# RxJava

##### 6.1 Rxjava的定义

Github上的介绍：RxJava：a library for composing asynchronous and event-based programs using observable sequences for the Java VM

翻译：RxJava 是一个在 Java VM 上使用可观测的序列来组成异步的、基于事件的程序的库

总结：Rxjava是一个基于事件流，实现异步操作的库，是一个响应式编程框架RX在java语言中的使用。

##### 6.2 Rxjava的作用

实现异步操作，相当于AsyncTask、Handler

##### 6.3 Rxjava的特点

基于事件流的链式调用，

* 逻辑简洁
* 实现优雅
* 使用简单

随着程序逻辑的复杂性提高，它依然能够保持简洁和优雅

**6.4 响应式编程**

定义：基于异步事件流的编程模式，基于变化的反应

事件：唯一合理的方式将我们的现实世界映射到我们的软件中

流：万物皆流，事件的体现形式，以时间为序，可以被观察、过滤、组合，有源头有目的地

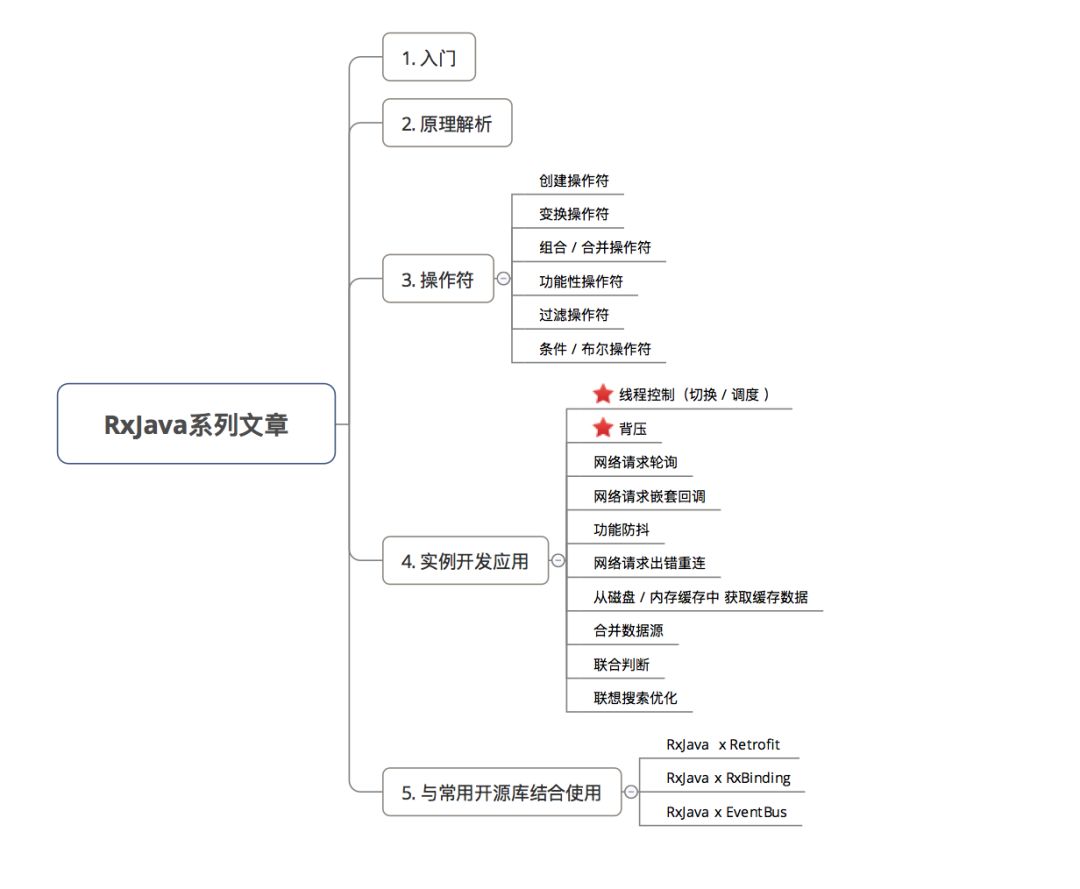
**6.5 Rxjava的要点**

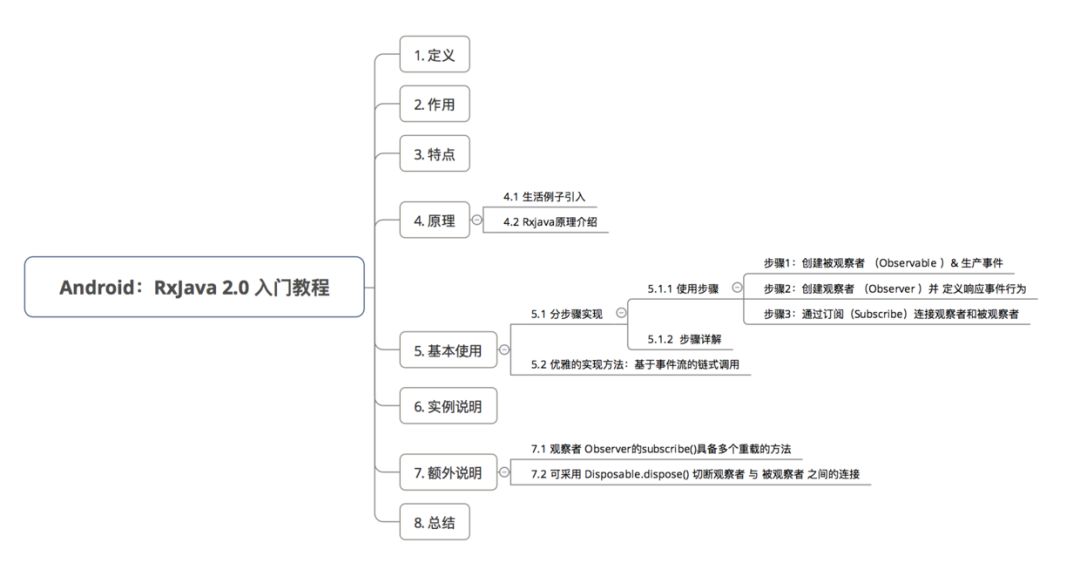
Observable：异步数据流

LINQ：操作符处理数据流(简化、清晰)

Schdulers：线程调度器处理并发

##### 6.6 Rxjava的学习要点

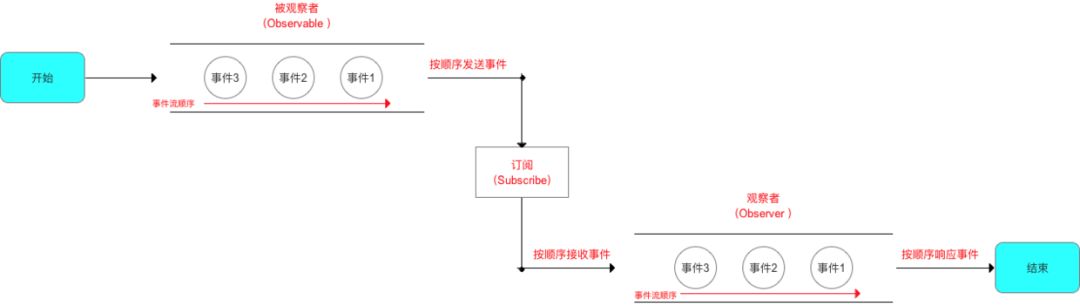




##### 6.5 Rxjava的原理

* Rxjava是一种扩展的观察者模式
* Rxjava的观察者模式中有四个角色
* 被观察者Observable：产生事件
* 观察者Observer：接受事件，并给出响应动作
* 订阅Subscribe：连接观察者&被观察者
* 事件Event：被观察者&观察者 沟通的载体

Rxjava的原理可以总结为：被观察者通过订阅按顺序发送事件给观察者，观察者按顺序收到通知事件并做出相应的响应动作



##### 6.6 Rxjava的使用过程

6.6.1 创建Observable

Observable.create创建

Observable<Integer> observable = Observable

.create(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> emitter)

throws Exception {  
  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onComplete();  
 }  
});

