0. 简介

RxJava 其实就是提供一套异步编程的 API，这套 API 是基于观察者模式的，而且是链式调用的，所以使用 RxJava 编写的代码的逻辑会非常简洁。

RxJava 有以下三个基本的元素：

1. 被观察者（Observable）
2. 观察者（Observer）
3. 订阅（subscribe）

下面来说说以上三者是如何协作的：

首先在 gradle 文件中添加依赖：

```
implementation 'io.reactivex.rxjava2:rxjava:2.1.4'
implementation 'io.reactivex.rxjava2:rxandroid:2.0.2'
```

1. 创建被观察者:

```
Observable observable = Observable.create(new
ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        Log.d(TAG, "=======================currentThread name: " +
Thread.currentThread().getName());
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
});
```

2. 创建观察者:

```
Observer observer = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "====================onSubscribe");
    }
```

```
    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "====================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "====================onComplete");
    }
};
```

3. 订阅

```
observable.subscribe(observer);
```

这里其实也可以使用链式调用：

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        Log.d(TAG, "========================currentThread name: " +
Thread.currentThread().getName());
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "====================onSubscribe");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "====================onNext " + integer);
```

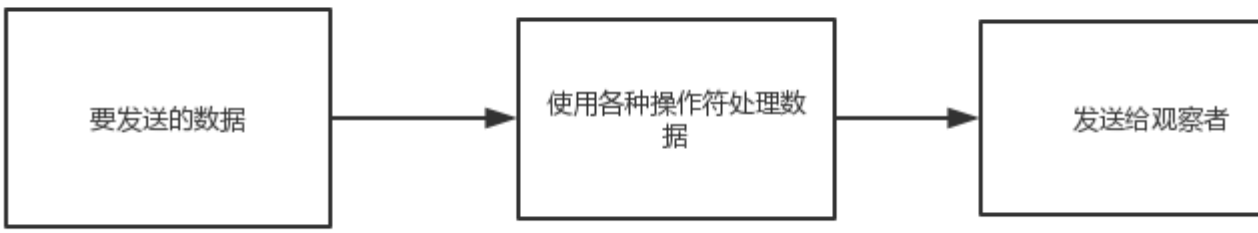
```
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "====================onComplete");
    }
});
```

被观察者发送的事件有以下几种，总结如下表：

事件种类	作用
onNext()	发送该事件时，观察者会回调 onNext() 方法
onError()	发送该事件时，观察者会回调 onError() 方法，当发送该事件之后，其他事件将不会继续发送
onComplete()	发送该事件时，观察者会回调 onComplete() 方法，当发送该事件之后，其他事件将不会继续发送

其实可以把 RxJava 比喻成一个做果汁,家里有很多种水果(要发送的原始数据)，你想榨点水果汁喝一下,这时候你就要想究竟要喝什么水果汁呢？如果你想喝牛油果雪梨柠檬汁，那你就要把这三种水果混在一起榨汁（使用各种操作符变换你想发送给观察者的数据），榨完后，你就可以喝上你想要的果汁了（把处理好的数据发送给观察者）。

总结如下图：



下面就来讲解 RxJava 各种常见的操作符。

1. 创建操作符

以下就是讲解创建被观察者的各种操作符。

1.1 create()

方法预览:

```
public static <T> Observable<T> create(ObservableOnSubscribe<T> source)
```

有什么用:

创建一个被观察者

怎么用:

```
Observable<String> observable = Observable.create(new
ObservableOnSubscribe<String>() {
    @Override
    public void subscribe(ObservableEmitter<String> e) throws Exception
{
        e.onNext("Hello Observer");
        e.onComplete();
    }
});
```

上面的代码非常简单,创建 ObservableOnSubscribe 并重写其 subscribe 方法,就可以通过 ObservableEmitter 发射器向观察者发送事件。

以下创建一个观察者，来验证这个被观察者是否成功创建。

```
Observer<String> observer = new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
```

```
    public void onNext(String s) {
        Log.d("chan","============onNext " + s);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {
        Log.d("chan","============onComplete ");
    }
};

observable.subscribe(observer);
```

打印结果：

```
05-20 16:16:50.654 22935-22935/com.example.louder.rxjavademo D/chan:
============onNext Hello Observer
============onComplete
```

1.2 just()

方法预览:

```
public static <T> Observable<T> just(T item)
......
public static <T> Observable<T> just(T item1, T item2, T item3, T item4,
T item5, T item6, T item7, T item8, T item9, T item10)
```

有什么用?

创建一个被观察者，并发送事件，发送的事件不可以超过 10 个以上。

怎么用?

```
Observable.just(1, 2, 3)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "=================onSubscribe");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "=================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }
});
```

上面的代码直接使用链式调用，代码也非常简单，这里就不细说了，看看打印结果：

```
05-20 16:27:26.938 23281-23281/? D/chan: =================onSubscribe
=================onNext 1
=================onNext 2
=================onNext 3
=================onComplete
```

1.3 From 操作符

1.3.1 fromArray()

方法预览:

```
public static <T> Observable<T> fromArray(T... items)
```

有什么用?

这个方法和 just() 类似，只不过 fromArray 可以传入多于 10 个的变量，并且可以传入一个数组。

怎么用?

```
Integer array[] = {1, 2, 3, 4};
Observable.fromArray(array)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "=================onSubscribe");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "=================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }
});
```

代码和 just() 基本上一样，直接看打印结果：

```
05-20 16:35:23.797 23574-23574/com.example.louder.rxjavademo D/chan:
================onSubscribe
================onNext 1
================onNext 2
================onNext 3
================onNext 4
================onComplete
```

1.3.2 fromCallable()

方法预览:

```
public static <T> Observable<T> fromCallable(Callable<? extends T>
supplier)
```

有什么用?

这里的 Callable 是 java.util.concurrent 中的 Callable，Callable 和 Runnable 的用法基本一致，只是它会返回一个结果值，这个结果值就是发给观察者的。

怎么用?

```
Observable.fromCallable(new Callable < Integer > () {

    @Override
    public Integer call() throws Exception {
        return 1;
    }
})
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "===============accept " + integer);
    }
});
```

打印结果：

```
05-26 13:01:43.009 6890-6890/? D/chan: ===============accept 1
```

1.3.3 fromFuture()

方法预览:

```
public static <T> Observable<T> fromFuture(Future<? extends T> future)
```

有什么用?

参数中的 Future 是 java.util.concurrent 中的 Future,Future 的作用是增加了 cancel() 等方法操作 Callable，它可以通过 get() 方法来获取 Callable 返回的值。

怎么用?

```
FutureTask < String > futureTask = new FutureTask < > (new Callable < String > () {
    @Override
    public String call() throws Exception {
        Log.d(TAG, "CallableDemo is Running");
        return "返回结果";
    }
});

Observable.fromFuture(futureTask)
    .doOnSubscribe(new Consumer < Disposable > () {
    @Override
    public void accept(Disposable disposable) throws Exception {
        futureTask.run();
    }
})
.subscribe(new Consumer < String > () {
    @Override
    public void accept(String s) throws Exception {
        Log.d(TAG, "===============accept " + s);
    }
```

```
});
```

doOnSubscribe() 的作用就是只有订阅时才会发送事件，具体会在下面讲解。

打印结果：

```
05-26 13:54:00.470 14429-14429/com.example.rxjavademo D/chan:
CallableDemo is Running
===============accept 返回结果
```

1.3.4 fromIterable()

方法预览:

```
public static <T> Observable<T> fromIterable(Iterable<? extends T>
source)
```

有什么用?

直接发送一个 List 集合数据给观察者

怎么用?

```
List<Integer> list = new ArrayList<>();
list.add(0);
list.add(1);
list.add(2);
list.add(3);
Observable.fromIterable(list)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "================onSubscribe");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "================onNext " + integer);
```

```
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }
});
```

打印结果如下：

```
05-20 16:43:28.874 23965-23965/? D/chan: =================onSubscribe
=================onNext 0
=================onNext 1
=================onNext 2
=================onNext 3
=================onComplete
```

1.4 defer()

方法预览:

```
public static <T> Observable<T> defer(Callable<? extends
ObservableSource<? extends T>> supplier)
```

有什么用?

这个方法的作用就是直到被观察者被订阅后才会创建被观察者。

怎么用?

```
// i 要定义为成员变量
Integer i = 100;
```

```
Observable<Integer> observable = Observable.defer(new
Callable<ObservableSource<? extends Integer>>() {
    @Override
    public ObservableSource<? extends Integer> call() throws Exception
{
        return Observable.just(i);
    }
});

i = 200;

Observer observer = new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "===============onNext " + integer);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
};

observable.subscribe(observer);

i = 300;

observable.subscribe(observer);
```

打印结果如下：

```
05-20 20:05:01.443 26622-26622/? D/chan: ===============onNext 200
```

```
===============onNext 300
```

因为 defer() 只有观察者订阅的时候才会创建新的被观察者，所以每订阅一次就会打印一次，并且都是打印 i 最新的值。

1.5 timer()

方法预览:

```
public static Observable<Long> timer(long delay, TimeUnit unit)
......
```

有什么用?

当到指定时间后就会发送一个 0L 的值给观察者。

怎么用?

```
Observable.timer(2, TimeUnit.SECONDS)
.subscribe(new Observer < Long > () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Long aLong) {
        Log.d(TAG, "===============onNext " + aLong);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
```

```
});
```

打印结果：

```
05-20 20:27:48.004 27204-27259/com.example.louder.rxjavademo D/chan:
==============onNext 0
```

1.6 interval()

方法预览:

```
public static Observable<Long> interval(long period, TimeUnit unit)
public static Observable<Long> interval(long initialDelay, long period,
TimeUnit unit)
......
```

有什么用?

每隔一段时间就会发送一个事件，这个事件是从 0 开始，不断增 1 的数字。

怎么用?

```
Observable.interval(4, TimeUnit.SECONDS)
.subscribe(new Observer < Long > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==============onSubscribe ");
    }

    @Override
    public void onNext(Long aLong) {
        Log.d(TAG, "==============onNext " + aLong);
    }

    @Override
    public void onError(Throwable e) {

    }
```

```
    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-20 20:48:10.321 28723-28723/com.example.louder.rxjavademo D/chan:
==============onSubscribe
05-20 20:48:14.324 28723-28746/com.example.louder.rxjavademo D/chan:
==============onNext 0
05-20 20:48:18.324 28723-28746/com.example.louder.rxjavademo D/chan:
==============onNext 1
05-20 20:48:22.323 28723-28746/com.example.louder.rxjavademo D/chan:
==============onNext 2
05-20 20:48:26.323 28723-28746/com.example.louder.rxjavademo D/chan:
==============onNext 3
05-20 20:48:30.323 28723-28746/com.example.louder.rxjavademo D/chan:
==============onNext 4
05-20 20:48:34.323 28723-28746/com.example.louder.rxjavademo D/chan:
==============onNext 5
```

从时间就可以看出每隔 4 秒就会发出一次数字递增 1 的事件。这里说下 interval() 第三个方法的 initialDelay 参数，这个参数的意思就是 onSubscribe 回调之后，再次回调 onNext 的间隔时间。

1.7 intervalRange()

方法预览:

```
public static Observable<Long> intervalRange(long start, long count,
long initialDelay, long period, TimeUnit unit)
public static Observable<Long> intervalRange(long start, long count,
long initialDelay, long period, TimeUnit unit, Scheduler scheduler)
```

有什么用?

可以指定发送事件的开始值和数量，其他与 interval() 的功能一样。

怎么用?

