**Rx操作符**

**RxJava 到底是什么**

一个词：异步

异步操作很关键的一点是程序的简洁性，因为在调度过程比较复杂的情况下，异步代码经常会既难写也难被读懂。 Android 创造的AsyncTask 和Handler ，其实都是为了让异步代码更加简洁。RxJava 的优势也是简洁，但它的简洁的与众不同之处在于，**随着程序逻辑变得越来越复杂，它依然能够保持简洁**

### API 介绍和原理简析

### RxJava 的异步实现，是通过一种扩展的观察者模式来实现的。

##### RxJava 的观察者模式

RxJava 有四个基本概念：

Observable (可观察者，即被观察者)

Observer (观察者)

subscribe (订阅)、事件

Observable 和Observer 通过 subscribe() 方法实现订阅关系，从而 Observable 可以在需要的时候发出事件来通知 Observer。

与传统观察者模式不同， RxJava 的事件回调方法除了普通事件 onNext() （相当于 onClick() / onEvent()）之外，还定义了两个特殊的事件：onCompleted() 和 onError()。onCompleted(): 事件队列完结。RxJava 不仅把每个事件单独处理，还会把它们看做一个队列。RxJava 规定，当不会再有新的onNext() 发出时，需要触发 onCompleted() 方法作为标志。

onError(): 事件队列异常。在事件处理过程中出异常时，onError() 会被触发，同时队列自动终止，不允许再有事件发出。

在一个正确运行的事件序列中, onCompleted() 和 onError() 有且只有一个，并且是事件序列中的最后一个。需要注意的是，onCompleted() 和 onError() 二者也是互斥的，即在队列中调用了其中一个，就不应该再调用另一个。

**2. 基本实现**

基于以上的概念， RxJava 的基本实现主要有三点：

**1) 创建 Observer**

Observer 即观察者，它决定事件触发的时候将有怎样的行为。 RxJava 中的 Observer 接口的实现方式：

Observer<String> observer = new Observer<String>() {

@Override

public void onNext(String s) {

Log.d(tag, "Item: " + s);

}

@Override

public void onCompleted() {

Log.d(tag, "Completed!");

}

@Override

public void onError(Throwable e) {

Log.d(tag, "Error!");

}

};

除了 Observer 接口之外，RxJava 还内置了一个实现了 Observer 的抽象类：Subscriber。 Subscriber 对 Observer 接口进行了一些扩展，但他们的基本使用方式是完全一样的：

Subscriber<String> subscriber = new Subscriber<String>() {

@Override

public void onNext(String s) {

Log.d(tag, "Item: " + s);

}

@Override

public void onCompleted() {

Log.d(tag, "Completed!");

}

@Override

public void onError(Throwable e) {

Log.d(tag, "Error!");

}

### };

### 不仅基本使用方式一样，实质上，在 RxJava 的 subscribe 过程中，Observer 也总是会先被转换成一个 Subscriber 再使用。所以如果你只想使用基本功能，选择 Observer 和 Subscriber 是完全一样的。它们的区别对于使用者来说主要有两点：

1. onStart(): 这是 Subscriber 增加的方法。它会在 subscribe 刚开始，而事件还未发送之前被调用，可以用于做一些准备工作，例如数据的清零或重置。这是一个可选方法，默认情况下它的实现为空。需要注意的是，如果对准备工作的线程有要求（例如弹出一个显示进度的对话框，这必须在主线程执行）， onStart() 就不适用了，因为它总是在 subscribe 所发生的线程被调用，而不能指定线程。要在指定的线程来做准备工作，可以使用 doOnSubscribe() 方法，具体可以在后面的文中看到。
2. unsubscribe(): 这是 Subscriber 所实现的另一个接口 Subscription 的方法，用于取消订阅。在这个方法被调用后，Subscriber 将不再接收事件。一般在这个方法调用前，可以使用 isUnsubscribed() 先判断一下状态。 unsubscribe() 这个方法很重要，因为在subscribe() 之后， Observable 会持有 Subscriber 的引用，这个引用如果不能及时被释放，将有内存泄露的风险。所以最好保持一个原则：要在不再使用的时候尽快在合适的地方（例如 onPause() onStop() 等方法中）调用 unsubscribe() 来解除引用关系，以避免内存泄露的发生。

**2) 创建 Observable**

Observable 即被观察者，它决定什么时候触发事件以及触发怎样的事件。 RxJava 使用 create() 方法来创建一个 Observable ，并为它定义事件触发规则：

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

@Override

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

subscriber.onNext("Hello");

subscriber.onNext("Hi");

subscriber.onNext("Aloha");

subscriber.onCompleted();