Observable.just创建，循环发送数组里数据

Observable<Integer> justObservable = Observable.just(1, 2, 3, 4);

6.6.2 创建Observer

使用Observer接口创建

Observer<Integer> observer = new Observer<Integer>() {  
 @Override  
 public void onSubscribe(@NonNull Disposable d) {  
 Log.d(TAG, "onSubscribe: 开始使用");  
 }  
  
 @Override  
 public void onNext(@NonNull Integer integer) {  
 Log.d(TAG, "onNext: 对next事件响应");  
 }  
  
 @Override  
 public void onError(@NonNull Throwable e) {  
 Log.d(TAG, "onError: 对error事件响应");  
 }  
  
 @Override  
 public void onComplete() {  
 Log.d(TAG, "onComplete: 对complete事件响应");  
 }  
};

使用Subscriber接口

Subscriber<Integer> subscriber = new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
  
 }  
  
 @Override  
 public void onNext(Integer integer) {  
  
 }  
  
 @Override  
 public void onError(Throwable t) {  
  
 }  
  
 @Override  
 public void onComplete() {  
  
 }  
};

这两个接口方法的区别主要是一个回调的方法不同

* onSubscribe：为观察者提供取消(处理)的方法连接与被观察者的同步和异步方式

public void onSubscribe(@NonNull Disposable d) {  
 Log.d(TAG, "onSubscribe: 开始使用");  
 }

* onSubscribe：在Publisher#subscribe(Subscriber)调用后调用；在执行了Subscription#request(long)之后才会有数据流动；当需要更多数据时，这个订阅者Subscriber实例的职责就是去调用Subscription#request(long)方法；发布者Publisher只会在响应Subscription#request(long)订阅请求时才会发送通知

public void onSubscribe(Subscription s) {  
  
 }

我们先看看Disposable这个类

public interface Disposable {  
 /\*\*  
 \* Dispose the resource, the operation should be idempotent.  
 \*/  
 void dispose();  
  
 /\*\*  
 \* Returns true if this resource has been disposed.  
 \* @return true if this resource has been disposed  
 \*/  
 boolean isDisposed();  
}

两个方法

* dispose()：取消连接
* IsDisposed()：观察者和被观察者连接的标记位

我们在create创建方式下，找到发送onNext方法的源码，会先判断连接是否被取消

@Override  
public void onNext(T t) {  
 if (t == null) {  
 onError(new NullPointerException("onNext called with null. Null values are generally not allowed in 2.x operators and sources."));  
 return;  
 }  
 if (!isDisposed()) {  
 observer.onNext(t);  
 }  
}

我们接下来看看Subscription

public interface Subscription {  
  
 public void request(long n);  
  
 public void cancel();  
}

它里面有两个方法：

* request(long)：

发布者Publisher不会发送任何事件，直到调用这个方法发送信号

无论何时，只要有需求，它都可以被调用，但是未完成的累计需求不能超过Long.MAX\_Value

任何被请求的东西都可以通过Publisher发布者发送，因此只需要对可以安全处理的东西发出信号

参数long是对上游发布者请求的严格数量的元素数量

* cancel()

请求发布者停止发送数据和清理资源

当请求是异步时，在调用了cancel方法后，数据仍然被发送去满足先前的信号要求

6.6.3 通过订阅（Subscribe）连接观察者和被观察者

observable.subscribe(observer);

6.6.4 基于事件流的链式调用以及Disposable的使用

Observable.create(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull ObservableEmitter<Integer> emitter)

throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onComplete();  
 }  
}).subscribe(new Observer<Integer>() {

Disposable d;  
 @Override  
 public void onSubscribe(@NonNull Disposable d) {

this.d = d;

}  
  
 @Override  
 public void onNext(@NonNull Integer integer) {  
 if (integer == 2) {  
 d.dispose();  
 }

}  
  
 @Override  
 public void onError(@NonNull Throwable e) {}  
  
 @Override  
 public void onComplete() {}  
});

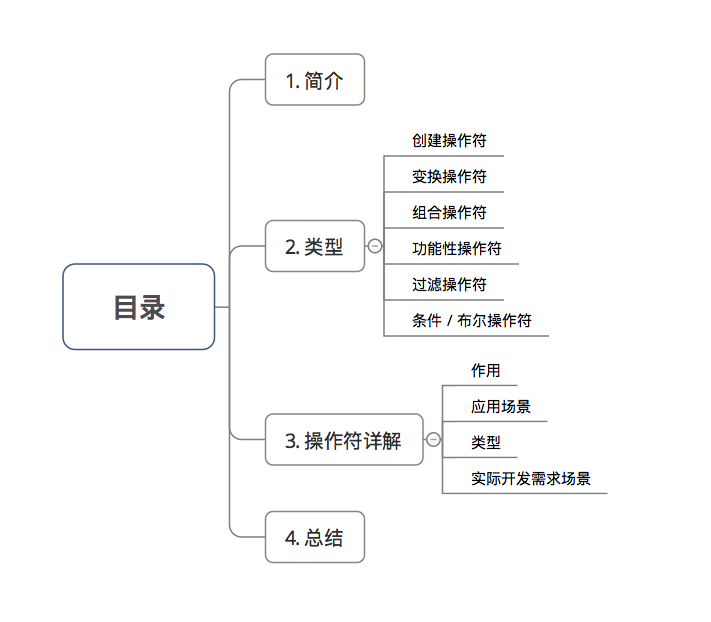
6.6.5 Rxjava提供了许多函数式接口，用于实现简便式的观察者模式

以Consumer为例：

Observable.just("hello").subscribe(new Consumer<String>() {  
 @Override  
 public void accept(@NonNull String s) throws Exception {  
   
 }  
});

##### 6.7 Rxjava的操作符

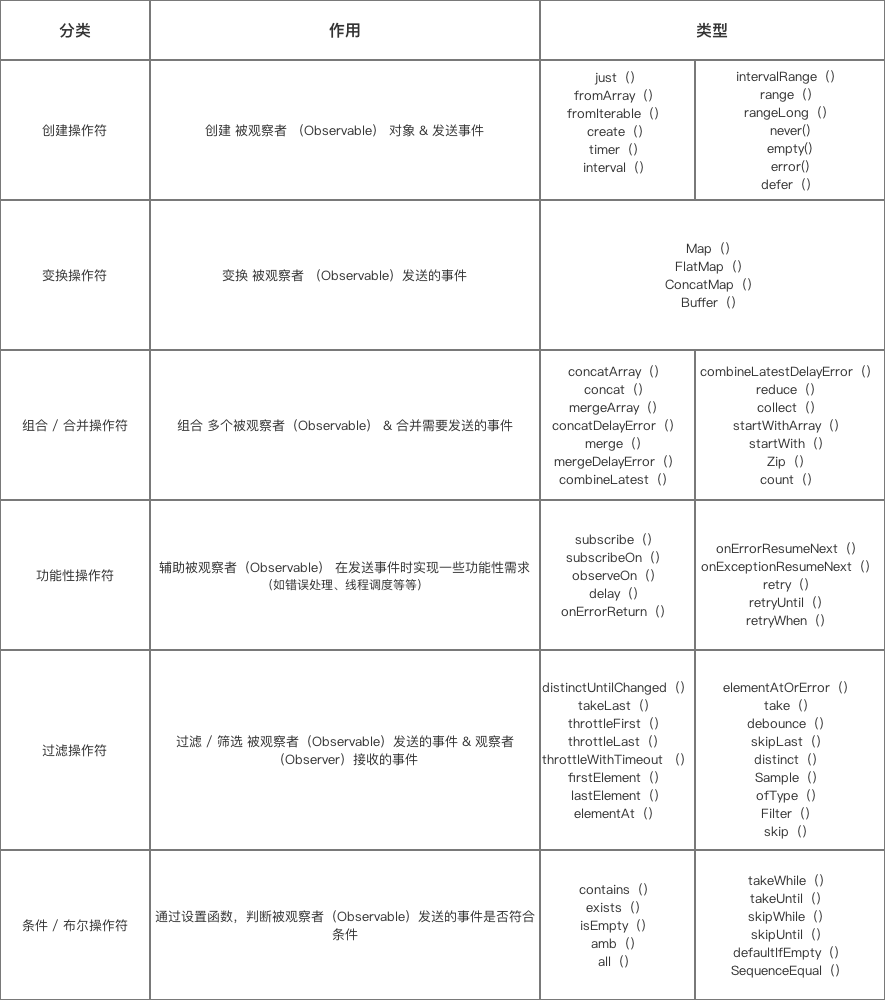
先看看本节的学习目录



6.7.1 操作符的简介

* 定义：Rxjava封装好的使用方法/api
* 作用：实现RXjava的各种功能(创建、变换、线程调度等)
* 特点：丰富、功能强大，通过使用、组合这些操作符，能完成大部分异步场景下的功能需求

