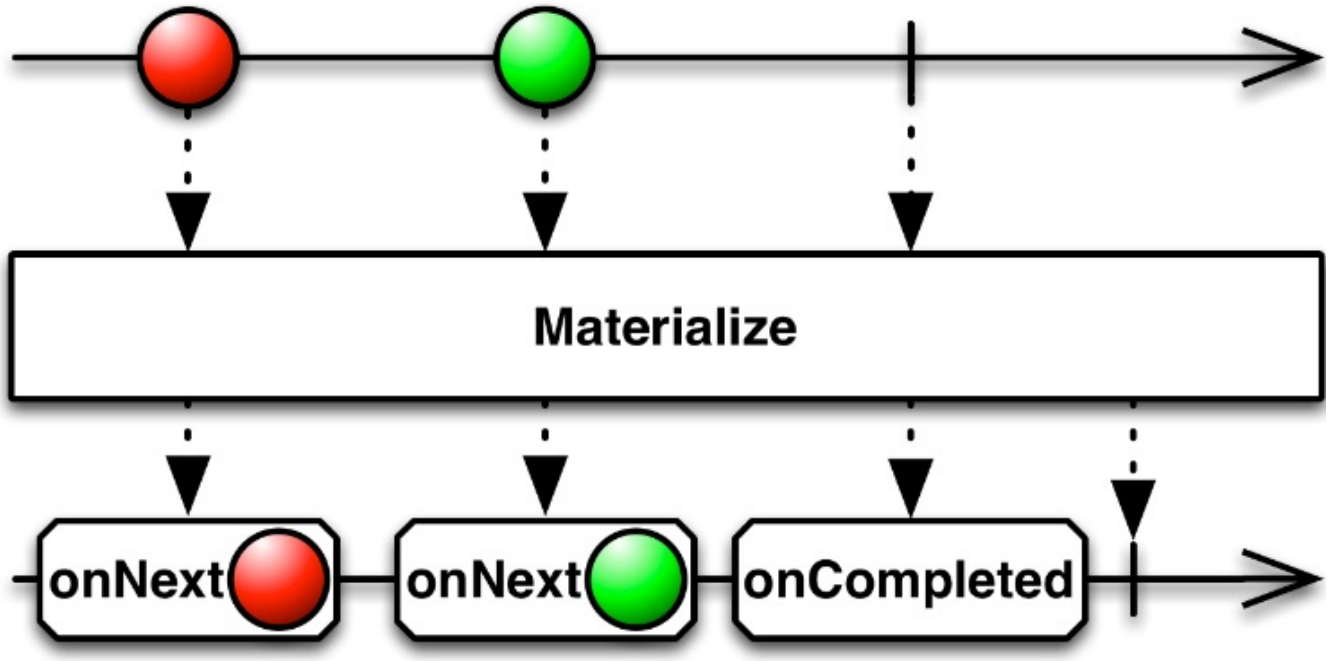
2.有onError通知
onStart
doOnSubscribe

```
doOnNext

doOnError
doOnError

doOnUnsubscribe
```