}

### });

### 3) Subscribe (订阅)

创建了 Observable 和 Observer 之后，再用 subscribe() 方法将它们联结起来，整条链子就可以工作了。代码形式很简单：

observable.subscribe(observer);

// 或者：

observable.subscribe(subscriber);

**创建操作符**

### 基本创建

1. **create（）**

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

@Override

public void subscribe(ObservableEmitter<Integer> e) throws Exception {

e.onNext(1);

e.onNext(2);

e.onNext(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: "+e);

}

@Override

public void onComplete() {

Log.d(TAG, "onComplete: ");

}

});

### 快速创建

**1.just()**

* 作用:快速创建1个被观察者对象（Observable）
* 发送事件的特点：直接发送传入的事件
* 注意1:just最多只能发送9个参数
* 注意2：如果你传递null给Just，它会返回一个发射null值的Observable

Observable.just(1, 2, 3)

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "onSubscribe: "+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() {

Log.d(TAG, "onComplete: ");

}

});

2.fromArray（）

* 作用：快速创建一个被观察者对象
* 发送事件的特点：直接发送传入的数据数组
* fromArray会将数组中的数据转换为Observable对象
* 应用场景:被观察者对象（Observable） & 发送10个以上事件（数组形式),数组遍历

Integer[] items = { 0, 1, 2, 3, 4 };

Observable.fromArray(items)

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "onSubscribe: "+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() {

Log.d(TAG, "onComplete: ");

}

});

3.fromIterable（）

* 作用:快速创建1个被观察者对象（Observable）
* 发送事件的特点:直接发送 传入的集合List数据
* 应用场景:1.快速创建 被观察者对象（Observable） & 发送10个以上事件（集合形式）2.集合元素遍历

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

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: "+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() {

Log.d(TAG, "onComplete: ");

}

});

}

### 延迟创建

需求场景：

* 定时操作：在经过了x秒后，需要自动执行y操作
* 周期性操作：每隔x秒后，需要自动执行y操作

1.defer（）

* 作用:defer（）操作符会一直等待直到有观察者订阅它，然后它使用Observable工厂方法生成一个Observable。在某些情况下，等待直到最后一分钟（就是知道订阅发生时）才生成Observable可以确保Observable包含最新的数据。
* 使用场景：动态创建被观察者对象（Observable） & 获取最新的Observable对象数据

// 1. 第1次对i赋值

int i= 10;

Observable<Integer> observable = Observable.defer(new Callable<ObservableSource<? extends Integer>>() {

@Override

public ObservableSource<? extends Integer> call() throws Exception {

return Observable.just(i);

}

});

// 1. 第1次对i赋值

i = 15;

observable

.subscribe(new Observer<Object>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "onSubscribe: " + d);

}

@Override

public void onNext(Object o) {

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

}

@Override

public void onError(Throwable e) {

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

}

@Override

public void onComplete() {

Log.d(TAG, "onComplete: ");

}

});

2.timer（）

* 作用:创建一个Observable，它在一个给定的延迟后发射一个特殊的值。
* 应用:延迟指定事件，发送一个0，一般用于检测

// 该例子 = 延迟2s后，发送一个long类型数值

Observable.timer(2, TimeUnit.SECONDS).subscribe(new Observer<Long>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

}

@Override

public void onNext(Long value) {

Log.d(TAG, "接收到了事件" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

3.interval（）

* 作用 : 按固定的时间间隔发射一个无限递增的整数序列。
* interval(long,TimeUnit,Scheduler))
* 参数说明： 参数1 = 第1次延迟时间，参数2 = 间隔时间数字，参数3 = 时间单位；

Observable.interval(3, 1, TimeUnit.SECONDS)

// 该例子发送的事件序列特点：延迟3s后发送事件，每隔1秒产生1个数字（从0开始递增1，无限个）

.subscribe(new Observer<Long>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

}

@Override

public void onNext(Long value) {

Log.d(TAG, "接收到了事件" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

4.intervalRange（）

* 作用:发送事件的特点：每隔指定时间 就发送 事件，可指定发送的数据（从0开始、无限递增1的的整数）的数量

// 参数1 = 事件序列起始点；

// 参数2 = 事件数量；

// 参数3 = 第1次事件延迟发送时间；

// 参数4 = 间隔时间数字；

// 参数5 = 时间单位

Observable.intervalRange(3, 10, 2, 1, TimeUnit.SECONDS)

// 该例子发送的事件序列特点：

// 1. 从3开始，一共发送10个事件；

// 2. 第1次延迟2s发送，之后每隔2秒产生1个数字（从0开始递增1，无限个）

.subscribe(new Observer<Long>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