6.7.2 操作符的类型

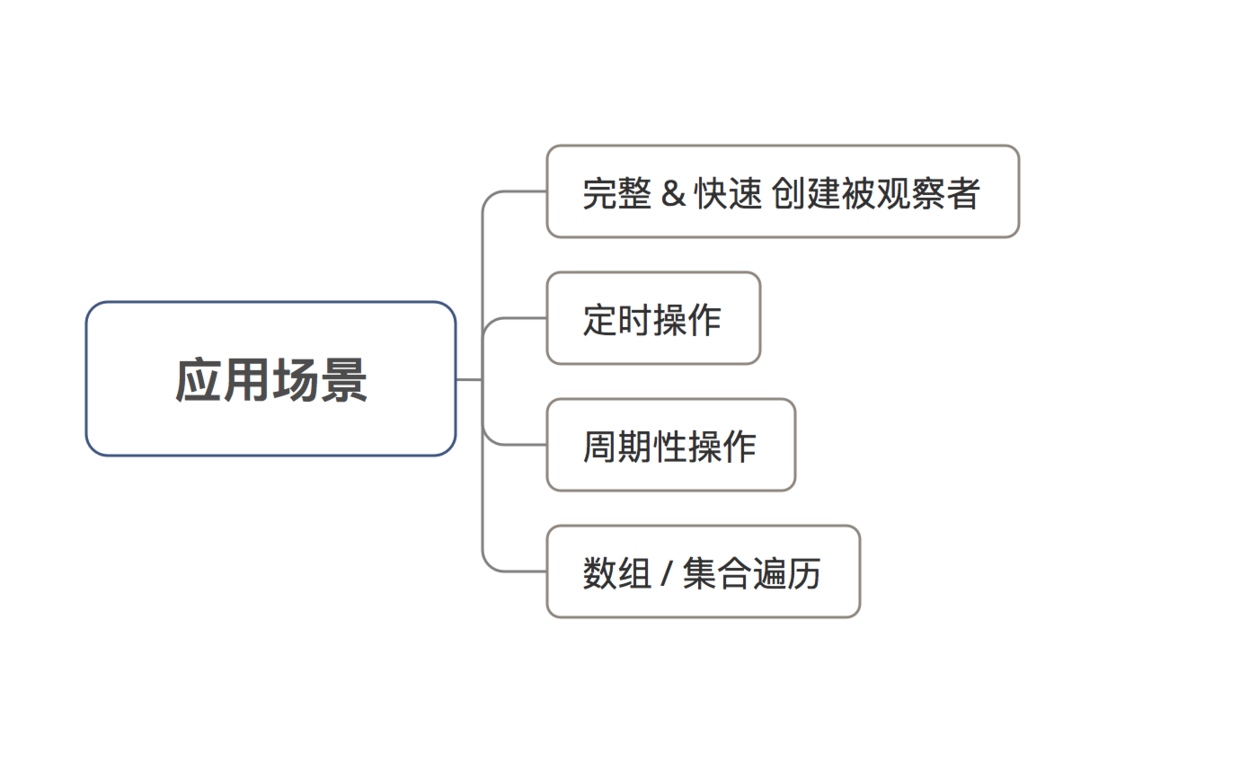


6.7.3 操作符详解

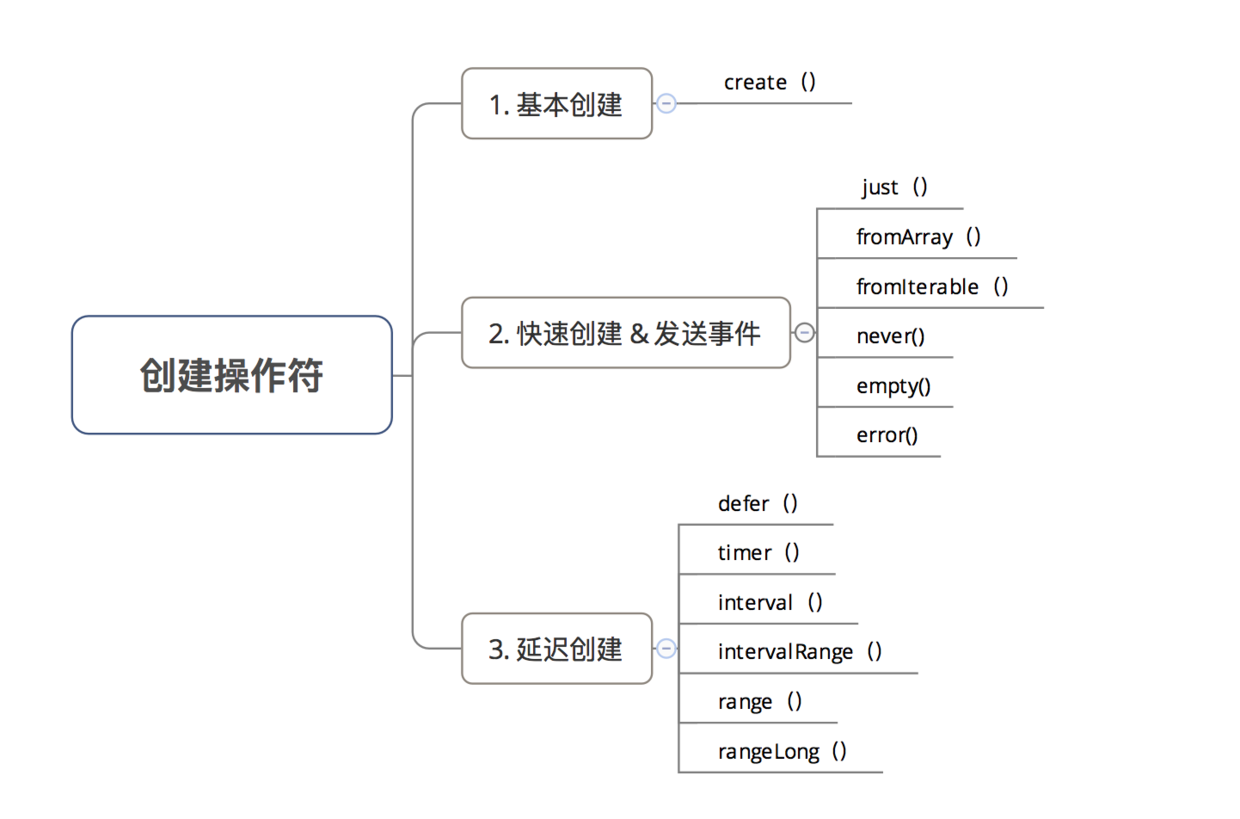
##### 6.7.3.1 创建操作符

创建被观察者Observable对象&发送事件