```
Observable.intervalRange(2, 5, 2, 1, TimeUnit.SECONDS)
.subscribe(new Observer < Long > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==============onSubscribe ");
    }

    @Override
    public void onNext(Long aLong) {
        Log.d(TAG, "==============onNext " + aLong);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-21 00:03:01.672 2504-2504/com.example.louder.rxjavademo D/chan:
==============onSubscribe
05-21 00:03:03.674 2504-2537/com.example.louder.rxjavademo D/chan:
==============onNext 2
05-21 00:03:04.674 2504-2537/com.example.louder.rxjavademo D/chan:
==============onNext 3
05-21 00:03:05.674 2504-2537/com.example.louder.rxjavademo D/chan:
==============onNext 4
05-21 00:03:06.673 2504-2537/com.example.louder.rxjavademo D/chan:
==============onNext 5
```

```
05-21 00:03:07.674 2504-2537/com.example.louder.rxjavademo D/chan:
==============onNext 6
```

可以看出收到 5 次 onNext 事件，并且是从 2 开始的。

1.8 range()

方法预览:

```
public static Observable<Integer> range(final int start, final int count)
```

有什么用?

同时发送一定范围的事件序列。

怎么用?

```
Observable.range(2, 5)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==============onSubscribe ");
    }

    @Override
    public void onNext(Integer aLong) {
        Log.d(TAG, "==============onNext " + aLong);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-21 00:09:17.202 2921-2921/? D/chan: ==============onSubscribe
==============onNext 2
==============onNext 3
==============onNext 4
==============onNext 5
==============onNext 6
```

1.9 rangeLong()

方法预览:

```
public static Observable<Long> rangeLong(long start, long count)
```

有什么用?

作用与 range() 一样，只是数据类型为 Long

怎么用?

用法与 range() 一样，这里就不再赘述了。

1.10 empty() & never() & error()

方法预览:

```
public static <T> Observable<T> empty()
public static <T> Observable<T> never()
public static <T> Observable<T> error(final Throwable exception)
```

有什么用?

1. empty() ： 直接发送 onComplete() 事件

2. never(): 不发送任何事件
3. error(): 发送 onError() 事件

怎么用?

```
Observable.empty()
.subscribe(new Observer < Object > () {

    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe");
    }

    @Override
    public void onNext(Object o) {
        Log.d(TAG, "==================onNext");
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete");
    }
});
```

打印结果：

```
05-26 14:06:11.881 15798-15798/com.example.rxjavademo D/chan:
==================onSubscribe
==================onComplete
```

换成 never() 的打印结果：

```
05-26 14:12:17.554 16805-16805/com.example.rxjavademo D/chan:
==================onSubscribe
```

换成 error() 的打印结果：

```
05-26 14:12:58.483 17817-17817/com.example.rxjavademo D/chan:
==================onSubscribe
==================onError java.lang.NullPointerException
```

2. 转换操作符

2.1 map()

方法预览:

```
public final <R> Observable<R> map(Function<? super T, ? extends R>
mapper)
```

有什么用?

map 可以将被观察者发送的数据类型转变成其他的类型

怎么用?

以下代码将 Integer 类型的数据转换成 String。

```
Observable.just(1, 2, 3)
.map(new Function < Integer, String > () {
    @Override
    public String apply(Integer integer) throws Exception {
        return "I'm " + integer;
    }
})
.subscribe(new Observer < String > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.e(TAG, "==================onSubscribe");
    }

    @Override
    public void onNext(String s) {
        Log.e(TAG, "==================onNext " + s);
    }
```

```
    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-21 09:16:03.490 5700-5700/com.example.rxjavademo E/chan:
==================onSubscribe
==================onNext I'm 1
==================onNext I'm 2
==================onNext I'm 3
```

2.2 flatMap()

方法预览:

```
public final <R> Observable<R> flatMap(Function<? super T, ? extends
ObservableSource<? extends R>> mapper)
......
```

有什么用?

这个方法可以将事件序列中的元素进行整合加工，返回一个新的被观察者。

怎么用?

flatMap() 其实与 map() 类似,但是 flatMap() 返回的是一个 Observerable。现在用一个例子来说明 flatMap() 的用法。

假设一个有一个 Person 类，这个类的定义如下：

```
public class Person {

    private String name;
    private List<Plan> planList = new ArrayList<>();

    public Person(String name, List<Plan> planList) {
        this.name = name;
        this.planList = planList;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public List<Plan> getPlanList() {
        return planList;
    }

    public void setPlanList(List<Plan> planList) {
        this.planList = planList;
    }

}
```

Person 类有一个 name 和 planList 两个变量,分别代表的是人名和计划清单。

Plan 类的定义如下：

```
public class Plan {

    private String time;
    private String content;
    private List<String> actionList = new ArrayList<>();

    public Plan(String time, String content) {
        this.time = time;
        this.content = content;
    }

    public String getTime() {
```

```
        return time;
    }

    public void setTime(String time) {
        this.time = time;
    }

    public String getContent() {
        return content;
    }

    public void setContent(String content) {
        this.content = content;
    }

    public List<String> getActionList() {
        return actionList;
    }

    public void setActionList(List<String> actionList) {
        this.actionList = actionList;
    }
}
```

现在有一个需求就是要将 Person 集合中的每个元素中的 Plan 的 action 打印出来。 首先用 map() 来实现这个需求看看：

```
Observable.fromIterable(personList)
.map(new Function < Person, List < Plan >> () {
    @Override
    public List < Plan > apply(Person person) throws Exception {
        return person.getPlanList();
    }
})
.subscribe(new Observer < List < Plan >> () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(List < Plan > plans) {
        for (Plan plan: plans) {
```

```
            List < String > planActionList = plan.getActionList();
            for (String action: planActionList) {
                Log.d(TAG, "=================action " + action);
            }
        }
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

可以看到 onNext() 用了嵌套 for 循环来实现，如果代码逻辑复杂起来的话，可能需要多重循环才可以实现。

现在看下使用 flatMap() 实现：

```
Observable.fromIterable(personList)
.flatMap(new Function < Person, ObservableSource < Plan >> () {
    @Override
    public ObservableSource < Plan > apply(Person person) {
        return Observable.fromIterable(person.getPlanList());
    }
})
.flatMap(new Function < Plan, ObservableSource < String >> () {
    @Override
    public ObservableSource < String > apply(Plan plan) throws Exception
{
        return Observable.fromIterable(plan.getActionList());
    }
})
.subscribe(new Observer < String > () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
```

```
    public void onNext(String s) {
        Log.d(TAG, "=================action: " + s);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

从代码可以看出，只需要两个 flatMap() 就可以完成需求，并且代码逻辑非常清晰。

2.3 concatMap()

方法预览:

```
public final <R> Observable<R> concatMap(Function<? super T, ? extends ObservableSource<? extends R>> mapper)
public final <R> Observable<R> concatMap(Function<? super T, ? extends ObservableSource<? extends R>> mapper, int prefetch)
```

有什么用?

concatMap() 和 flatMap() 基本上是一样的，只不过 concatMap() 转发出来的事件是有序的，而 flatMap() 是无序的。

怎么用?

还是使用上面 flatMap() 的例子来讲解，首先来试下 flatMap() 来验证发送的事件是否是无序的，代码如下：

```
Observable.fromIterable(personList)
.flatMap(new Function < Person, ObservableSource < Plan >> () {
```

```
    @Override
    public ObservableSource < Plan > apply(Person person) {
        if ("chan".equals(person.getName())) {
            return
Observable.fromIterable(person.getPlanList()).delay(10,
TimeUnit.MILLISECONDS);
        }
        return Observable.fromIterable(person.getPlanList());
    }
})
.subscribe(new Observer < Plan > () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Plan plan) {
        Log.d(TAG, "==================plan " + plan.getContent());
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

为了更好的验证 flatMap 是无序的，使用了一个 delay() 方法来延迟，直接看打印结果：

```
05-21 13:57:14.031 21616-21616/com.example.rxjavademo D/chan:
==================plan chan 上课
==================plan chan 写作业
==================plan chan 打篮球
05-21 13:57:14.041 21616-21641/com.example.rxjavademo D/chan:
==================plan Zede 开会
==================plan Zede 写代码
==================plan Zede 写文章
```

可以看到本来 Zede 的事件发送顺序是排在 chan 事件之前，但是经过延迟后，这两个事件序列发送顺序互换了。

现在来验证下 concatMap() 是否是有序的，使用上面同样的代码，只是把 flatMap() 换成 concatMap()，打印结果如下：

```
05-21 13:58:42.917 21799-21823/com.example.rxjavademo D/chan:
==================plan Zede 开会
==================plan Zede 写代码
==================plan Zede 写文章
==================plan chan 上课
==================plan chan 写作业
==================plan chan 打篮球
```

这就代表 concatMap() 转换后发送的事件序列是有序的了。

2.4 buffer()

方法预览:

```
public final Observable<List<T>> buffer(int count, int skip)
......
```

有什么用?

从需要发送的事件当中获取一定数量的事件,并将这些事件放到缓冲区当中一并发出。

怎么用?

buffer 有两个参数，一个是 count，另一个 skip。count 缓冲区元素的数量，skip 就代表缓冲区满了之后，发送下一次事件序列的时候要跳过多少元素。这样说可能还是有点抽象，直接看代码：

```
Observable.just(1, 2, 3, 4, 5)
.buffer(2, 1)
.subscribe(new Observer < List < Integer >> () {
```

```
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(List < Integer > integers) {
        Log.d(TAG, "===============缓冲区大小： " + integers.size());
        for (Integer i: integers) {
            Log.d(TAG, "===============元素： " + i);
        }
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-21 14:09:34.015 22421-22421/com.example.rxjavademo D/chan:
===============缓冲区大小： 2
===============元素： 1
===============元素： 2
===============缓冲区大小： 2
===============元素： 2
===============元素： 3
===============缓冲区大小： 2
===============元素： 3
===============元素： 4
===============缓冲区大小： 2
===============元素： 4
===============元素： 5
===============缓冲区大小： 1
===============元素： 5
```

从结果可以看出，每次发送事件，指针都会往后移动一个元素再取值，直到指针移动到没有元素的时候就会停止取值。

2.5 groupBy()

方法预览:

```
public final <K> Observable<GroupedObservable<K, T>> groupBy(Function<? super T, ? extends K> keySelector)
```

有什么用?