} // 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(Long value) {

Log.d(TAG, "接收到了事件" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

5.range（） /rangeLong（）

range（） 作用：连续发送 1个事件序列，可指定范围,作用类似于intervalRange（），但区别在于：无延迟发送事件

rangeLong（）类似于range（），区别在于该方法支持数据类型 = Long

// 参数说明：

// 参数1 = 事件序列起始点；

// 参数2 = 事件数量；

// 注：若设置为负数，则会抛出异常

Observable.range(3,10)

// 该例子发送的事件序列特点：从3开始发送，每次发送事件递增1，一共发送10个事件

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

} // 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(Integer value) {

Log.d(TAG, "接收到了事件" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

变换操作符

1.Map()

* 作用：对Observable发射的每一项数据应用一个函数，执行变换操作
* 应用场景：数据类型转化

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

@Override

public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

}

// 使用Map变换操作符中的Function函数对被观察者发送的事件进行统一变换：整型变换成字符串类型

}).map(new Function<Integer,String>() {

@Override

public String apply(Integer integer) throws Exception {

return "使用Map操作符 将事件"+integer+"的参数从 整型"+integer + " 变换成 字符串类型" + integer ;

}

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

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

} // 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(String value) {

Log.d(TAG, "接收到了事件" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

2.flatMap()

* 作用：将一个发射数据的Observable变换为多个Observables，然后将它们发射的数据合并后放进一个单独的Observable
* 应用场景:**无序**的将被观察者发送的整个事件序列进行变换

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

@Override

public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

}

// 使用Map变换操作符中的Function函数对被观察者发送的事件进行统一变换：整型变换成字符串类型

}).flatMap(new Function<Integer, ObservableSource<String>>() {

@Override

public ObservableSource<String> apply(Integer integer) throws Exception {

final List<String> list = new ArrayList<>();

for (int i = 0; i < 3; i++) {

list.add("我是事件 " + integer + "拆分后的子事件" + i);

// 通过flatMap中将被观察者生产的事件序列先进行拆分，再将每个事件转换为一个新的发送三个String事件

// 最终合并，再发送给被观察者

}

return Observable.fromIterable(list);

}

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

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

} // 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(String value) {

Log.d(TAG, "接收到了事件" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

3.ConcatMap()

* 作用类似与FlatMap（）操作符
* 区别：拆分 & 重新合并生成的事件序列 的顺序 = 被观察者旧序列生产的顺序
* 用于场景:有序的将被观察者发送的整个事件序列进行变换

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

@Override

public void subscribe(ObservableEmitter<Integer> emitter) throws Exception {

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

}

// 使用Map变换操作符中的Function函数对被观察者发送的事件进行统一变换：整型变换成字符串类型

}).concatMap(new Function<Integer, ObservableSource<String>>() {

@Override

public ObservableSource<String> apply(Integer integer) throws Exception {

final List<String> list = new ArrayList<>();

for (int i = 0; i < 3; i++) {

list.add("我是事件 " + integer + "拆分后的子事件" + i);

// 通过flatMap中将被观察者生产的事件序列先进行拆分，再将每个事件转换为一个新的发送三个String事件

// 最终合并，再发送给被观察者

}

return Observable.fromIterable(list);

}

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

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

} // 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(String value) {

Log.d(TAG, "接收到了事件：" + value);

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

4.Buffrt()

* 作用：定期从 被观察者（Obervable）需要发送的事件中 获取一定数量的事件 & 放到缓存区中，最终发送
* 应用场景：缓存被观察者发送的事件

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

.buffer(3, 1) // 设置缓存区大小 & 步长

// 缓存区大小 = 每次从被观察者中获取的事件数量

// 步长 = 每次获取新事件的数量

.subscribe(new Observer<List<Integer>>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

} // 默认最先调用复写的 onSubscribe（）

@Override

public void onNext(List<Integer> stringList) {

Log.d(TAG, " 缓存区里的事件数量 = " + stringList.size());

for (Integer value : stringList) {

Log.d(TAG, " 事件 = " + value);

}

}

@Override

public void onError(Throwable e) {

Log.d(TAG, "对Error事件作出响应");

}

@Override

public void onComplete() {

Log.d(TAG, "对Complete事件作出响应");

}

});

**过滤操作符**

### 根据指定事件条件过滤事件

### 1. filter()

* 作用：通过一定逻辑来过滤被观察者发送的事件，如果返回true则发送事件，否则不会发送
* 应用场景：筛选符合要求的事件

Observable.just(1,2,3,4,5).filter(new Predicate<Integer>() {

@Override

public boolean test(Integer integer) throws Exception {

return integer % 3 == 1;

}

}).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: ");

}

});

**2.ofType()**

* 作用：ofType是filter操作符的一个特殊形式。它过滤一个Observable只返回指定类型的数据。

Observable.just(1, 2, "hello", "world")

.ofType(Integer.class)

.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: ");

}

});