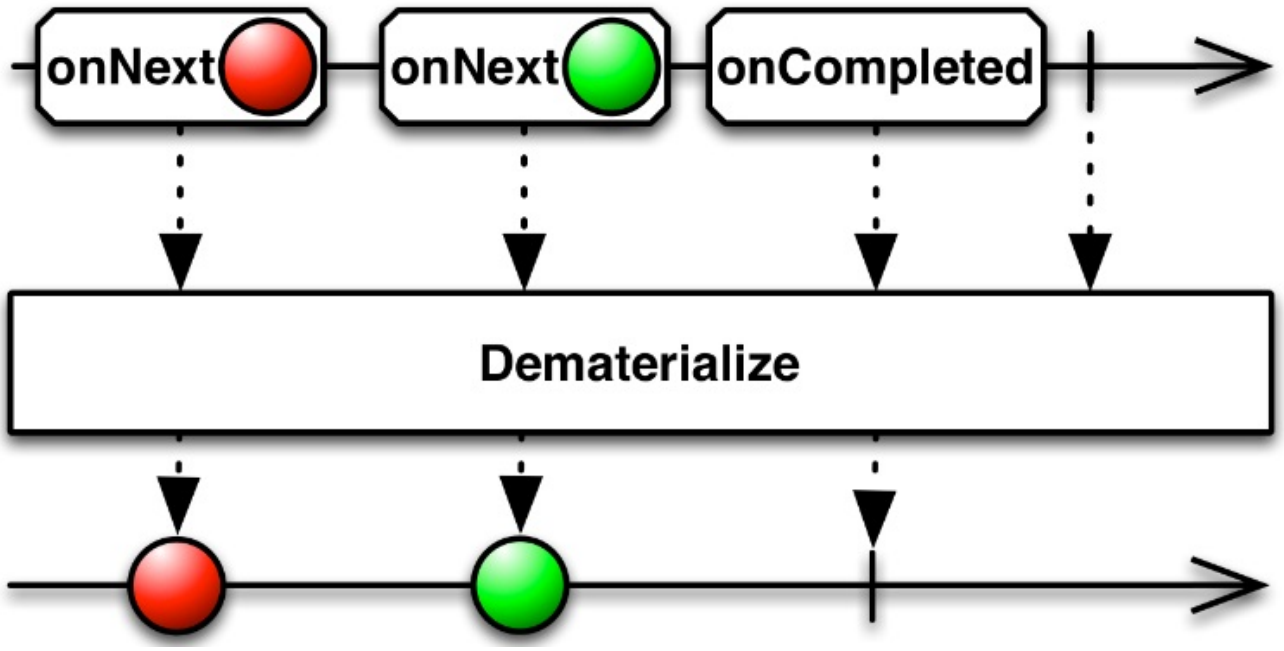
Materialize/Dematerialize

Materialize将数据项和事件通知都当做数据项发射，Dematerialize刚好相反。



一个合法的有限的Obversable将调用它的观察者的onNext方法零次或多次，然后调用观察者的onCompleted或onError正好一次。Materialize操作符将这一系列调用，包括原来的onNext通知和终止通知onCompleted或onError都转换为一个Observable发射的数据序列。

RxJava的materialize将来自原始Observable的通知转换为Notification对象，然后它返回的Observable会发射这些数据。



Dematerialize操作符是Materialize的逆向过程，它将Materialize转换的结果还原成它原本的形式。

dematerialize反转这个过程，将原始Observable发射的Notification对象还原成Observable的通知。

两者默认不在任何特定的调度器 (Scheduler) 上执行

```
private Observable<Notification<Integer>> meterializeObserver() {
        return Observable.just(1, 2, 3).materialize();
    }

    private Observable<Integer> deMeterializeObserver() {
        return eterializeObserver().dematerialize();
    }
```

```
mLButton.setText("meterialize");
mLButton.setOnClickListener(e -> meterializeObserver().subscribe(i -> log("mete
rialize:" + i.getValue() + " type" + i.getKind())));
mRButton.setText("deMeterialize");
mRButton.setOnClickListener(e -> deMeterializeObserver().subscribe(i->log("deMe
terialize:"+i)));
```