将发送的数据进行分组，每个分组都会返回一个被观察者。

怎么用?

```
Observable.just(5, 2, 3, 4, 1, 6, 8, 9, 7, 10)
.groupBy(new Function < Integer, Integer > () {
    @Override
    public Integer apply(Integer integer) throws Exception {
        return integer % 3;
    }
})
.subscribe(new Observer < GroupedObservable < Integer, Integer >> () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "====================onSubscribe ");
    }

    @Override
    public void onNext(GroupedObservable < Integer, Integer >
integerIntegerGroupedObservable) {
        Log.d(TAG, "====================onNext ");
        integerIntegerGroupedObservable.subscribe(new Observer <
Integer > () {
            @Override
            public void onSubscribe(Disposable d) {
                Log.d(TAG, "====================GroupedObservable
onSubscribe ");
            }
```

```
            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "====================GroupedObservable
onNext  groupName: " + integerIntegerGroupedObservable.getKey() + "
value: " + integer);
            }

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

            @Override
            public void onComplete() {
                Log.d(TAG, "====================GroupedObservable
onComplete ");
            }
        });
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "====================onComplete ");
    }
});
```

在 groupBy() 方法返回的参数是分组的名字，每返回一个值，那就代表会创建一个组，以上的代码就是将 1~10 的数据分成 3 组，来看看打印结果：

```
05-26 14:38:02.062 21451-21451/com.example.rxjavademo D/chan:
====================onSubscribe
05-26 14:38:02.063 21451-21451/com.example.rxjavademo D/chan:
====================onNext
====================GroupedObservable onSubscribe
====================GroupedObservable onNext  groupName: 2 value: 5
====================GroupedObservable onNext  groupName: 2 value: 2
```

```
==================onNext
==================GroupedObservable onSubscribe
==================GroupedObservable onNext  groupName: 0 value: 3
05-26 14:38:02.064 21451-21451/com.example.rxjavademo D/chan:
==================onNext
==================GroupedObservable onSubscribe
==================GroupedObservable onNext  groupName: 1 value: 4
==================GroupedObservable onNext  groupName: 1 value: 1
==================GroupedObservable onNext  groupName: 0 value: 6
==================GroupedObservable onNext  groupName: 2 value: 8
==================GroupedObservable onNext  groupName: 0 value: 9
==================GroupedObservable onNext  groupName: 1 value: 7
==================GroupedObservable onNext  groupName: 1 value: 10
05-26 14:38:02.065 21451-21451/com.example.rxjavademo D/chan:
==================GroupedObservable onComplete
==================GroupedObservable onComplete
==================GroupedObservable onComplete
==================onComplete
```

可以看到返回的结果中是有 3 个组的。

2.6 scan()

方法预览:

```
public final Observable<T> scan(BiFunction<T, T, T> accumulator)
```

有什么用?

将数据以一定的逻辑聚合起来。

怎么用?

```
Observable.just(1, 2, 3, 4, 5)
.scan(new BiFunction < Integer, Integer, Integer > () {
    @Override
    public Integer apply(Integer integer, Integer integer2) throws
Exception {
        Log.d(TAG, "==================apply ");
```

```
        Log.d(TAG, "==================integer " + integer);
        Log.d(TAG, "==================integer2 " + integer2);
        return integer + integer2;
    }
})
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "==================accept " + integer);
    }
});
```

打印结果：

```
05-26 14:45:27.784 22519-22519/com.example.rxjavademo D/chan:
==================accept 1
==================apply
==================integer 1
==================integer2 2
==================accept 3
==================apply
05-26 14:45:27.785 22519-22519/com.example.rxjavademo D/chan:
==================integer 3
==================integer2 3
==================accept 6
==================apply
==================integer 6
==================integer2 4
==================accept 10
==================apply
==================integer 10
==================integer2 5
==================accept 15
```

2.7 window()

方法预览:

```
public final Observable<Observable<T>> window(long count)
......
```

有什么用?

发送指定数量的事件时，就将这些事件分为一组。window 中的 count 的参数就是代表指定的数量，例如将 count 指定为 2，那么每发 2 个数据就会将这 2 个数据分成一组。

怎么用?

```
Observable.just(1, 2, 3, 4, 5)
.window(2)
.subscribe(new Observer < Observable < Integer >> () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "====================onSubscribe ");
    }

    @Override
    public void onNext(Observable < Integer > integerObservable) {
        integerObservable.subscribe(new Observer < Integer > () {
            @Override
            public void onSubscribe(Disposable d) {
                Log.d(TAG, "====================integerObservable
onSubscribe ");
            }

            @Override
            public void onNext(Integer integer) {
                Log.d(TAG, "====================integerObservable
onNext " + integer);
            }

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

            @Override
            public void onComplete() {
                Log.d(TAG, "====================integerObservable
onComplete ");
            }
```

```
        });
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "====================onComplete ");
    }
});
```

打印结果：

```
05-26 15:02:20.654 25838-25838/com.example.rxjavademo D/chan:
====================onSubscribe
05-26 15:02:20.655 25838-25838/com.example.rxjavademo D/chan:
====================integerObservable onSubscribe
05-26 15:02:20.656 25838-25838/com.example.rxjavademo D/chan:
====================integerObservable onNext 1
====================integerObservable onNext 2
====================integerObservable onComplete
====================integerObservable onSubscribe
====================integerObservable onNext 3
====================integerObservable onNext 4
====================integerObservable onComplete
====================integerObservable onSubscribe
====================integerObservable onNext 5
====================integerObservable onComplete
====================onComplete
```

从结果可以发现，window() 将 1~5 的事件分成了 3 组。

3. 组合操作符

3.1 concat()

方法预览:

```
public static <T> Observable<T> concat(ObservableSource<? extends T>
source1, ObservableSource<? extends T> source2, ObservableSource<?
extends T> source3, ObservableSource<? extends T> source4)
......
```

有什么用?

可以将多个观察者组合在一起,然后按照之前发送顺序发送事件。需要注意的是,concat() 最多只可以发送 4 个事件。

怎么用?

```
Observable.concat(Observable.just(1, 2),
Observable.just(3, 4),
Observable.just(5, 6),
Observable.just(7, 8))
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "===============onNext " + integer);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
```

```
});
```

打印如下：

```
05-21 15:40:26.738 7477-7477/com.example.rxjavademo D/chan:
===============onNext 1
===============onNext 2
05-21 15:40:26.739 7477-7477/com.example.rxjavademo D/chan:
===============onNext 3
===============onNext 4
===============onNext 5
===============onNext 6
===============onNext 7
===============onNext 8
```

3.2 concatArray()

方法预览:

```
public static <T> Observable<T> concatArray(ObservableSource<? extends
T>... sources)
```

有什么用?

与 concat() 作用一样，不过 concatArray() 可以发送多于 4 个被观察者。

怎么用?

```
Observable.concatArray(Observable.just(1, 2),
Observable.just(3, 4),
Observable.just(5, 6),
Observable.just(7, 8),
Observable.just(9, 10))
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {

    }
```

```
    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "===============onNext " + integer);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-21 15:47:18.581 9129-9129/com.example.rxjavademo D/chan:
===============onNext 1
===============onNext 2
===============onNext 3
===============onNext 4
===============onNext 5
===============onNext 6
===============onNext 7
===============onNext 8
===============onNext 9
===============onNext 10
```

3.3 merge()

方法预览:

```
 public static <T> Observable<T> merge(ObservableSource<? extends T>
source1, ObservableSource<? extends T> source2, ObservableSource<?
extends T> source3, ObservableSource<? extends T> source4)
......
```

有什么用?

这个方法月 concat() 作用基本一样，知识 concat() 是串行发送事件，而 merge() 并行发送事件。

怎么用?

现在来演示 concat() 和 merge() 的区别。

```
Observable.merge(
Observable.interval(1, TimeUnit.SECONDS).map(new Function < Long,
String > () {
    @Override
    public String apply(Long aLong) throws Exception {
        return "A" + aLong;
    }
}),
Observable.interval(1, TimeUnit.SECONDS).map(new Function < Long,
String > () {
    @Override
    public String apply(Long aLong) throws Exception {
        return "B" + aLong;
    }
}))
    .subscribe(new Observer < String > () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(String s) {
        Log.d(TAG, "====================onNext " + s);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {
```

```
    }
});
```

打印结果如下：

```
05-21 16:10:31.125 12801-12850/com.example.rxjavademo D/chan:
====================onNext B0
05-21 16:10:31.125 12801-12849/com.example.rxjavademo D/chan:
====================onNext A0
05-21 16:10:32.125 12801-12849/com.example.rxjavademo D/chan:
====================onNext A1
05-21 16:10:32.126 12801-12850/com.example.rxjavademo D/chan:
====================onNext B1
05-21 16:10:33.125 12801-12849/com.example.rxjavademo D/chan:
====================onNext A2
05-21 16:10:33.125 12801-12850/com.example.rxjavademo D/chan:
====================onNext B2
05-21 16:10:34.125 12801-12849/com.example.rxjavademo D/chan:
====================onNext A3
05-21 16:10:34.125 12801-12850/com.example.rxjavademo D/chan:
====================onNext B3
05-21 16:10:35.124 12801-12849/com.example.rxjavademo D/chan:
====================onNext A4
05-21 16:10:35.125 12801-12850/com.example.rxjavademo D/chan:
====================onNext B4
05-21 16:10:36.125 12801-12849/com.example.rxjavademo D/chan:
====================onNext A5
05-21 16:10:36.125 12801-12850/com.example.rxjavademo D/chan:
====================onNext B5
......
```

从结果可以看出，A 和 B 的事件序列都可以发出，将以上的代码换成 concat() 看看打印结果：

```
05-21 16:17:52.352 14597-14621/com.example.rxjavademo D/chan:
====================onNext A0
05-21 16:17:53.351 14597-14621/com.example.rxjavademo D/chan:
====================onNext A1
05-21 16:17:54.351 14597-14621/com.example.rxjavademo D/chan:
====================onNext A2
05-21 16:17:55.351 14597-14621/com.example.rxjavademo D/chan:
====================onNext A3
```

```
05-21 16:17:56.351 14597-14621/com.example.rxjavademo D/chan:
====================onNext A4
05-21 16:17:57.351 14597-14621/com.example.rxjavademo D/chan:
====================onNext A5
......
```

从结果可以知道，只有等到第一个被观察者发送完事件之后，第二个被观察者才会发送事件。

mergeArray() 与 merge() 的作用是一样的，只是它可以发送 4 个以上的被观察者，这里就不再赘述了。

3.4 concatArrayDelayError() & mergeArrayDelayError()

方法预览:

```
public static <T> Observable<T> concatArrayDelayError(ObservableSource<?
extends T>... sources)
public static <T> Observable<T> mergeArrayDelayError(ObservableSource<?
extends T>... sources)
```

有什么用?