应用场景



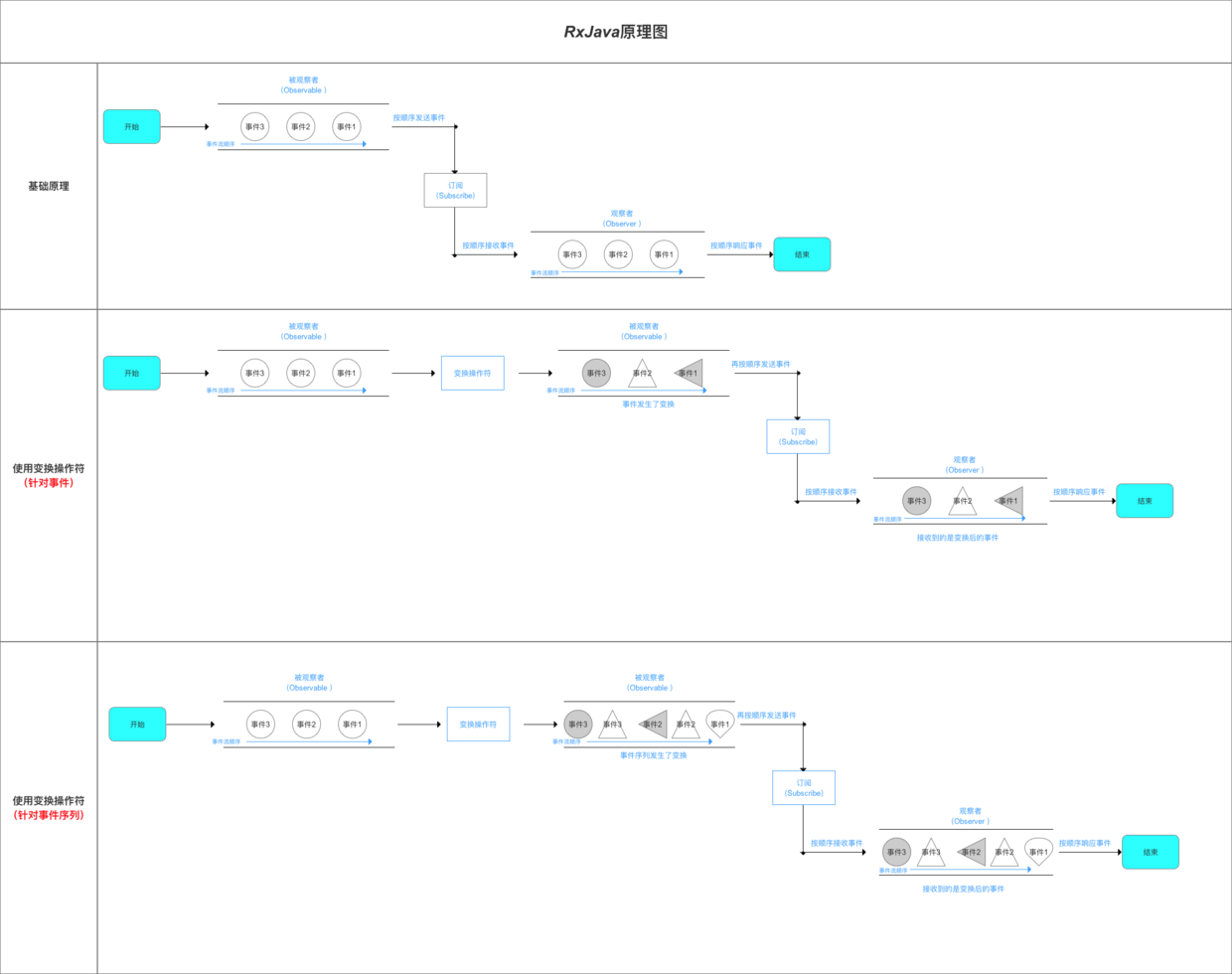
类型



##### 6.7.3.2 变换操作符

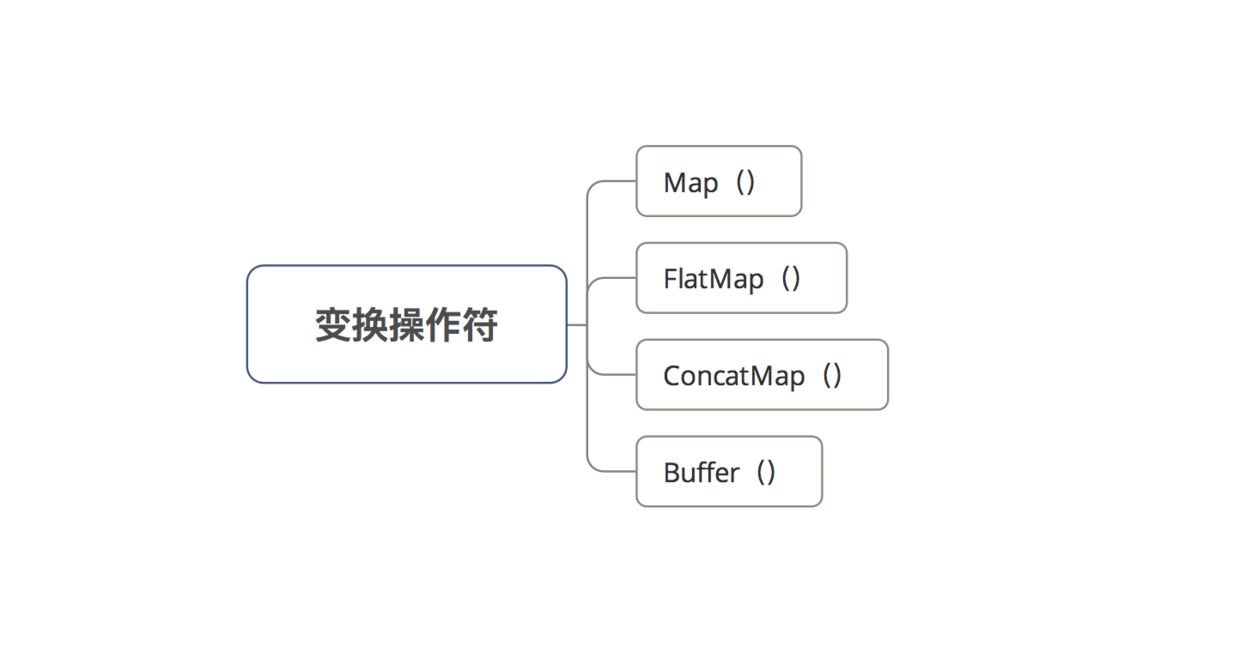
对事件序列中的事件/整个事件序列进行加工(即变换)，使其转变成不用的事件/整个事件序列