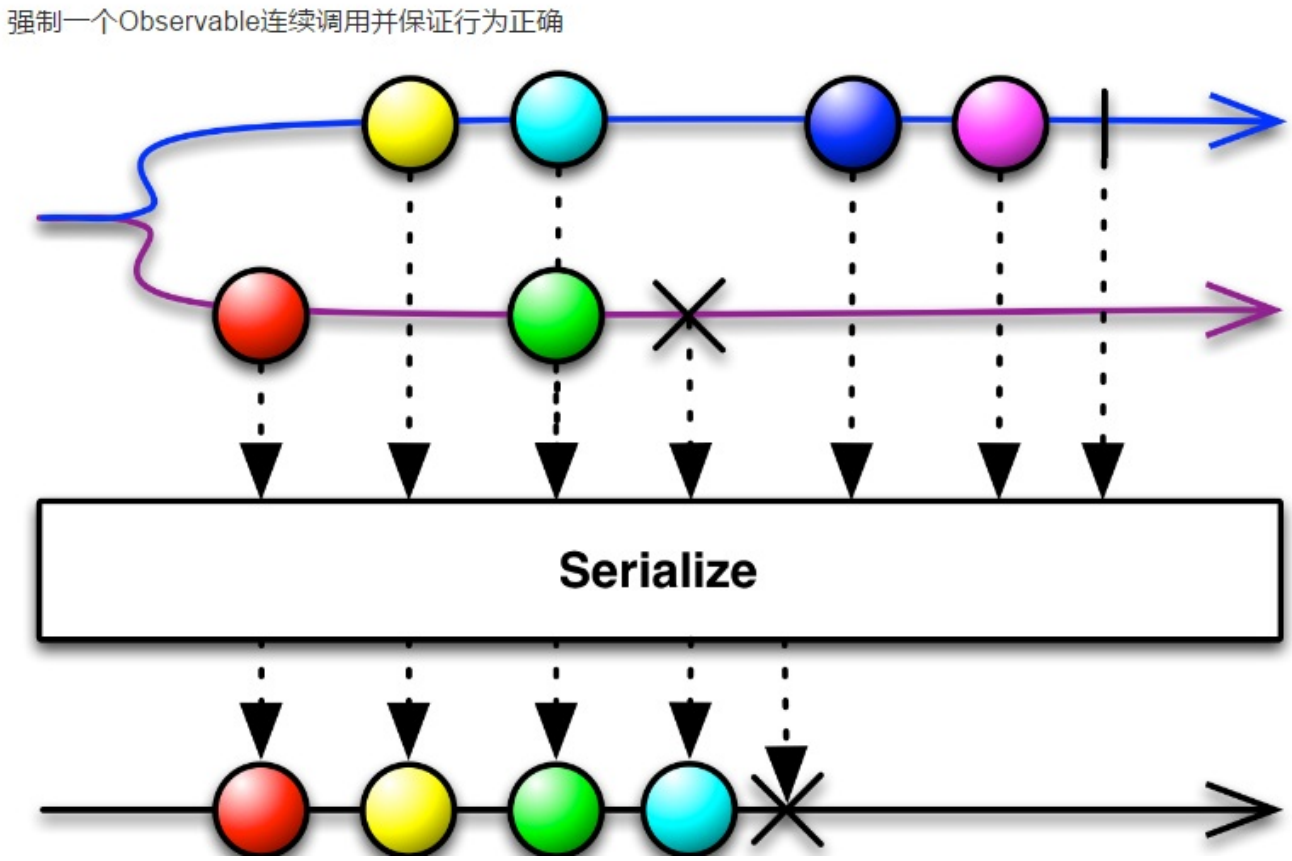
ObserveOn

RxJava中，要指定Observable应该在哪个调度器上调用观察者的onNext, onCompleted, onError方法，你需要使用observeOn操作符，传递给它一个合适的Scheduler。

相信大家一定都遇到过不能在主线程修改UI的问题，所以不得不使用Handler、AsyncTask等来更新UI界面。使用SubscribOn和ObserverOn操作符，各种线程的问题都将变得十分地简单。

SubscribOn用来指定Observable在哪个线程上运行，我们可以指定在IO线程上运行也可以让其新开一个线程运行，当然也可以在当前线程上运行。一般来讲会指定在各种后台线程而不是主线程上运行，就如同AsyncTask的doInBackground一样。