在 concatArray() 和 mergeArray() 两个方法当中，如果其中有一个被观察者发送了一个 Error 事件，那么就会停止发送事件，如果你想 onError() 事件延迟到所有被观察者都发送完事件后再执行的话，就可以使用 concatArrayDelayError() 和 mergeArrayDelayError()

怎么用?

首先使用 concatArray() 来验证一下发送 onError() 事件是否会中断其他被观察者发送事件，代码如下：

```
Observable.concatArray(
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
```

```
        e.onError(new NumberFormatException());
    }
}), Observable.just(2, 3, 4))
    .subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {

    }
});
```

打印结果：

```
05-21 16:38:59.725 17985-17985/com.example.rxjavademo D/chan:
==================onNext 1
==================onError
```

从结果可以知道,确实中断了,现在换用 concatArrayDelayError(),代码如下：

```
Observable.concatArrayDelayError(
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onError(new NumberFormatException());
    }
}), Observable.just(2, 3, 4))
.subscribe(new Observer < Integer > () {
```

```
    @Override
    public void onSubscribe(Disposable d) {

    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {

    }
});
```

打印结果如下：

```
05-21 16:40:59.329 18199-18199/com.example.rxjavademo D/chan:
==================onNext 1
==================onNext 2
==================onNext 3
==================onNext 4
==================onError
```

从结果可以看到，onError 事件是在所有被观察者发送完事件才发送的。
mergeArrayDelayError() 也是有同样的作用，这里不再赘述。

3.5 zip()

方法预览:

```
public static <T1, T2, R> Observable<R> zip(ObservableSource<? extends
T1> source1, ObservableSource<? extends T2> source2, BiFunction<? super
T1, ? super T2, ? extends R> zipper)
......
```

有什么用?

会将多个被观察者合并，根据各个被观察者发送事件的顺序一个个结合起来，最终发送的事件数量会与源 Observable 中最少事件的数量一样。

怎么用?

```
Observable.zip(Observable.intervalRange(1, 5, 1, 1, TimeUnit.SECONDS)
    .map(new Function<Long, String>() {
        @Override
        public String apply(Long aLong) throws Exception {
            String s1 = "A" + aLong;
            Log.d(TAG, "==================A 发送的事件 " + s1);
            return s1;
        }}),
        Observable.intervalRange(1, 6, 1, 1, TimeUnit.SECONDS)
            .map(new Function<Long, String>() {
            @Override
            public String apply(Long aLong) throws Exception {
                String s2 = "B" + aLong;
                Log.d(TAG, "==================B 发送的事件 " + s2);
                return s2;
            }
        }),
        new BiFunction<String, String, String>() {
            @Override
            public String apply(String s, String s2) throws Exception {
                String res = s + s2;
                return res;
            }
        })
.subscribe(new Observer<String>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(String s) {
        Log.d(TAG, "==================onNext " + s);
    }
```

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

上面代码中有两个 Observable，第一个发送事件的数量为 5 个，第二个发送事件的数量为 6 个。现在来看下打印结果：

```
05-22 09:10:39.952 5338-5338/com.example.rxjavademo D/chan:
==================onSubscribe
05-22 09:10:40.953 5338-5362/com.example.rxjavademo D/chan:
==================A 发送的事件 A1
05-22 09:10:40.953 5338-5363/com.example.rxjavademo D/chan:
==================B 发送的事件 B1
==================onNext A1B1
05-22 09:10:41.953 5338-5362/com.example.rxjavademo D/chan:
==================A 发送的事件 A2
05-22 09:10:41.954 5338-5363/com.example.rxjavademo D/chan:
==================B 发送的事件 B2
==================onNext A2B2
05-22 09:10:42.953 5338-5362/com.example.rxjavademo D/chan:
==================A 发送的事件 A3
05-22 09:10:42.953 5338-5363/com.example.rxjavademo D/chan:
==================B 发送的事件 B3
05-22 09:10:42.953 5338-5362/com.example.rxjavademo D/chan:
==================onNext A3B3
05-22 09:10:43.953 5338-5362/com.example.rxjavademo D/chan:
==================A 发送的事件 A4
05-22 09:10:43.953 5338-5363/com.example.rxjavademo D/chan:
==================B 发送的事件 B4
05-22 09:10:43.954 5338-5363/com.example.rxjavademo D/chan:
==================onNext A4B4
05-22 09:10:44.953 5338-5362/com.example.rxjavademo D/chan:
==================A 发送的事件 A5
05-22 09:10:44.953 5338-5363/com.example.rxjavademo D/chan:
==================B 发送的事件 B5
```

```
05-22 09:10:44.954 5338-5363/com.example.rxjavademo D/chan:
==================onNext A5B5
==================onComplete
```

可以发现最终接收到的事件数量是 5，那么为什么第二个 Observable 没有发送第 6 个事件呢？因为在这之前第一个 Observable 已经发送了 onComplete 事件，所以第二个 Observable 不会再发送事件。

3.6 combineLatest() & combineLatestDelayError()

方法预览:

```
public static <T1, T2, R> Observable<R> combineLatest(ObservableSource<? extends T1> source1, ObservableSource<? extends T2> source2, BiFunction<? super T1, ? super T2, ? extends R> combiner)
.......
```

有什么用?

combineLatest() 的作用与 zip() 类似，但是 combineLatest() 发送事件的序列是与发送的时间线有关的，当 combineLatest() 中所有的 Observable 都发送了事件，只要其中有一个 Observable 发送事件，这个事件就会和其他 Observable 最近发送的事件结合起来发送，这样可能还是比较抽象，看看以下例子代码。

怎么用?

```
Observable.combineLatest(
Observable.intervalRange(1, 4, 1, 1, TimeUnit.SECONDS)
    .map(new Function < Long, String > () {@Override
    public String apply(Long aLong) throws Exception {
        String s1 = "A" + aLong;
        Log.d(TAG, "==================A 发送的事件 " + s1);
        return s1;
    }
}),
Observable.intervalRange(1, 5, 2, 2, TimeUnit.SECONDS)
    .map(new Function < Long, String > () {@Override
    public String apply(Long aLong) throws Exception {
```

```
        String s2 = "B" + aLong;
        Log.d(TAG, "==================B 发送的事件 " + s2);
        return s2;
    }
}),
new BiFunction < String, String, String > () {@Override
    public String apply(String s, String s2) throws Exception {
        String res = s + s2;
        return res;
    }
})
.subscribe(new Observer < String > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(String s) {
        Log.d(TAG, "==================最终接收到的事件 " + s);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

分析上面的代码，Observable A 会每隔 1 秒就发送一次事件，Observable B 会隔 2 秒发送一次事件。来看看打印结果：

```
05-22 11:41:20.859 15104-15104/? D/chan: ==================onSubscribe
05-22 11:41:21.859 15104-15128/com.example.rxjavademo D/chan:
==================A 发送的事件 A1
05-22 11:41:22.860 15104-15128/com.example.rxjavademo D/chan:
==================A 发送的事件 A2
05-22 11:41:22.861 15104-15129/com.example.rxjavademo D/chan:
==================B 发送的事件 B1
```

```
05-22 11:41:22.862 15104-15129/com.example.rxjavademo D/chan:
==================最终接收到的事件 A2B1
05-22 11:41:23.860 15104-15128/com.example.rxjavademo D/chan:
==================A 发送的事件 A3
==================最终接收到的事件 A3B1
05-22 11:41:24.860 15104-15128/com.example.rxjavademo D/chan:
==================A 发送的事件 A4
05-22 11:41:24.861 15104-15129/com.example.rxjavademo D/chan:
==================B 发送的事件 B2
05-22 11:41:24.861 15104-15128/com.example.rxjavademo D/chan:
==================最终接收到的事件 A4B1
05-22 11:41:24.861 15104-15129/com.example.rxjavademo D/chan:
==================最终接收到的事件 A4B2
05-22 11:41:26.860 15104-15129/com.example.rxjavademo D/chan:
==================B 发送的事件 B3
05-22 11:41:26.861 15104-15129/com.example.rxjavademo D/chan:
==================最终接收到的事件 A4B3
05-22 11:41:28.860 15104-15129/com.example.rxjavademo D/chan:
==================B 发送的事件 B4
05-22 11:41:28.861 15104-15129/com.example.rxjavademo D/chan:
==================最终接收到的事件 A4B4
05-22 11:41:30.860 15104-15129/com.example.rxjavademo D/chan:
==================B 发送的事件 B5
05-22 11:41:30.861 15104-15129/com.example.rxjavademo D/chan:
==================最终接收到的事件 A4B5
==================onComplete
```

分析上述结果可以知道，当发送 A1 事件之后，因为 B 并没有发送任何事件，所以根本不会发生结合。当 B 发送了 B1 事件之后，就会与 A 最近发送的事件 A2 结合成 A2B1，这样只有后面一有被观察者发送事件，这个事件就会与其他被观察者最近发送的事件结合起来了。

因为 combineLatestDelayError() 就是多了延迟发送 onError() 功能，这里就不再赘述了。

3.7 reduce()

方法预览:

```
public final Maybe<T> reduce(BiFunction<T, T, T> reducer)
```

有什么用?

与 scan() 操作符的作用也是将发送数据以一定逻辑聚合起来,这两个的区别在于 scan() 每处理一次数据就会将事件发送给观察者,而 reduce() 会将所有数据聚合在一起才会发送事件给观察者。

怎么用?

```
Observable.just(0, 1, 2, 3)
.reduce(new BiFunction < Integer, Integer, Integer > () {
    @Override
    public Integer apply(Integer integer, Integer integer2) throws
Exception {
        int res = integer + integer2;
        Log.d(TAG, "====================integer " + integer);
        Log.d(TAG, "====================integer2 " + integer2);
        Log.d(TAG, "====================res " + res);
        return res;
    }
})
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "=================accept " + integer);
    }
});
```

打印结果:

```
05-22 14:21:46.042 17775-17775/? D/chan: ====================integer 0
====================integer2 1
====================res 1
====================integer 1
====================integer2 2
====================res 3
====================integer 3
====================integer2 3
====================res 6
=================accept 6
```

从结果可以看到,其实就是前2个数据聚合之后,然后再与后1个数据进行聚合,一直到没有数据为止。

3.8 collect()

方法预览:

```
public final <U> Single<U> collect(Callable<? extends U>
initialValueSupplier, BiConsumer<? super U, ? super T> collector)
```

有什么用?

将数据收集到数据结构当中。

怎么用?

```
Observable.just(1, 2, 3, 4)
.collect(new Callable < ArrayList < Integer >> () {
    @Override
    public ArrayList < Integer > call() throws Exception {
        return new ArrayList < > ();
    }
},
new BiConsumer < ArrayList < Integer > , Integer > () {
    @Override
    public void accept(ArrayList < Integer > integers, Integer integer)
throws Exception {
        integers.add(integer);
    }
})
.subscribe(new Consumer < ArrayList < Integer >> () {
    @Override
    public void accept(ArrayList < Integer > integers) throws Exception
{
        Log.d(TAG, "==============accept " + integers);
    }
});
```

打印结果:

```
05-22 16:47:18.257 31361-31361/com.example.rxjavademo D/chan:
==============accept [1, 2, 3, 4]
```

3.9 startWith() & startWithArray()

方法预览:

```
public final Observable<T> startWith(T item)
public final Observable<T> startWithArray(T... items)
```

有什么用?

在发送事件之前追加事件，startWith() 追加一个事件，startWithArray() 可以追加多个事件。追加的事件会先发出。

怎么用?