原理图如下：



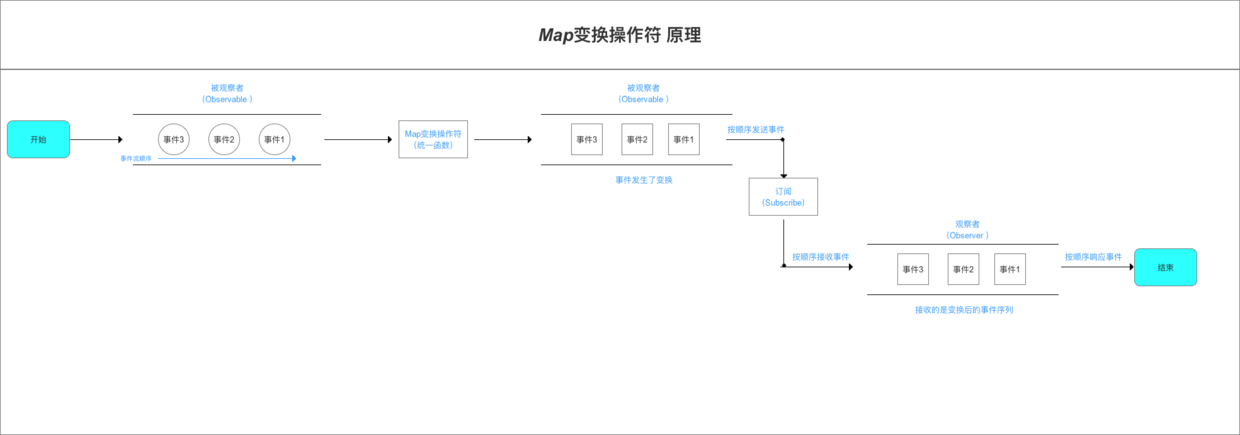
应用场景：嵌套回调（Callback hell）

类型：



**map操作符**：对被观察者发送的每一个事件都通过指定的函数进行处理，从而转变成另一种事件

map操作符原理图：



map操作符的代码实例：

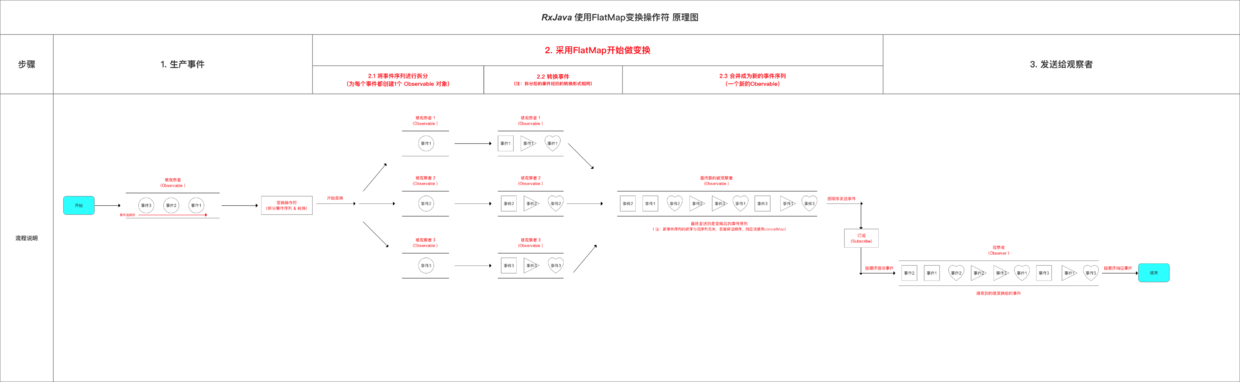
Observable.create(new ObservableOnSubscribe<Integer>() {  
  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter)

throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
  
 }  
}).map(new Function<Integer, String>() {  
 @Override  
 public String apply(Integer integer) throws Exception {  
 return "..." + integer ;  
 }  
}).subscribe(new Consumer<String>() {  
  
 @Override  
 public void accept(String s) throws Exception {  
 Log.d(TAG, s);  
 }  
});

**FlatMap()操作符**：将被观察者发送的事件序列进行拆分&单独转换，再合并成一个新的事件序列，然后再进行发送。

原理：

* 为事件序列中每个事件都创建一个 Observable 对象；
* 将对每个 原始事件 转换后的 新事件 都放入到对应 Observable对象；
* 将新建的每个Observable 都合并到一个 新建的、总的Observable 对象；
* 新建的、总的Observable 对象 将 新合并的事件序列 发送给观察者（Observer）



Flatmap操作符使用实例

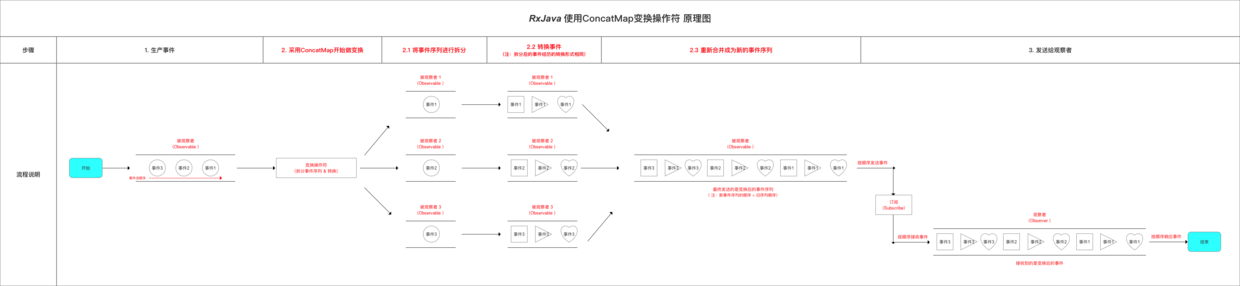
Observable.create(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter)

throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
 }  
 }).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);  
 }  
 return Observable.fromIterable(list);  
 }  
 }).subscribe(new Consumer<String>() {  
 @Override  
 public void accept(String s) throws Exception {  
 Log.d(TAG, s);  
 }  
 });  
}

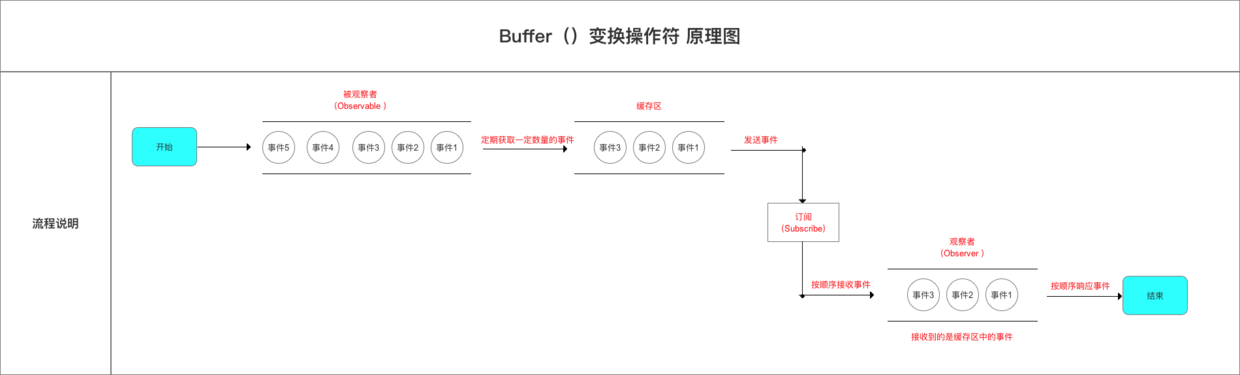
注：新合并生成的事件序列顺序是无序的，即 与旧序列发送事件的顺序无关

**ConcatMap()操作符**：作用和原理跟flatmap类型，但是唯一区别是拆分 & 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



**Buffer操作符**：定期从被观察者Observable需要发送的事件中获取一定数量的事件放到缓冲区中，最后发送

原理图：



应用场景：缓存被观察者发送的事件

Buffer()操作符的使用用例：

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) {  
  
 }  
  
 @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");  
 }  
 });