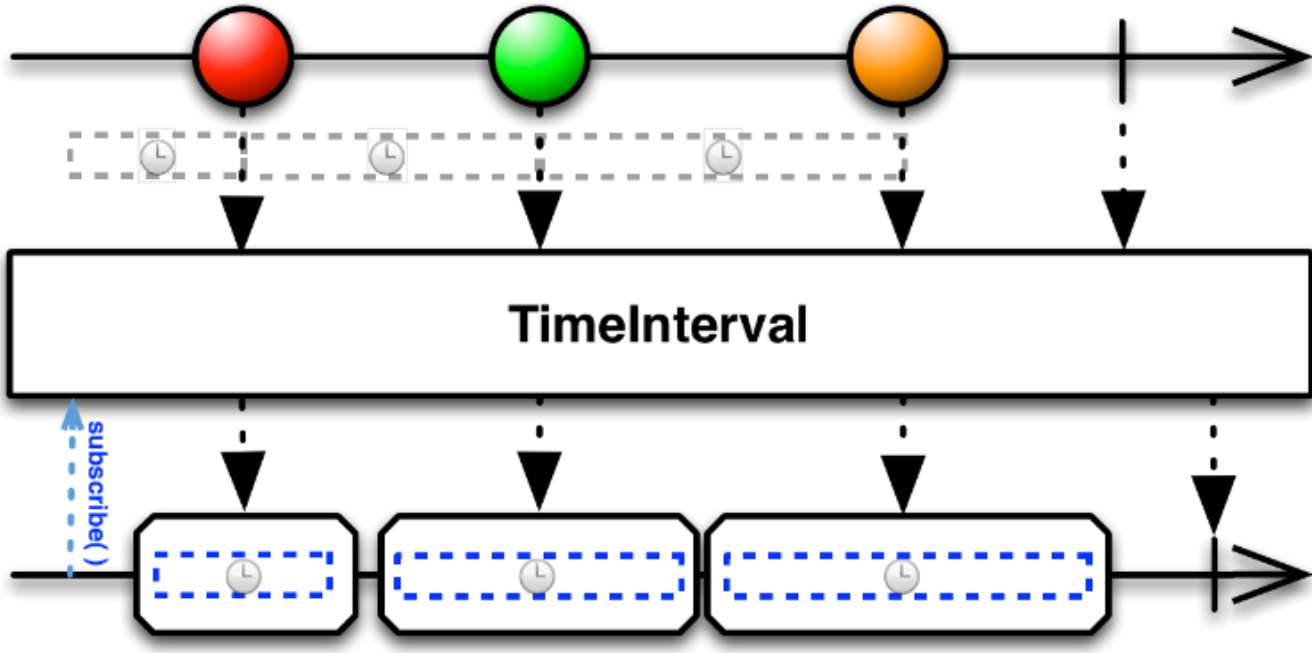
Serialize



一个Observable可以异步调用它的观察者的方法，可能是从不同的线程调用。这可能会让Observable行为不正确，它可能会在某一个onNext调用之前尝试调用onCompleted或onError方法，或者从两个不同的线程同时调用onNext方法。使用Serialize操作符，你可以纠正这个Observable的行为，保证它的行为是正确的且是同步的

TimeInterval

将一个发射数据的Observable 转换为 发射那些数据发射时间间隔的Observable

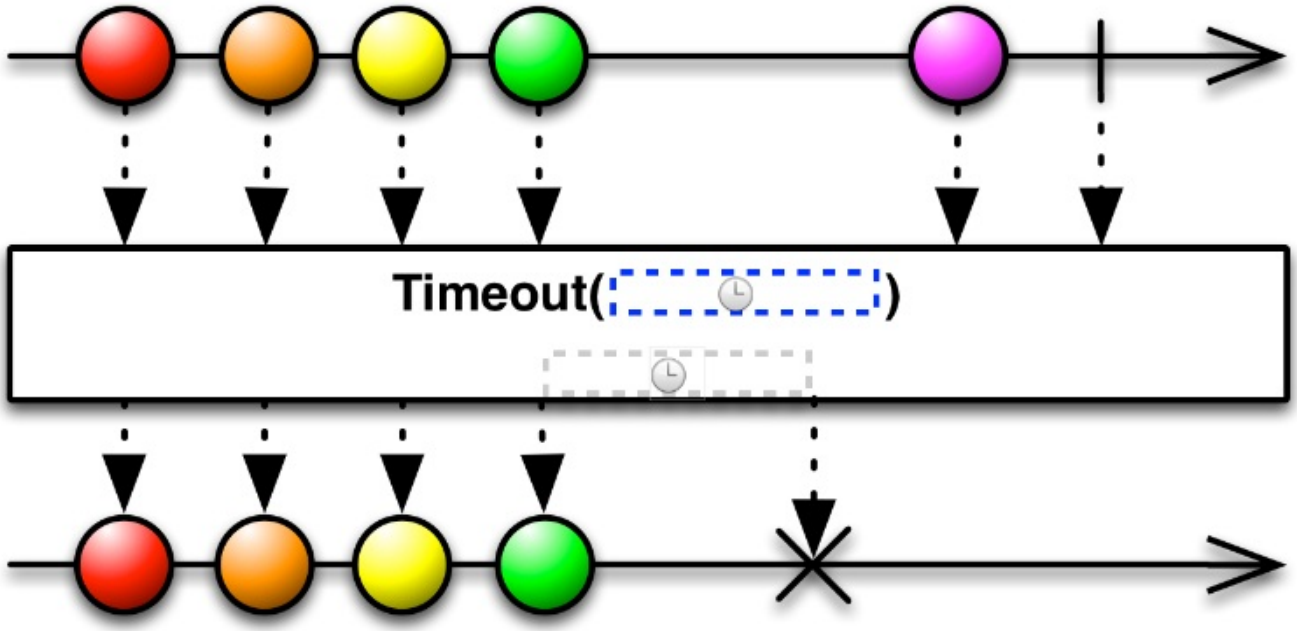


timeInterval操作符将拦截原Observable发射的数据项，替换为发射表示相邻发射物时间间隔的对象。但，未创建与原Observable发射最后一项数据和发射onCompleted通知之间时长对应的时间间隔的对象。