```
Observable.just(5, 6, 7)
.startWithArray(2, 3, 4)
.startWith(1)
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "===============accept " + integer);
    }
});
```

打印结果：

```
05-22 17:08:21.282 4505-4505/com.example.rxjavademo D/chan:
===============accept 1
===============accept 2
===============accept 3
===============accept 4
===============accept 5
===============accept 6
===============accept 7
```

3.10 count()

方法预览:

```
public final Single<Long> count()
```

有什么用?

返回被观察者发送事件的数量。

怎么用?

```
Observable.just(1, 2, 3)
.count()
.subscribe(new Consumer < Long > () {
    @Override
    public void accept(Long aLong) throws Exception {
        Log.d(TAG, "======================aLong " + aLong);
    }
});
```

打印结果：

```
05-22 20:41:25.025 14126-14126/? D/chan: ======================aLong 3
```

4. 功能操作符

4.1 delay()

方法预览:

```
public final Observable<T> delay(long delay, TimeUnit unit)
```

有什么用?

延迟一段事件发送事件。

怎么用?

```
Observable.just(1, 2, 3)
.delay(2, TimeUnit.SECONDS)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "======================onSubscribe");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "======================onNext " + integer);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
    public void onComplete() {
        Log.d(TAG, "======================onSubscribe");
    }
});
```

这里延迟了两秒才发送事件，来看看打印结果：

```
05-22 20:53:43.618 16880-16880/com.example.rxjavademo D/chan:
======================onSubscribe
05-22 20:53:45.620 16880-16906/com.example.rxjavademo D/chan:
======================onNext 1
05-22 20:53:45.621 16880-16906/com.example.rxjavademo D/chan:
======================onNext 2
======================onNext 3
======================onSubscribe
```

从打印结果可以看出 onSubscribe 回调 2 秒之后 onNext 才会回调。

4.2 doOnEach()

方法预览:

```
public final Observable<T> doOnEach(final Consumer<? super
Notification<T>> onNotification)
```

有什么用?

Observable 每发送一件事件之前都会先回调这个方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        //      e.onError(new NumberFormatException());
        e.onComplete();
    }
})
.doOnEach(new Consumer < Notification < Integer >> () {
    @Override
    public void accept(Notification < Integer > integerNotification)
throws Exception {
        Log.d(TAG, "==================doOnEach " +
integerNotification.getValue());
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
```

```
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 09:07:05.547 19867-19867/? D/chan: ==================onSubscribe
==================doOnEach 1
==================onNext 1
==================doOnEach 2
==================onNext 2
==================doOnEach 3
==================onNext 3
==================doOnEach null
==================onComplete
```

从结果就可以看出每发送一个事件之前都会回调 doOnEach 方法，并且可以取出 onNext() 发送的值。

4.3 doOnNext()

方法预览:

```
public final Observable<T> doOnNext(Consumer<? super T> onNext)
```

有什么用?

Observable 每发送 onNext() 之前都会先回调这个方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.doOnNext(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "==================doOnNext " + integer);
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果:

```
05-23 09:09:36.769 20020-20020/com.example.rxjavademo D/chan:
==================onSubscribe
```

```
==================doOnNext 1
==================onNext 1
==================doOnNext 2
==================onNext 2
==================doOnNext 3
==================onNext 3
==================onComplete
```

4.4 doAfterNext()

方法预览:

```
public final Observable<T> doAfterNext(Consumer<? super T> onAfterNext)
```

有什么用?

Observable 每发送 onNext() 之后都会回调这个方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.doAfterNext(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "==================doAfterNext " + integer);
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }
```

```
    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 09:15:49.215 20432-20432/com.example.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================doAfterNext 1
==================onNext 2
==================doAfterNext 2
==================onNext 3
==================doAfterNext 3
==================onComplete
```

4.5 doOnComplete()

方法预览:

```
public final Observable<T> doOnComplete(Action onComplete)
```

有什么用?

Observable 每发送 onComplete() 之前都会回调这个方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.doOnComplete(new Action() {
    @Override
    public void run() throws Exception {
        Log.d(TAG, "==================doOnComplete ");
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果:

```
05-23 09:32:18.031 20751-20751/? D/chan: ==================onSubscribe
==================onNext 1
```

```
==================onNext 2
==================onNext 3
==================doOnComplete
==================onComplete
```

4.6 doOnError()

方法预览:

```
public final Observable<T> doOnError(Consumer<? super Throwable>
onError)
```

有什么用?

Observable 每发送 onError() 之前都会回调这个方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onError(new NullPointerException());
    }
})
.doOnError(new Consumer < Throwable > () {
    @Override
    public void accept(Throwable throwable) throws Exception {
        Log.d(TAG, "==================doOnError " + throwable);
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
```

```
    public void onNext(Integer integer) {
        Log.d(TAG, "=================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }
});
```

打印结果：

```
05-23 09:35:04.150 21051-21051/? D/chan: =================onSubscribe
=================onNext 1
=================onNext 2
=================onNext 3
=================doOnError java.lang.NullPointerException
=================onError
```

4.7 doOnSubscribe()

方法预览:

```
public final Observable<T> doOnSubscribe(Consumer<? super Disposable>
onSubscribe)
```

有什么用?

Observable 每发送 onSubscribe() 之前都会回调这个方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
```

```
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.doOnSubscribe(new Consumer < Disposable > () {
    @Override
    public void accept(Disposable disposable) throws Exception {
        Log.d(TAG, "==================doOnSubscribe ");
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 09:39:25.778 21245-21245/? D/chan:
==================doOnSubscribe
==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onComplete
```

4.8 doOnDispose()

方法预览:

```
public final Observable<T> doOnDispose(Action onDispose)
```

有什么用?

当调用 Disposable 的 dispose() 之后回调该方法。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.doOnDispose(new Action() {
    @Override
    public void run() throws Exception {
        Log.d(TAG, "==================doOnDispose ");
    }
})
.subscribe(new Observer < Integer > () {
    private Disposable d;

    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
        this.d = d;
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
        d.dispose();
```

```
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 09:55:48.122 22023-22023/com.example.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================doOnDispose
```

4.9 doOnLifecycle()

方法预览:

```
public final Observable<T> doOnLifecycle(final Consumer<? super
Disposable> onSubscribe, final Action onDispose)
```

有什么用?

在回调 onSubscribe 之前回调该方法的第一个参数的回调方法，可以使用该回调方法决定是否取消订阅。

怎么用?

doOnLifecycle() 第二个参数的回调方法的作用与 doOnDispose() 是一样的，现在用下面的例子来讲解：

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
```

```
            e.onNext(1);
            e.onNext(2);
            e.onNext(3);
            e.onComplete();
        }
    })
    .doOnLifecycle(new Consumer<Disposable>() {
        @Override
        public void accept(Disposable disposable) throws Exception {
            Log.d(TAG, "==================doOnLifecycle accept");
        }
    }, new Action() {
        @Override
        public void run() throws Exception {
            Log.d(TAG, "==================doOnLifecycle Action");
        }
    })
    .doOnDispose(
        new Action() {
            @Override
            public void run() throws Exception {
                Log.d(TAG, "==================doOnDispose Action");
            }
    })
    .subscribe(new Observer<Integer>() {
        private Disposable d;
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "==================onSubscribe ");
            this.d = d;
        }

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "==================onNext " + integer);
            d.dispose();
        }

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

        @Override
```

```
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }

});
```

打印结果：

```
05-23 10:20:36.345 23922-23922/? D/chan:
=================doOnLifecycle accept
=================onSubscribe
=================onNext 1
=================doOnDispose Action
=================doOnLifecycle Action
```

可以看到当在 onNext() 方法进行取消订阅操作后，doOnDispose() 和 doOnLifecycle() 都会被回调。

如果使用 doOnLifecycle 进行取消订阅，来看看打印结果：

```
05-23 10:32:20.014 24652-24652/com.example.rxjavademo D/chan:
=================doOnLifecycle accept
=================onSubscribe
```

可以发现 doOnDispose Action 和 doOnLifecycle Action 都没有被回调。

4.10 doOnTerminate() & doAfterTerminate()

方法预览:

```
public final Observable<T> doOnTerminate(final Action onTerminate)
public final Observable<T> doAfterTerminate(Action onFinally)
```

有什么用?

doOnTerminate 是在 onError 或者 onComplete 发送之前回调，而 doAfterTerminate 则是 onError 或者 onComplete 发送之后回调。

怎么用?

```
Observable.create(new ObservableOnSubscribe<Integer>() {
```

```
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
//      e.onError(new NullPointerException());
        e.onComplete();
    }
})
.doOnTerminate(new Action() {
    @Override
    public void run() throws Exception {
        Log.d(TAG, "==================doOnTerminate ");
    }
})
.subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }

});
```

打印结果：

```
05-23 10:00:39.503 22398-22398/com.example.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
```

```
==================onNext 2
05-23 10:00:39.504 22398-22398/com.example.rxjavademo D/chan:
==================onNext 3
==================doOnTerminate
==================onComplete
```

doAfterTerminate 也是差不多，这里就不再赘述。

4.11 doFinally()

方法预览:

```
public final Observable<T> doFinally(Action onFinally)
```

有什么用?

在所有事件发送完毕之后回调该方法。

怎么用?

这里可能你会有个问题，那就是 doFinally() 和 doAfterTerminate() 到底有什么区别？区别就是在于取消订阅，如果取消订阅之后 doAfterTerminate() 就不会被回调，而 doFinally() 无论怎么样都会被回调，且都会在事件序列的最后。

现在用以下例子说明下：

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.doFinally(new Action() {
    @Override
    public void run() throws Exception {
        Log.d(TAG, "==================doFinally ");
```

```
    }
})
.doOnDispose(new Action() {
    @Override
    public void run() throws Exception {
        Log.d(TAG, "==================doOnDispose ");
    }
})
.doAfterTerminate(new Action() {
    @Override
    public void run() throws Exception {
        Log.d(TAG, "==================doAfterTerminate ");
    }
})
.subscribe(new Observer<Integer>() {
    private Disposable d;
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
        this.d = d;
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
        d.dispose();
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 10:10:10.469 23196-23196/? D/chan: ==================onSubscribe
05-23 10:10:10.470 23196-23196/? D/chan: ==================onNext 1
==================doOnDispose
```

==========doFinally

可以看到如果调用了 dispose() 方法，doAfterTerminate() 不会被回调。

现在试试把 dispose() 注释掉看看，看看打印结果：

```
05-23 10:13:34.537 23439-23439/com.example.rxjavademo D/chan:
==========onSubscribe
==========onNext 1
==========onNext 2
==========onNext 3
==========onComplete
==========doAfterTerminate
==========doFinally
```

doAfterTerminate() 已经成功回调，doFinally() 还是会在事件序列的最后。

4.12 onErrorReturn()

方法预览:

```
public final Observable<T> onErrorReturn(Function<? super Throwable, ?
extends T> valueSupplier)
```

有什么用?