**3.skip() & skipLast()**

* skip()作用：忽略Observable’发射的前N项数据，只保留之后的数据。
* skipLast()作用:从结尾往前数跳过制定数量的事件

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

.skip(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: ");

}

});

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

.skipLast(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: ");

}

});

**4.distinct（） / distinctUntilChanged（）**

* 作用：过滤事件序列中重复的事件 / 连续重复的事件

// 使用1：过滤事件序列中重复的事件

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

.distinct()

.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: ");

}

});

//使用2：过滤事件序列中 连续重复的事件

// 下面序列中，连续重复的事件 = 3、4

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

.distinctUntilChanged()

.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: ");

}

### });

### 根据指定事件数量过滤事件

需求场景：通过设置指定的事件数量，仅发送特定数量的事件

**1.take（）**

* 作用： 只发射前面的N项数据、

// 使用1：过滤事件序列中重复的事件

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

.take(4)

.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: ");

}

### });

### 2.takeLast（）

* 作用:发射Observable发射的最后N项数据

// 使用1：过滤事件序列中重复的事件

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

.takeLast(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: ");

}

});

### 根据指定时间过滤事件

### 1.throttleFirst（）/ throttleLast（）

* 作用：在某段时间内，只发送该段时间内第1次事件 / 最后1次事件（1段时间内连续点击按钮，但只执行第1次的点击操作）

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

@Override

public void subscribe(ObservableEmitter<Integer> e) throws Exception {

e.onNext(1);

Thread.sleep(500);

e.onNext(2);

Thread.sleep(400);

e.onNext(3);

Thread.sleep(300);

e.onNext(4);

Thread.sleep(300);

e.onNext(5);

Thread.sleep(300);

e.onNext(6);

Thread.sleep(400);

e.onNext(7);

Thread.sleep(300);

e.onNext(8);

Thread.sleep(300);

e.onNext(9);

Thread.sleep(300);

e.onComplete();

}

}).throttleFirst(1, TimeUnit.SECONDS)//每1秒中采用数据

.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: ");

}

});

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

@Override

public void subscribe(ObservableEmitter<Integer> e) throws Exception {

e.onNext(1);

Thread.sleep(500);

e.onNext(2);

Thread.sleep(400);

e.onNext(3);

Thread.sleep(300);

e.onNext(4);

Thread.sleep(300);

e.onNext(5);

Thread.sleep(300);

e.onNext(6);

Thread.sleep(400);

e.onNext(7);

Thread.sleep(300);

e.onNext(8);

Thread.sleep(300);

e.onNext(9);

Thread.sleep(300);

e.onComplete();

}

}).throttleLast(1, TimeUnit.SECONDS)//每1秒中采用数据

.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: ");

}

});

2.Sample（）

* 作用:在某段时间内，只发送该段时间内最新（最后）1次事件
* 和throttleLast（） 操作符类似，仅需要把上文的 throttleLast（） 改成Sample（）操作符即可

3.throttleWithTimeout （） / debounce（）

* 作用：发送数据事件时，若2次发送事件的间隔＜指定时间，就会丢弃前一次的数据，直到指定时间内都没有新数据发射时才会发送后一次的数据

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

@Override

public void subscribe(ObservableEmitter<Integer> e) throws Exception {

e.onNext(1);

Thread.sleep(500);

e.onNext(2);

Thread.sleep(1011);

e.onNext(3);

Thread.sleep(300);

e.onNext(4);

Thread.sleep(600);

e.onNext(5);

Thread.sleep(1200);

e.onNext(6);

Thread.sleep(400);

e.onNext(7);

Thread.sleep(300);

e.onNext(8);

Thread.sleep(300);

e.onNext(9);

Thread.sleep(300);

e.onComplete();

}

}).throttleWithTimeout(1, TimeUnit.SECONDS)//每1秒中采用数据

.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: ");

}

});

### 六. 根据指定事件位置过滤事件

### 1.firstElement（） / lastElement（）

* 作用：仅选取第1个元素 / 最后一个元素

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

.firstElement()

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.d(TAG, "获取到的第一个事件是： " + integer);

}

});

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

.lastElement()

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.d(TAG, "获取到的第一个事件是： " + integer);

}

});

2.elementAt（）

* 作用：指定接收某个元素（通过 索引值 确定）
* 允许越界，即获取的位置索引 ＞ 发送事件序列长度

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

.elementAt(2)

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.d(TAG, "获取到的第一个事件是： " + integer);

}

});

3.elementAtOrError（）

* 作用:在elementAt（）的基础上，当出现越界情况（即获取的位置索引 ＞ 发送事件序列长度）时，即抛出异常

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

.elementAtOrError(7)

.subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.d(TAG, "获取到的第一个事件是： " + integer);

}

});