RxJava中的实现为timeInterval，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。

Timeout

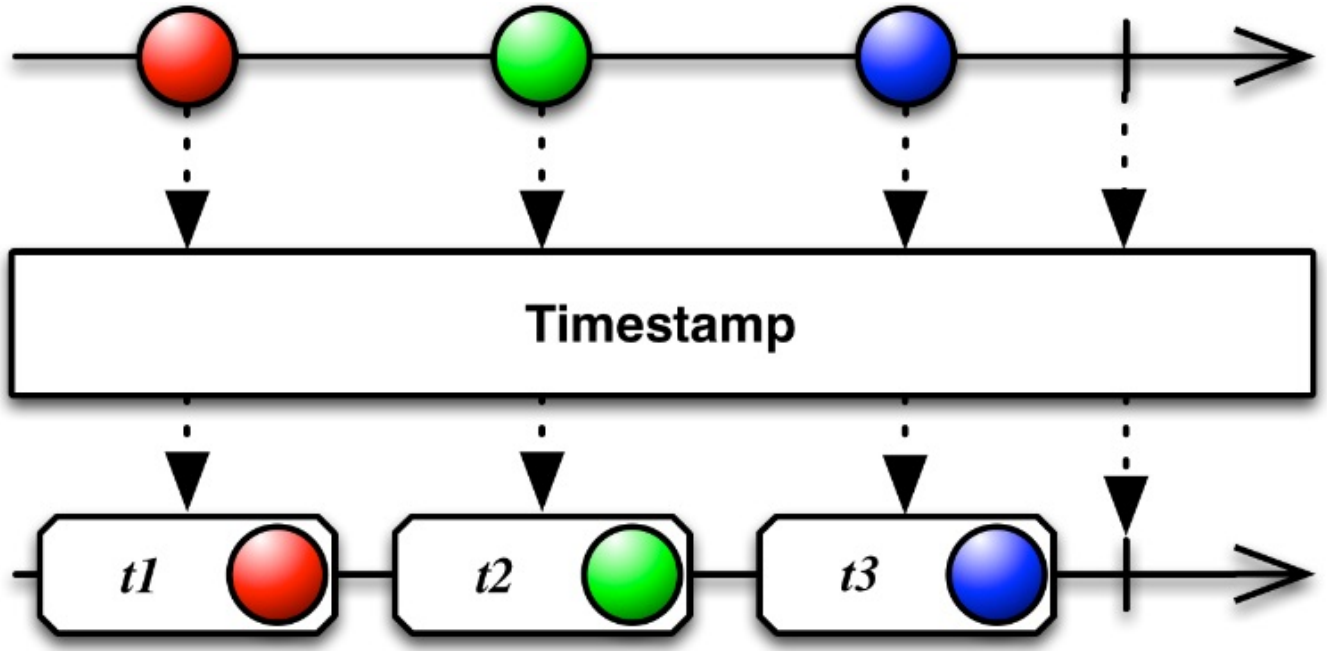
对原始Observable的一个镜像，如果过了一个指定的时长仍没有发射数据，它会发一个错误通知