这里需要看看打印信息：

缓存区里的事件数量 = 3

事件 = 1

事件 = 2

事件 = 3

缓存区里的事件数量 = 3

事件 = 2

事件 = 3

事件 = 4

缓存区里的事件数量 = 3

事件 = 3

事件 = 4

事件 = 5

缓存区里的事件数量 = 2

事件 = 4

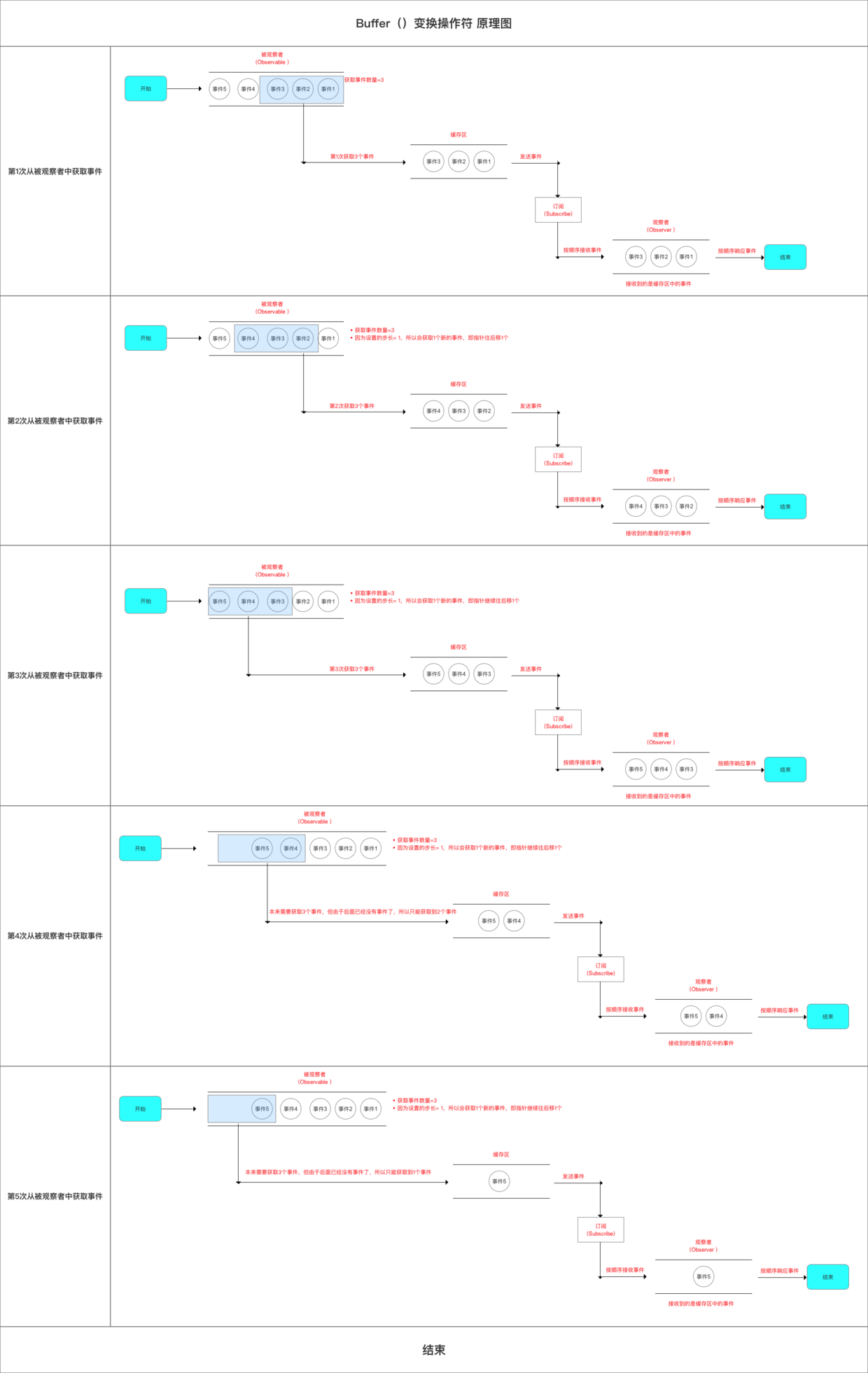
事件 = 5

缓存区里的事件数量 = 1

事件 = 5

对Complete事件作出响应

图解过程解释什么是步长



##### 转换操作符实际运用实例

模仿账号注册成功到登陆的过程

1. 注册和登陆的api

public interface Api {

@GET

Observable<LoginResponse> login(@Body LoginRequest request);

@GET

Observable<RegisterResponse> register(@Body RegisterRequest request);

}

1. 注册登陆过程

api.register(new RegisterRequest())  
 .subscribeOn(Schedulers.io())  
 .observeOn(AndroidSchedulers.mainThread())  
 .doOnNext(new Consumer<RegisterResponse>() {  
 @Override  
 public void accept(RegisterResponse registerResponse){  
 //先根据注册的响应结果去做一些操作  
 }  
 }).observeOn(Schedulers.io())  
 //回到IO线程去发起登录请求  
 .flatMap(new Function<RegisterResponse

, ObservableSource<LoginResponse>>() {  
 @Override  
 public ObservableSource<LoginResponse> apply(

RegisterResponse registerResponse){  
 return api.login(new LoginRequest());  
 }  
 }).observeOn(AndroidSchedulers.mainThread())  
 //回到主线程去处理请求登录的结果  
 .subscribe(new Consumer<LoginResponse>() {  
 @Override  
 public void accept(LoginResponse loginResponse) throws Exception {  
 Log.d(TAG, "accept: 登陆成功");  
 }  
 }, new Consumer<Throwable>() {  
 @Override  
 public void accept(Throwable throwable) throws Exception {  
 Log.e(TAG, "accept: 登陆失败");  
 }  
 });

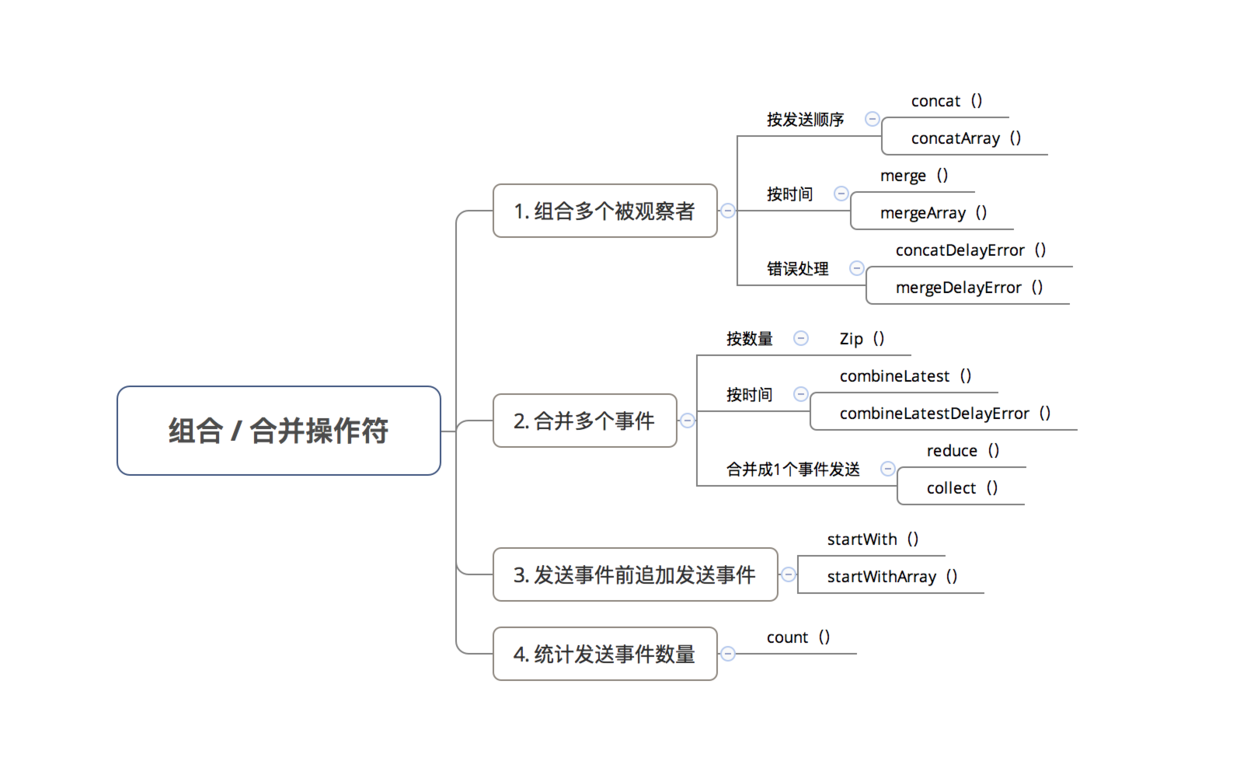
##### 6.7.3.3 组合/合并操作符

作用：组合多个被观察者&合并需要发送的事件

应用场景：

* 组合多个被观察者
* 合并多个事件
* 发送事件前追加事件
* 统计发送事件数量

类型：常用类型如下图



**Concat()/concatArray()操作符：**组合多个被观察者一起发送数据，合并后按发送顺序串行执行

二者区别：组合被观察者的数量，即concat（）组合被观察者数量≤4个，而concatArray（）则可＞4个

Concat()操作符代码示例

// concat（）：组合多个被观察者（≤4个）一起发送数据  
 // 注：串行执行  
Observable.concat(Observable.just(1, 2, 3),  
 Observable.just(4, 5, 6),  
 Observable.just(7, 8, 9),  
 Observable.just(10, 11, 12))  
 .subscribe(new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {}  
  
 @Override  
 public void onNext(Integer value) {}  
  
 @Override  
 public void onError(Throwable e) {}  
  
 @Override  
 public void onComplete() {}  
 });

concatArray()操作符示例：

Observable.concatArray(Observable.just(1, 2, 3),  
 Observable.just(4, 5, 6),  
 Observable.just(7, 8, 9),  
 Observable.just(10, 11, 12),  
 Observable.just(13, 14, 15))  
 .subscribe(new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {}  
  
 @Override  
 public void onNext(Integer value) {}  
  
 @Override  
 public void onError(Throwable e) {}  
  
 @Override  
 public void onComplete() {}  
 });

如果打印结果会是顺序输出1，2，3，4，5，6，7，8，9...