当接受到一个 onError() 事件之后回调，返回的值会回调 onNext() 方法，并正常结束该事件序列。

怎么用?

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onError(new NullPointerException());
    }
})
```

```
.onErrorReturn(new Function<Throwable, Integer>() {
    @Override
    public Integer apply(Throwable throwable) throws Exception {
        Log.d(TAG, "==================onErrorReturn " + throwable);
        return 404;
    }
})
.subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 18:35:18.175 19239-19239/? D/chan: ==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onErrorReturn java.lang.NullPointerException
==================onNext 404
==================onComplete
```

4.13 onErrorResumeNext()

方法预览:

```
public final Observable<T> onErrorResumeNext(Function<? super
Throwable, ? extends ObservableSource<? extends T>> resumeFunction)
```

有什么用?

当接收到 onError() 事件时,返回一个新的 Observable,并正常结束事件序列。

怎么用?

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onError(new NullPointerException());
    }
})
.onErrorResumeNext(new Function<Throwable, ObservableSource<? extends
Integer>>() {
    @Override
    public ObservableSource<? extends Integer> apply(Throwable
throwable) throws Exception {
        Log.d(TAG, "==================onErrorResumeNext " + throwable);
        return Observable.just(4, 5, 6);
    }
})
.subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
```

```
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 18:43:10.910 26469-26469/? D/chan: ==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onErrorResumeNext java.lang.NullPointerException
==================onNext 4
==================onNext 5
==================onNext 6
==================onComplete
```

4.14 onExceptionResumeNext()

方法预览:

```
public final Observable<T> onExceptionResumeNext(final
ObservableSource<? extends T> next)
```

有什么用?

与 onErrorResumeNext() 作用基本一致，但是这个方法只能捕捉 Exception。

怎么用?

先来试试 onExceptionResumeNext() 是否能捕捉 Error。

```
Observable.create(new ObservableOnSubscribe<Integer>() {
```

```
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onError(new Error("404"));
    }
})
.onExceptionResumeNext(new Observable<Integer>() {
    @Override
    protected void subscribeActual(Observer<? super Integer> observer)
{
        observer.onNext(333);
        observer.onComplete();
    }
})
.subscribe(new Observer<Integer>() {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 22:23:08.873 1062-1062/com.example.louder.rxjavademo D/chan:
==================onSubscribe
```

```
05-23 22:23:08.874 1062-1062/com.example.louder.rxjavademo D/chan:
==================onNext 1
==================onNext 2
==================onNext 3
==================onError
```

从打印结果可以知道,观察者收到 onError() 事件,证明 onErrorResumeNext() 不能捕捉 Error 事件。

将被观察者的 e.onError(new Error("404")) 改为 e.onError(new Exception("404")), 现在看看是否能捕捉 Exception 事件:

```
05-23 22:32:14.563 10487-10487/com.example.louder.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onNext 333
==================onComplete
```

从打印结果可以知道，这个方法成功捕获 Exception 事件。

4.15 retry()

方法预览:

```
public final Observable<T> retry(long times)
......
```

有什么用?

如果出现错误事件，则会重新发送所有事件序列。times 是代表重新发的次数。

怎么用?

```
Observable.create(new ObservableOnSubscribe<Integer>() {
    @Override
    public void subscribe(ObservableEmitter<Integer> e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
```

```
            e.onNext(3);
            e.onError(new Exception("404"));
        }
    })
    .retry(2)
    .subscribe(new Observer<Integer>() {
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "==================onSubscribe ");
        }

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "==================onNext " + integer);
        }

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

        @Override
        public void onComplete() {
            Log.d(TAG, "==================onComplete ");
        }
    });
```

打印结果：

```
05-23 22:46:18.537 22239-22239/com.example.louder.rxjavademo D/chan:
==================onSubscribe
05-23 22:46:18.538 22239-22239/com.example.louder.rxjavademo D/chan:
==================onNext 1
==================onNext 2
==================onNext 3
==================onNext 1
==================onNext 2
==================onNext 3
==================onNext 1
==================onNext 2
==================onNext 3
==================onError
```

4.16 retryUntil()

方法预览:

```
public final Observable<T> retryUntil(final BooleanSupplier stop)
```

有什么用?

出现错误事件之后，可以通过此方法判断是否继续发送事件。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onError(new Exception("404"));
    }
})
.retryUntil(new BooleanSupplier() {
    @Override
    public boolean getAsBoolean() throws Exception {
        if (i == 6) {
            return true;
        }
        return false;
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "==================onNext " + integer);
```

```
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-23 22:57:32.905 23063-23063/com.example.louder.rxjavademo D/chan:
==================onSubscribe
05-23 22:57:32.906 23063-23063/com.example.louder.rxjavademo D/chan:
==================onNext 1
==================onNext 2
==================onNext 3
==================onError
```

4.17 retryWhen()

方法预览:

```
public final void safeSubscribe(Observer<? super T> s)
```

有什么用?

当被观察者接收到异常或者错误事件时会回调该方法，这个方法会返回一个新的被观察者。如果返回的被观察者发送 Error 事件则之前的被观察者不会继续发送事件，如果发送正常事件则之前的被观察者会继续不断重试发送事件。

怎么用?

```
Observable.create(new ObservableOnSubscribe < String > () {
    @Override
    public void subscribe(ObservableEmitter < String > e) throws
Exception {
```

```
        e.onNext("chan");
        e.onNext("ze");
        e.onNext("de");
        e.onError(new Exception("404"));
        e.onNext("haha");
    }
})
.retryWhen(new Function<Observable<Throwable>, ObservableSource<?>>
() {
    @Override
    public ObservableSource <? > apply(Observable < Throwable >
throwableObservable) throws Exception {
        return throwableObservable.flatMap(new Function < Throwable,
ObservableSource <? >> () {
            @Override
            public ObservableSource <? > apply(Throwable throwable)
throws Exception {
                if(!throwable.toString().equals("java.lang.Exception:
404")) {
                    return Observable.just("可以忽略的异常");
                } else {
                    return Observable.error(new Throwable("终止啦"));
                }
            }
        });
    }
})
.subscribe(new Observer < String > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(String s) {
        Log.d(TAG, "==================onNext " + s);
    }

    @Override
    public void onError(Throwable e) {
        Log.d(TAG, "==================onError " + e.toString());
    }

    @Override
```

```
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }
});
```

打印结果：

```
05-24 09:13:25.622 28372-28372/com.example.rxjavademo D/chan:
=================onSubscribe
05-24 09:13:25.623 28372-28372/com.example.rxjavademo D/chan:
=================onNext chan
=================onNext ze
=================onNext de
05-24 09:13:25.624 28372-28372/com.example.rxjavademo D/chan:
=================onError java.lang.Throwable: 终止啦
```

将 onError(new Exception("404")) 改为 onError(new Exception("303")) 看看打印结果：

```
=================onNext chan
05-24 09:54:08.653 29694-29694/? D/chan: =================onNext ze
=================onNext de
=================onNext chan
=================onNext ze
=================onNext de
=================onNext chan
=================onNext ze
=================onNext de
=================onNext chan
=================onNext ze
=================onNext de
=================onNext chan
=================onNext ze
=================onNext de
=================onNext chan
.....
```

从结果可以看出，会不断重复发送消息。

4.18 repeat()

方法预览:

```
public final Observable<T> repeat(long times)
......
```

有什么用?

重复发送被观察者的事件，times 为发送次数。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.repeat(2)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

    @Override
    public void onError(Throwable e) {

    }

    @Override
```

```
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-24 11:33:29.565 8544-8544/com.example.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onNext 1
==================onNext 2
==================onNext 3
05-24 11:33:29.565 8544-8544/com.example.rxjavademo D/chan:
==================onComplete
```

从结果可以看出，该事件发送了两次。

4.19 repeatWhen()

方法预览:

```
public final Observable<T> repeatWhen(final Function<? super
Observable<Object>, ? extends ObservableSource<?>> handler)
```

有什么用?

这个方法可以会返回一个新的被观察者设定一定逻辑来决定是否重复发送事件。

怎么用?

这里分三种情况，如果新的被观察者返回 onComplete 或者 onError 事件，则旧的被观察者不会继续发送事件。如果被观察者返回其他事件，则会重复发送事件。

现在试验发送 onComplete 事件，代码如下：

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
```

```
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.repeatWhen(new Function < Observable < Object > , ObservableSource <? >>
() {
    @Override
    public ObservableSource <? > apply(Observable < Object >
objectObservable) throws Exception {
        return Observable.empty();
    //  return Observable.error(new Exception("404"));
    //  return Observable.just(4); null;
    }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-24 11:44:33.486 9379-9379/com.example.rxjavademo D/chan:
==================onSubscribe
```

```
05-24 11:44:33.487 9379-9379/com.example.rxjavademo D/chan:
==================onComplete
```

下面直接看看发送 onError 事件和其他事件的打印结果。

发送 onError 打印结果：

```
05-24 11:46:29.507 9561-9561/com.example.rxjavademo D/chan:
==================onSubscribe
05-24 11:46:29.508 9561-9561/com.example.rxjavademo D/chan:
==================onError
```

发送其他事件的打印结果：

```
05-24 11:48:35.844 9752-9752/com.example.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onComplete
```

4.20 subscribeOn()

方法预览:

```
public final Observable<T> subscribeOn(Scheduler scheduler)
```

有什么用?

指定被观察者的线程，要注意的时，如果多次调用此方法，只有第一次有效。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {
    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        Log.d(TAG, "=======================currentThread name: " +
Thread.currentThread().getName());
        e.onNext(1);
        e.onNext(2);
```

```
            e.onNext(3);
            e.onComplete();
        }
    })
    //.subscribeOn(Schedulers.newThread())
    .subscribe(new Observer < Integer > () {
        @Override
        public void onSubscribe(Disposable d) {
            Log.d(TAG, "======================onSubscribe");
        }

        @Override
        public void onNext(Integer integer) {
            Log.d(TAG, "======================onNext " + integer);
        }

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

        @Override
        public void onComplete() {
            Log.d(TAG, "======================onComplete");
        }
    });
```

现在不调用 subscribeOn() 方法，来看看打印结果：

```
05-26 10:40:42.246 21466-21466/? D/chan:
======================onSubscribe
05-26 10:40:42.247 21466-21466/? D/chan:
========================currentThread name: main
======================onNext 1
======================onNext 2
======================onNext 3
======================onComplete
```

可以看到打印被观察者的线程名字是主线程。

接着调用 subscribeOn(Schedulers.newThread()) 来看看打印结果：

```
05-26 10:43:26.964 22530-22530/com.example.rxjavademo D/chan:
======================onSubscribe
```

```
05-26 10:43:26.966 22530-22569/com.example.rxjavademo D/chan:
=======================currentThread name: RxNewThreadScheduler-1
05-26 10:43:26.967 22530-22569/com.example.rxjavademo D/chan:
====================onNext 1
====================onNext 2
====================onNext 3
====================onComplete
```

可以看到打印结果被观察者是在一条新的线程。

现在看看多次调用会不会有效，代码如下：

```
Observable.create(new ObservableOnSubscribe < Integer > () {

    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        Log.d(TAG, "=======================currentThread name: " +
Thread.currentThread().getName());
        e.onNext(1);
        e.onNext(2);
        e.onNext(3);
        e.onComplete();
    }
})
.subscribeOn(Schedulers.computation())
.subscribeOn(Schedulers.newThread())
.subscribe(new Observer < Integer > () {@Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "====================onSubscribe");
    }

    @Override
    public void onNext(Integer integer) {
        Log.d(TAG, "====================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "====================onComplete");
```

```
    }
});
```

打印结果：

```
05-26 10:47:20.925 23590-23590/com.example.rxjavademo D/chan:
======================onSubscribe
05-26 10:47:20.930 23590-23629/com.example.rxjavademo D/chan:
========================currentThread name: RxComputationThreadPool-1
======================onNext 1
======================onNext 2
======================onNext 3
======================onComplete
```

可以看到第二次调动的 subscribeOn(Schedulers.newThread()) 并没有效果。

4.21 observeOn()

方法预览:

```
public final Observable<T> observeOn(Scheduler scheduler)
```

有什么用?