各种变体。类似与catch的。只是条件不同

Timestamp

给Observable发射的数据项附加一个时间戳

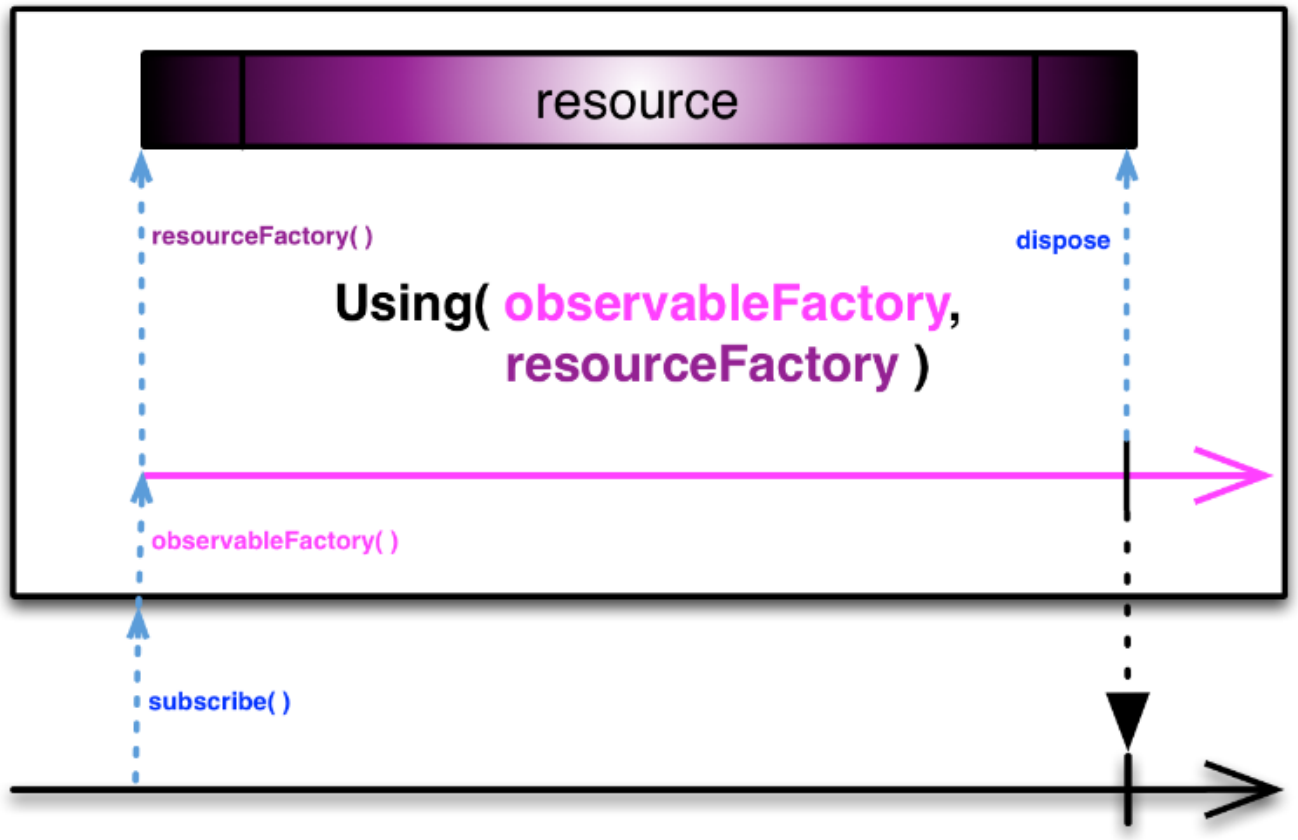


RxJava中的实现为timestamp，它将一个发射T类型数据的Observable转换为一个发射类型为Timestamped的数据的Observable，每一项都包含数据的原始发射时间。

Using

Using操作符创建一个在Observable生命周期内存活的资源，也可以这样理解：我们创建一个资源并使用它，用一个Observable来限制这个资源的使用时间，当这个Observable终止的时候，这个资源就会被销毁。
Using需要使用三个参数，分别是：
1.创建这个一次性资源的函数
2.创建Observable的函数
3.释放资源的函数

当一个观察者订阅using返回的Observable时，using将会使用Observable工厂函数创建观察者要观察的Observable，同时使用资源工厂函数创建一个你想要创建的资源。当观察者取消订阅这个Observable时，或者当观察者终止时（无论是正常终止还是因错误而终止），using使用第三个函数释放它创建的资源。



`Using` 操作符让你可以指示Observable创建一个只在它的生命周期内存在的资源，当Observable终止时这个资源会被自动释放。

```
private Observable<Long> usingObserver() {
        return Observable.using(() -> new Animal(), i -> Observable.timer(5000,
TimeUnit.MILLISECONDS), o -> o.relase());
    }

    private class Animal {
        Subscriber subscriber = new Subscriber() {
            @Override
            public void onCompleted() {

            }

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onNext(Object o) {
                log("animal eat");
            }
        };

        public Animal() {
```

```
            log("create animal");
            Observable.interval(1000, TimeUnit.MILLISECONDS)
                    .subscribe(subscriber);
        }

        public void relase() {
            log("animal released");
            subscriber.unsubscribe();
        }
    }
```

```
Observable<Long> observable = usingObserver();
        Subscriber subscriber = new Subscriber() {
            @Override
            public void onCompleted() {
                log("onCompleted");
            }

            @Override
            public void onError(Throwable e) {
                log("onError");
            }

            @Override
            public void onNext(Object o) {
                log("onNext"+o);
            }
        };
        mLButton.setText("using");
        mLButton.setOnClickListener(e -> observable.subscribe(subscriber));
        mRButton.setText("unSubscrib");
        mRButton.setOnClickListener(e -> subscriber.unsubscribe());
```

运行结果如下。在订阅了几秒之后，对其进行反订阅，Observable就会终止从而触发Animal的释放。

```
create animal
animal eat
animal eat
animal eat
animal released
```