**Merge()/mergeArray()操作符**：组合多个被观察者一起发送数据，合并后按时间线并行执行

* 二者区别：组合被观察者的数量，即merge（）组合被观察者数量≤4个，而mergeArray（）则可＞4个
* 区别上述concat（）操作符：同样是组合多个被观察者一起发送数据，但concat（）操作符合并后是按发送顺序串行执行

直接来看代码实例：

Observable.merge(  
 Observable.intervalRange(0, 3, 1, 1, TimeUnit.SECONDS),

// 从0开始发送、共发送3个数据、第1次事件延迟发送时间 = 1s、间隔时间 = 1s  
 Observable.intervalRange(2, 3, 1, 1, TimeUnit.SECONDS))

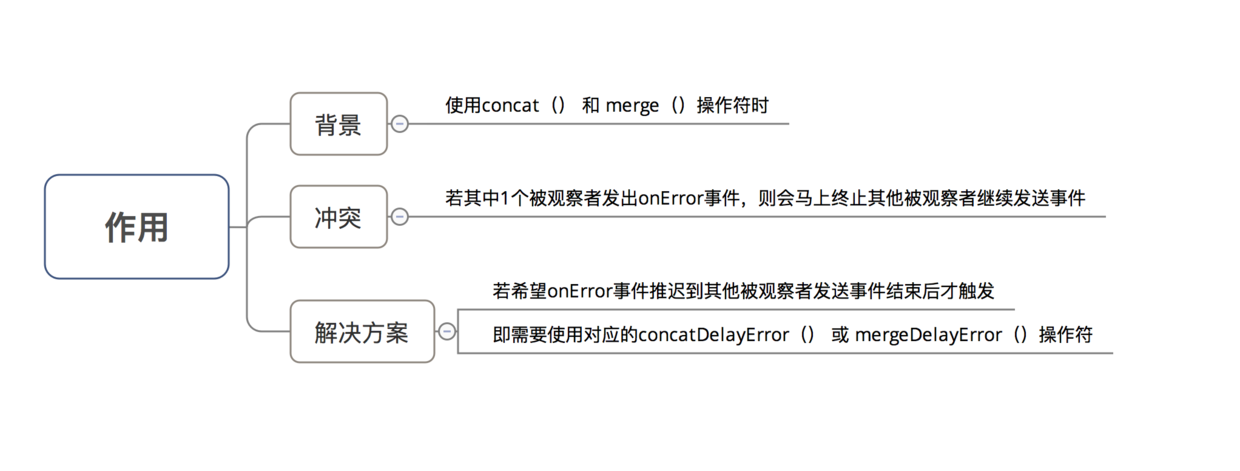
// 从2开始发送、共发送3个数据、第1次事件延迟发送时间 = 1s、间隔时间 = 1s  
 .subscribe(new Observer<Long>() {  
 @Override  
 public void onSubscribe(Disposable d) {}  
  
 @Override  
 public void onNext(Long value) {  
 Log.d(TAG, "接收到了事件" + value);  
 }  
  
 @Override  
 public void onError(Throwable e) {}  
  
 @Override  
 public void onComplete() {}  
 });

测试结构：

两个被观察者发送事件并行执行，输出结果 = 0,2 -> 1,3 -> 2,4

**ConcatDelayError()/mergeDelayError()操作符**

作用如图：



代码实例：

Observable.concatArrayDelayError(  
 Observable.create(new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter)

throws Exception {  
 emitter.onNext(1);  
 emitter.onNext(2);  
 emitter.onNext(3);  
 emitter.onError(new NullPointerException());

// 发送Error事件，因为使用了concatDelayError，

所以第2个Observable将会发送事件，等发送完毕后，再发送错误事件  
 emitter.onComplete();  
 }  
 }),  
 Observable.just(4, 5, 6))  
 .subscribe(new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {}  
  
 @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");  
 }  
 });

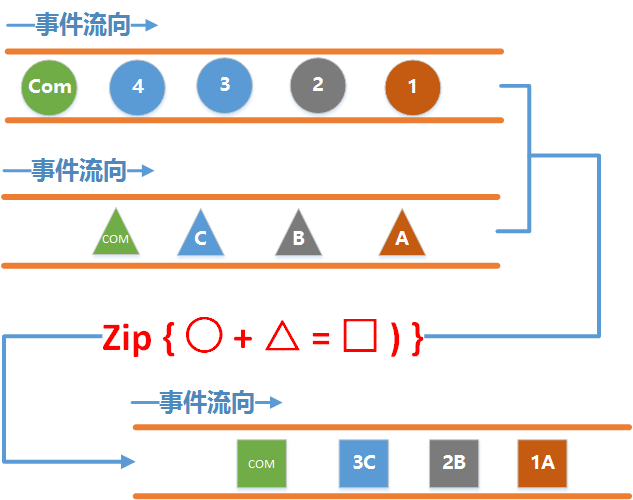
测试结果：1，2，3，4，5，6，error

**ZIP()操作符**：合并多个被观察者发送的事件，生成一个新的事件序列（即组合过后的事件序列），并最终发送

介绍：



原理图如下：



* 特别注意：
* 事件组合方式 = 严格按照原先事件序列 进行对位合并
* 最终合并的事件数量 = 多个被观察者（Observable）中数量最少的数量

代码实例：

* 创建第一个被观察者

Observable<Integer> observable1 = Observable.create(

new ObservableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(ObservableEmitter<Integer> emitter)

throws Exception {  
 emitter.onNext(1);  
 // 为了方便展示效果，所以在发送事件后加入2s的延迟  
 Thread.sleep(1000);  
  
 emitter.onNext(2);  
 Thread.sleep(1000);  
  
 emitter.onNext(3);  
 Thread.sleep(1000);  
  
 emitter.onComplete();  
 }  
}).subscribeOn(Schedulers.io());

* 创建第二个被观察者

Observable<String> observable2 = Observable.create(

new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter)

throws Exception {  
 emitter.onNext("A");  
 Thread.sleep(1000);  
  
 emitter.onNext("B");  
 Thread.sleep(1000);  
  
 emitter.onNext("C");  
 Thread.sleep(1000);  
  
 emitter.onNext("D");  
 Thread.sleep(1000);  
  
 emitter.onComplete();  
 }  
}).subscribeOn(Schedulers.newThread());

* 合并被观察者，创建BiFunction对象传入的第3个参数 = 合并后数据的数据类型

Observable.zip(observable1, observable2

, new BiFunction<Integer, String, String>() {  
 @Override  
 public String apply(Integer integer, String string) throws Exception {  
 return integer + string;  
 }})

.subscribe(new Observer<String>() {  
 @Override  
 public void onSubscribe(Disposable d) {}  
  
 @Override  
 public void onNext(String value) {  
 Log.d(TAG, "最终接收到的事件 = " + value);  
 }  
  
 @Override  
 public void onError(Throwable e) {}  
  
 @Override  
 public void onComplete() {}  
});

测试结果：

最终接收到的事件 = 1A

最终接收到的事件 = 2B

最终接收到的事件 = 3C

**combineLatest()操作符**：当两个Observables中的任何一个发送了数据后，将先发送了数据的Observables 的最新（最后）一个数据 与 另外一个Observable发送的每个数据结合，最终基于该函数的结果发送数据

与Zip（）的区别：Zip（） = 按个数合并，即1对1合并；CombineLatest（） = 按时间合并，即在同一个时间点上合并

代码实例：

Observable.combineLatest(  
 Observable.just(1L, 2L, 3L), // 第1个发送数据事件的Observable  
 Observable.intervalRange(0, 3, 1, 1, TimeUnit.SECONDS),

// 第2个发送数据事件的Observable：

从0开始发送、共发送3个数据、第1次事件延迟发送时间 = 1s、间隔时间 = 1s  
 new BiFunction<Long, Long, Long>() {  
 @Override  
 public Long apply(Long o1, Long o2) throws Exception {  
 // o1 = 第1个Observable发送的最新（最后）1个数据  
 // o2 = 第2个Observable发送的每1个数据  
 Log.e(TAG, "合并的数据是： "+ o1 + " "+ o2);  
 return o1 + o2;  
 // 合并的逻辑 = 相加  
 // 即第1个Observable发送的最后1个数据 与