指定观察者的线程，每指定一次就会生效一次。

怎么用?

```
Observable.just(1, 2, 3)
.observeOn(Schedulers.newThread())
.flatMap(new Function < Integer, ObservableSource < String >> () {
    @Override
    public ObservableSource < String > apply(Integer integer) throws
Exception {
        Log.d(TAG, "====================flatMap Thread name " +
Thread.currentThread().getName());
        return Observable.just("chan" + integer);
    }
})
.observeOn(AndroidSchedulers.mainThread())
.subscribe(new Observer < String > () {
```

```
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "=====================onSubscribe");
    }

    @Override
    public void onNext(String s) {
        Log.d(TAG, "=====================onNext Thread name " +
Thread.currentThread().getName());
        Log.d(TAG, "=====================onNext " + s);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "=====================onComplete");
    }
});
```

打印结果：

```
05-26 10:58:04.593 25717-25717/com.example.rxjavademo D/chan:
=====================onSubscribe
05-26 10:58:04.594 25717-25753/com.example.rxjavademo D/chan:
=====================flatMap Thread name RxNewThreadScheduler-1
05-26 10:58:04.595 25717-25753/com.example.rxjavademo D/chan:
=====================flatMap Thread name RxNewThreadScheduler-1
=====================flatMap Thread name RxNewThreadScheduler-1
05-26 10:58:04.617 25717-25717/com.example.rxjavademo D/chan:
=====================onNext Thread name main
=====================onNext chan1
=====================onNext Thread name main
=====================onNext chan2
=====================onNext Thread name main
=====================onNext chan3
05-26 10:58:04.618 25717-25717/com.example.rxjavademo D/chan:
=====================onComplete
```

从打印结果可以知道，observeOn 成功切换了线程。

下表总结了 RxJava 中的调度器：

调度器	作用
Schedulers.computation( )	用于使用计算任务，如事件循环和回调处理
Schedulers.immediate( )	当前线程
Schedulers.io( )	用于 IO 密集型任务，如果异步阻塞 IO 操作。
Schedulers.newThread( )	创建一个新的线程
AndroidSchedulers.mainThread()	Android 的 UI 线程，用于操作 UI。

5. 过滤操作符

5.1 filter()

方法预览:

```
public final Observable<T> filter(Predicate<? super T> predicate)
```

有什么用?

通过一定逻辑来过滤被观察者发送的事件，如果返回 true 则会发送事件，否则不会发送。

怎么用?

```
 Observable.just(1, 2, 3)
    .filter(new Predicate < Integer > () {
        @Override
        public boolean test(Integer integer) throws Exception {
            return integer < 2;
        }
})
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "=================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "=================onNext " + integer);
```

```
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

以上代码只有小于 2 的事件才会发送，来看看打印结果：

```
05-24 22:57:32.562 12776-12776/com.example.louder.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================onComplete
```

5.2 ofType()

方法预览:

```
public final <U> Observable<U> ofType(final Class<U> clazz)
```

有什么用?

可以过滤不符合该类型事件

怎么用?

```
Observable.just(1, 2, 3, "chan", "zhide")
.ofType(Integer.class)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
```

```
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "=================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "=================onComplete ");
    }
});
```

打印结果：

```
05-24 23:04:24.752 13229-13229/? D/chan: =================onSubscribe
=================onNext 1
=================onNext 2
=================onNext 3
05-24 23:04:24.753 13229-13229/? D/chan: =================onComplete
```

5.3 skip()

方法预览:

```
public final Observable<T> skip(long count)
.......
```

有什么用?

跳过正序某些事件，count 代表跳过事件的数量

怎么用?

```
Observable.just(1, 2, 3)
.skip(2)
.subscribe(new Observer < Integer > () {
    @Override
```

```
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-24 23:13:50.448 13831-13831/? D/chan: ==================onSubscribe
05-24 23:13:50.449 13831-13831/? D/chan: ==================onNext 3
==================onComplete
```

skipLast() 作用也是跳过某些事件，不过它是用来跳过正序的后面的事件，这里就不再讲解了。

5.4 distinct()

方法预览:

```
public final Observable<T> distinct()
```

有什么用?

过滤事件序列中的重复事件。

怎么用?

```
Observable.just(1, 2, 3, 3, 2, 1)
.distinct()
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果:

```
05-24 23:19:44.334 14206-14206/com.example.louder.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onComplete
```

5.5 distinctUntilChanged()

方法预览:

```
public final Observable<T> distinctUntilChanged()
```

有什么用?

过滤掉连续重复的事件

怎么用?

```
Observable.just(1, 2, 3, 3, 2, 1)
.distinctUntilChanged()
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果:

```
05-24 23:22:35.985 14424-14424/com.example.louder.rxjavademo D/chan:
==================onSubscribe
==================onNext 1
==================onNext 2
==================onNext 3
==================onNext 2
==================onNext 1
==================onComplete
```

因为事件序列中连续出现两次 3，所以第二次 3 并不会发出。

5.6 take()

方法预览:

```
public final Observable<T> take(long count)
......
```

有什么用?

控制观察者接收的事件的数量。

怎么用?

```
Observable.just(1, 2, 3, 4, 5)
.take(3)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
        i += integer;
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果:

```
05-24 23:28:32.899 14704-14704/? D/chan: ==================onSubscribe
```

```
==================onNext 1
==================onNext 2
==================onNext 3
==================onComplete
```

takeLast() 的作用就是控制观察者只能接受事件序列的后面几件事情，这里就不再讲解了，大家可以自己试试。

5.7 debounce()

方法预览:

```
public final Observable<T> debounce(long timeout, TimeUnit unit)
......
```

有什么用?

如果两件事件发送的时间间隔小于设定的时间间隔则前一件事件就不会发送给观察者。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {

    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onNext(1);
        Thread.sleep(900);
        e.onNext(2);
    }
})
.debounce(1, TimeUnit.SECONDS)
.subscribe(new Observer < Integer > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "==================onSubscribe ");
    }

    @Override
    public void onNext(Integer integer) {
```

```
        Log.d(TAG, "==================onNext " + integer);
    }

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

    @Override
    public void onComplete() {
        Log.d(TAG, "==================onComplete ");
    }
});
```

打印结果：

```
05-25 20:39:10.512 17441-17441/com.example.rxjavademo D/chan:
==================onSubscribe
05-25 20:39:12.413 17441-17478/com.example.rxjavademo D/chan:
==================onNext 2
```

可以看到事件 1 并没有发送出去，现在将间隔时间改为 1000，看看打印结果：

```
05-25 20:42:10.874 18196-18196/com.example.rxjavademo D/chan:
==================onSubscribe
05-25 20:42:11.875 18196-18245/com.example.rxjavademo D/chan:
==================onNext 1
05-25 20:42:12.875 18196-18245/com.example.rxjavademo D/chan:
==================onNext 2
```

throttleWithTimeout() 与此方法的作用一样，这里就不再赘述了。

5.8 firstElement() && lastElement()

方法预览：

```
public final Maybe<T> firstElement()
public final Maybe<T> lastElement()
```

有什么用?

firstElement() 取事件序列的第一个元素，lastElement() 取事件序列的最后一个元素。

怎么用?

```
Observable.just(1, 2, 3, 4)
.firstElement()
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "==================firstElement " + integer);
    }
});

Observable.just(1, 2, 3, 4)
.lastElement()
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "==================lastElement " + integer);
    }
});
```

打印结果：

```
05-25 20:47:22.189 19909-19909/? D/chan:
==================firstElement 1
==================lastElement 4
```

5.9 elementAt() & elementAtOrError()

方法预览:

```
public final Maybe<T> elementAt(long index)
public final Single<T> elementAtOrError(long index)
```

有什么用?

elementAt() 可以指定取出事件序列中事件，但是输入的 index 超出事件序列的总数的话就不会出现任何结果。这种情况下，你想发出异常信息的话就用 elementAtOrError() 。

怎么用?

```
Observable.just(1, 2, 3, 4)
.elementAt(0)
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "====================accept " + integer);
    }
});
```

打印结果：

```
05-25 20:56:22.266 23346-23346/com.example.rxjavademo D/chan:
====================accept 1
```

将 elementAt() 的值改为 5，这时是没有打印结果的，因为没有满足条件的元素。

替换 elementAt() 为 elementAtOrError()，代码如下：

```
Observable.just(1, 2, 3, 4)
.elementAtOrError(5)
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "====================accept " + integer);
    }
});
```

打印结果：

```
io.reactivex.exceptions.OnErrorNotImplementedException
at
io.reactivex.internal.functions.Functions$OnErrorMissingConsumer.acce
pt(Functions.java: 704)
at
io.reactivex.internal.functions.Functions$OnErrorMissingConsumer.acce
pt(Functions.java: 701)
at
io.reactivex.internal.observers.ConsumerSingleObserver.onError(Consum
erSingleObserver.java: 47)
```

```
at
io.reactivex.internal.operators.observable.ObservableElementAtSingle$
ElementAtObserver.onComplete(ObservableElementAtSingle.java: 117)
at
io.reactivex.internal.operators.observable.ObservableFromArray$FromAr
rayDisposable.run(ObservableFromArray.java: 110)
at
io.reactivex.internal.operators.observable.ObservableFromArray.subscr
ibeActual(ObservableFromArray.java: 36)
at io.reactivex.Observable.subscribe(Observable.java: 10903)
at
io.reactivex.internal.operators.observable.ObservableElementAtSingle.
subscribeActual(ObservableElementAtSingle.java: 37)
at io.reactivex.Single.subscribe(Single.java: 2707)
at io.reactivex.Single.subscribe(Single.java: 2693)
at io.reactivex.Single.subscribe(Single.java: 2664)
at com.example.rxjavademo.MainActivity.onCreate(MainActivity.java:
103)
at android.app.Activity.performCreate(Activity.java: 6942)
at
android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:
1126)
at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:
2880)
at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:
2988)
at android.app.ActivityThread. - wrap14(ActivityThread.java)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:
1631)
at android.os.Handler.dispatchMessage(Handler.java: 102)
at android.os.Looper.loop(Looper.java: 154)
at android.app.ActivityThread.main(ActivityThread.java: 6682)
at java.lang.reflect.Method.invoke(Native Method)
at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit
.java: 1520)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java: 1410)
Caused by: java.util.NoSuchElementException
at
io.reactivex.internal.operators.observable.ObservableElementAtSingle$
ElementAtObserver.onComplete(ObservableElementAtSingle.java: 117)
```

```
at
io.reactivex.internal.operators.observable.ObservableFromArray$FromAr
rayDisposable.run(ObservableFromArray.java: 110)
at
io.reactivex.internal.operators.observable.ObservableFromArray.subscr
ibeActual(ObservableFromArray.java: 36)
at io.reactivex.Observable.subscribe(Observable.java: 10903)
at
io.reactivex.internal.operators.observable.ObservableElementAtSingle.
subscribeActual(ObservableElementAtSingle.java: 37)
at io.reactivex.Single.subscribe(Single.java: 2707)
at io.reactivex.Single.subscribe(Single.java: 2693)
at io.reactivex.Single.subscribe(Single.java: 2664)
at com.example.rxjavademo.MainActivity.onCreate(MainActivity.java:
103)
at android.app.Activity.performCreate(Activity.java: 6942)
at
android.app.Instrumentation.callActivityOnCreate(Instrumentation.java:
1126)
at
android.app.ActivityThread.performLaunchActivity(ActivityThread.java:
2880)
at
android.app.ActivityThread.handleLaunchActivity(ActivityThread.java:
2988)
at android.app.ActivityThread. - wrap14(ActivityThread.java)
at android.app.ActivityThread$H.handleMessage(ActivityThread.java:
1631)
at android.os.Handler.dispatchMessage(Handler.java: 102)
at android.os.Looper.loop(Looper.java: 154)
at android.app.ActivityThread.main(ActivityThread.java: 6682)
at java.lang.reflect.Method.invoke(Native Method)
at
com.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit
.java: 1520)
at com.android.internal.os.ZygoteInit.main(ZygoteInit.java: 1410)
```

这时候会抛出 NoSuchElementException 异常。

6. 条件操作符

6.1 all()

方法预览:

```
public final Observable<T> ambWith(ObservableSource<? extends T> other)
```

有什么用?

判断事件序列是否全部满足某个事件，如果都满足则返回 true，反之则返回 false。

怎么用?

```
Observable.just(1, 2, 3, 4)
.all(new Predicate < Integer > () {
    @Override
    public boolean test(Integer integer) throws Exception {
        return integer < 5;
    }
})
.subscribe(new Consumer < Boolean > () {
    @Override
    public void accept(Boolean aBoolean) throws Exception {
        Log.d(TAG, "==================aBoolean " + aBoolean);
    }
});
```

打印结果：

```
05-26 09:39:51.644 1482-1482/com.example.rxjavademo D/chan:
==================aBoolean true
```

6.2 takeWhile()

方法预览:

```
public final Observable<T> takeWhile(Predicate<? super T> predicate)
```

有什么用?

可以设置条件，当某个数据满足条件时就会发送该数据，反之则不发送。

怎么用?

```
Observable.just(1, 2, 3, 4)
.takeWhile(new Predicate < Integer > () {
    @Override
    public boolean test(Integer integer) throws Exception {
        return integer < 3;
    }
})
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "======================integer " + integer);
    }
});
```

打印结果：

```
05-26 09:43:14.634 3648-3648/com.example.rxjavademo D/chan:
======================integer 1
======================integer 2
```

6.3 skipWhile()

方法预览:

```
public final Observable<T> skipWhile(Predicate<? super T> predicate)
```

有什么用?

可以设置条件，当某个数据满足条件时不发送该数据，反之则发送。

怎么用?

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

```
.skipWhile(new Predicate < Integer > () {
    @Override
    public boolean test(Integer integer) throws Exception {
        return integer < 3;
    }
})
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "=======================integer " + integer);
    }
});
```

打印结果：

```
05-26 09:47:32.653 4861-4861/com.example.rxjavademo D/chan:
=======================integer 3
=======================integer 4
```

6.4 takeUntil()

方法预览:

```
public final Observable<T> takeUntil(Predicate<? super T> stopPredicate
```

有什么用?

可以设置条件，当事件满足此条件时，下一次的事件就不会被发送了。

怎么用?

```
Observable.just(1, 2, 3, 4, 5, 6)
.takeUntil(new Predicate < Integer > () {
    @Override
    public boolean test(Integer integer) throws Exception {
        return integer > 3;
    }
})
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
```

```
        Log.d(TAG, "======================integer " + integer);
    }
});
```

打印结果：

```
05-26 09:55:12.918 7933-7933/com.example.rxjavademo D/chan:
======================integer 1
======================integer 2
05-26 09:55:12.919 7933-7933/com.example.rxjavademo D/chan:
======================integer 3
======================integer 4
```

6.5 skipUntil()

方法预览:

```
public final <U> Observable<T> skipUntil(ObservableSource<U> other)
```

有什么用?

当 skipUntil() 中的 Observable 发送事件了，原来的 Observable 才会发送事件给观察者。

怎么用?

```
Observable.intervalRange(1, 5, 0, 1, TimeUnit.SECONDS)
.skipUntil(Observable.intervalRange(6, 5, 3, 1, TimeUnit.SECONDS))
.subscribe(new Observer < Long > () {
    @Override
    public void onSubscribe(Disposable d) {
        Log.d(TAG, "======================onSubscribe ");
    }

    @Override
    public void onNext(Long along) {
        Log.d(TAG, "======================onNext " + along);
    }

    @Override
    public void onError(Throwable e) {
```

```
        Log.d(TAG, "======================onError ");
    }

    @Override
    public void onComplete() {
        Log.d(TAG, "======================onComplete ");
    }
});
```

打印结果：

```
05-26 10:08:50.574 13023-13023/com.example.rxjavademo D/chan:
======================onSubscribe
05-26 10:08:53.576 13023-13054/com.example.rxjavademo D/chan:
======================onNext 4
05-26 10:08:54.576 13023-13054/com.example.rxjavademo D/chan:
======================onNext 5
======================onComplete
```

从结果可以看出，skipUntil() 里的 Observable 并不会发送事件给观察者。

6.6 sequenceEqual()

方法预览:

```
public static <T> Single<Boolean> sequenceEqual(ObservableSource<?
extends T> source1, ObservableSource<? extends T> source2)
......
```

有什么用?

判断两个 Observable 发送的事件是否相同。

怎么用?

```
Observable.sequenceEqual(Observable.just(1, 2, 3),
Observable.just(1, 2, 3))
.subscribe(new Consumer < Boolean > () {
    @Override
    public void accept(Boolean aBoolean) throws Exception {
        Log.d(TAG, "======================onNext " + aBoolean);
```

```
    }
});
```

打印结果：

```
05-26 10:11:45.975 14157-14157/? D/chan: ======================onNext true
```

6.7 contains()

方法预览:

```
public final Single<Boolean> contains(final Object element)
```

有什么用?

判断事件序列中是否含有某个元素,如果有则返回 true,如果没有则返回 false。

怎么用?

```
Observable.just(1, 2, 3)
.contains(3)
.subscribe(new Consumer < Boolean > () {
    @Override
    public void accept(Boolean aBoolean) throws Exception {
        Log.d(TAG, "======================onNext " + aBoolean);
    }
});
```

打印结果：

```
05-26 10:14:23.522 15085-15085/com.example.rxjavademo D/chan: ======================onNext true
```

6.8 isEmpty()

方法预览:

```
public final Single<Boolean> isEmpty()
```

有什么用?

判断事件序列是否为空。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {

    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onComplete();
    }
})
.isEmpty()
.subscribe(new Consumer < Boolean > () {
    @Override
    public void accept(Boolean aBoolean) throws Exception {
        Log.d(TAG, "======================onNext " + aBoolean);
    }
});
```

打印结果:

```
05-26 10:17:16.725 16109-16109/com.example.rxjavademo D/chan:
======================onNext true
```

6.9 amb()

方法预览:

```
public static <T> Observable<T> amb(Iterable<? extends ObservableSource<?
extends T>> sources)
```

有什么用?

amb() 要传入一个 Observable 集合，但是只会发送最先发送事件的 Observable 中的事件，其余 Observable 将会被丢弃。

怎么用?

```
ArrayList < Observable < Long >> list = new ArrayList < > ();

list.add(Observable.intervalRange(1, 5, 2, 1, TimeUnit.SECONDS));
list.add(Observable.intervalRange(6, 5, 0, 1, TimeUnit.SECONDS));

Observable.amb(list)
.subscribe(new Consumer < Long > () {
    @Override
    public void accept(Long aLong) throws Exception {
        Log.d(TAG, "======================aLong " + aLong);
    }
});
```

打印结果：

```
05-26 10:21:29.580 17185-17219/com.example.rxjavademo D/chan:
======================aLong 6
05-26 10:21:30.580 17185-17219/com.example.rxjavademo D/chan:
======================aLong 7
05-26 10:21:31.579 17185-17219/com.example.rxjavademo D/chan:
======================aLong 8
05-26 10:21:32.579 17185-17219/com.example.rxjavademo D/chan:
======================aLong 9
05-26 10:21:33.579 17185-17219/com.example.rxjavademo D/chan:
======================aLong 10
```

6.10 defaultIfEmpty()

方法预览:

```
public final Observable<T> defaultIfEmpty(T defaultItem)
```

有什么用?

如果观察者只发送一个 onComplete() 事件，则可以利用这个方法发送一个值。

怎么用?

```
Observable.create(new ObservableOnSubscribe < Integer > () {

    @Override
    public void subscribe(ObservableEmitter < Integer > e) throws
Exception {
        e.onComplete();
    }
})
.defaultIfEmpty(666)
.subscribe(new Consumer < Integer > () {
    @Override
    public void accept(Integer integer) throws Exception {
        Log.d(TAG, "======================onNext " + integer);
    }
});
```

打印结果：

```
05-26 10:26:56.376 19249-19249/com.example.rxjavademo D/chan:
======================onNext 666
```

RxJava 常见的使用方式都已经介绍的差不多，相信大家如果都掌握这些操作符的用法的话，那么使用 RxJava 将不会再是难题了。