第2个Observable发送的每1个数据进行相加  
 }  
 }).subscribe(new Consumer<Long>() {  
 @Override  
 public void accept(Long s) throws Exception {  
 Log.e(TAG, "合并的结果是： "+s);  
 }  
});

打印结果：

合并的数据是： 3 0

合并的结果是： 3

合并的数据是： 3 1

合并的结果是： 4

合并的数据是： 3 2

合并的结果是： 5

**combineLatestDelayError（）操作符：**作用类似于concatDelayError（） / mergeDelayError（） ，即错误处理，此处不作过多描述

**reduce()操作符**：把被观察者要发送的事件聚合成一个事件并发送

聚合的逻辑根据需求撰写，但本质都是前2个数据聚合，然后与后1个数据继续进行聚合，依次类推

代码实例：

Observable.just(1,2,3,4)  
 .reduce(new BiFunction<Integer, Integer, Integer>() {  
 // 在该复写方法中复写聚合的逻辑  
 @Override  
 public Integer apply(@NonNull Integer s1, @NonNull Integer s2)

throws Exception {  
 Log.e(TAG, "本次计算的数据是： "+s1 +" 乘 "+ s2);  
 return s1 \* s2;  
 // 本次聚合的逻辑是：全部数据相乘起来  
 // 原理：第1次取前2个数据相乘，

之后每次获取到的数据 = 返回的数据x原始下1个数据每  
 }  
 }).subscribe(new Consumer<Integer>() {  
 @Override  
 public void accept(@NonNull Integer s) throws Exception {  
 Log.e(TAG, "最终计算的结果是： "+s);  
 }  
});

打印测试结果：

本次计算的数据是：1 乘 2

本次计算的数据是：2 乘 3

本次计算的数据是：6 乘 4

最终计算的结果是：24

**Collect()操作符**：将被观察者发送的数据事件收集到一个数据结构里

代码实例：

Observable.just(1, 2, 3, 4, 5, 6)  
 .collect(  
 // 1. 创建数据结构（容器），用于收集被观察者发送的数据  
 new Callable<ArrayList<Integer>>() {  
 @Override  
 public ArrayList<Integer> call() throws Exception {  
 return new ArrayList<>();  
 }  
 // 2. 对发送的数据进行收集  
 }, new BiConsumer<ArrayList<Integer>, Integer>() {  
 @Override  
 public void accept(ArrayList<Integer> list, Integer integer)  
 throws Exception {  
 // 参数说明：list = 容器，integer = 后者数据  
 list.add(integer);  
 // 对发送的数据进行收集  
 }  
 }).subscribe(new Consumer<ArrayList<Integer>>() {  
 @Override  
 public void accept(@NonNull ArrayList<Integer> s) throws Exception {  
 Log.e(TAG, "本次发送的数据是： " + s);  
 }  
});

打印结果：[1,2,3,4,5,6]

**startWith() / startWithArray()操作符**：在一个被观察者发送事件前，追加发送一些数据 / 一个新的被观察者

代码示例：

Observable.just(4, 5, 6)  
 .startWith(0) // 追加单个数据 = startWith()  
 .startWithArray(1, 2, 3) // 追加多个数据 = startWithArray()  
 .subscribe(new Observer<Integer>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 }  
  
 @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");  
 }  
 });

打印结果：1，2，3，0，4，5，6

也可以添加被观察者对象：

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

**Count()操作符**：统计被观察者发送事件的数量

##### 组合/合并操作符的使用情景

* 获取缓存(磁盘/内存)数据

//设置第1个Observable：检查内存缓存是否有该数据的缓存  
Observable<String> memory = Observable.create(

new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter)

throws Exception {  
  
 // 先判断内存缓存有无数据  
 if (memoryCache != null) {  
 // 若有该数据，则发送  
 emitter.onNext(memoryCache);  
 } else {  
 // 若无该数据，则直接发送结束事件  
 emitter.onComplete();  
 }  
 }  
});  
  
//设置第2个Observable：检查磁盘缓存是否有该数据的缓存  
Observable<String> disk = Observable.create(

new ObservableOnSubscribe<String>() {  
 @Override  
 public void subscribe(ObservableEmitter<String> emitter)

throws Exception {  
 // 先判断磁盘缓存有无数据  
 if (diskCache != null) {  
 // 若有该数据，则发送  
 emitter.onNext(diskCache);  
 } else {  
 // 若无该数据，则直接发送结束事件  
 emitter.onComplete();  
 }  
 }  
});  
  
//设置第3个Observable：通过网络获取数据  
Observable<String> network = Observable.just("从网络中获取数据");  
  
  
// 1. 通过concat（）合并memory、disk、network 3个被观察者的事件（即检查内存缓存、磁盘缓存 & 发送网络请求）  
Observable.concat(memory, disk, network)  
 // 2. 通过firstElement()，从串联队列中取出并发送第1个有效事件（Next事件）

，即依次判断检查memory、disk、network  
 .firstElement()  
 // 即本例的逻辑为：  
 // a. firstElement()取出第1个事件 = memory，即先判断内存缓存中有无数据缓存；由于memoryCache = null，即内存缓存中无数据，所以发送结束事件（视为无效事件）  
 // b. firstElement()继续取出第2个事件 = disk，即判断磁盘缓存中有无数据缓存：由于diskCache ≠ null，即磁盘缓存中有数据，所以发送Next事件（有效事件）  
 // c. 即firstElement()已发出第1个有效事件（disk事件），所以停止判断。  
  
 // 3. 观察者订阅  
 .subscribe(new Consumer<String>() {  
 @Override  
 public void accept( String s) throws Exception {  
 Log.d(TAG,"最终获取的数据来源 = "+ s);  
 }  
 });

* 合并数据源(网络/本地)，需要从不同的地方获取数据并且同时展示

使用merge操作符：

String result = "数据源来自 = ";  
  
/\*  
 \* 设置第1个Observable：通过网络获取数据  
 \* 此处仅作网络请求的模拟  
 \*\*/  
Observable<String> network = Observable.just("网络");  
  
/\*  
 \* 设置第2个Observable：通过本地文件获取数据  
 \* 此处仅作本地文件请求的模拟  
 \*\*/  
Observable<String> file = Observable.just("本地文件");  
  
  
/\*  
 \* 通过merge（）合并事件 & 同时发送事件  
 \*\*/  
Observable.merge(network, file)  
 .subscribe(new Observer<String>() {  
 @Override  
 public void onSubscribe(Disposable d) {}  
  
 @Override  
 public void onNext(String value) {  
 Log.d(TAG, "数据源有： " + value);  
 result += value + "+";  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 Log.d(TAG, "对Error事件作出响应");  
 }  
  
 // 接收合并事件后，统一展示  
 @Override  
 public void onComplete() {  
 Log.d(TAG, "获取数据完成");  
 Log.d(TAG, result);  
 }  
 });

使用zip()操作符：从两个不同的服务器获取数据，然后合并发送

Observable<UserBaseInfoResponse> observable1 = api.getUserBaseInfo(

new UserBaseInfoRequest())

.subscribeOn(Schedulers.io());  
Observable<UserExtraInfoResponse> observable2 = api.getUserExtraInfo(

new UserExtraInfoRequest())

.subscribeOn(Schedulers.io());

Observable.zip(observable1, observable2

, new BiFunction<UserBaseInfoResponse, UserExtraInfoResponse, UserInfo>() {  
 @Override  
 public UserInfo apply(UserBaseInfoResponse baseInfo

, UserExtraInfoResponse extraInfo){  
 return new UserInfo(baseInfo, extraInfo);  
 }  
}).observeOn(AndroidSchedulers.mainThread()).subscribe(new Consumer<UserInfo>() {  
 @Override  
 public void accept(UserInfo userInfo) throws Exception { //do something; } });  
  
 }

* 联合判断：同时对多个事件进行联合判断

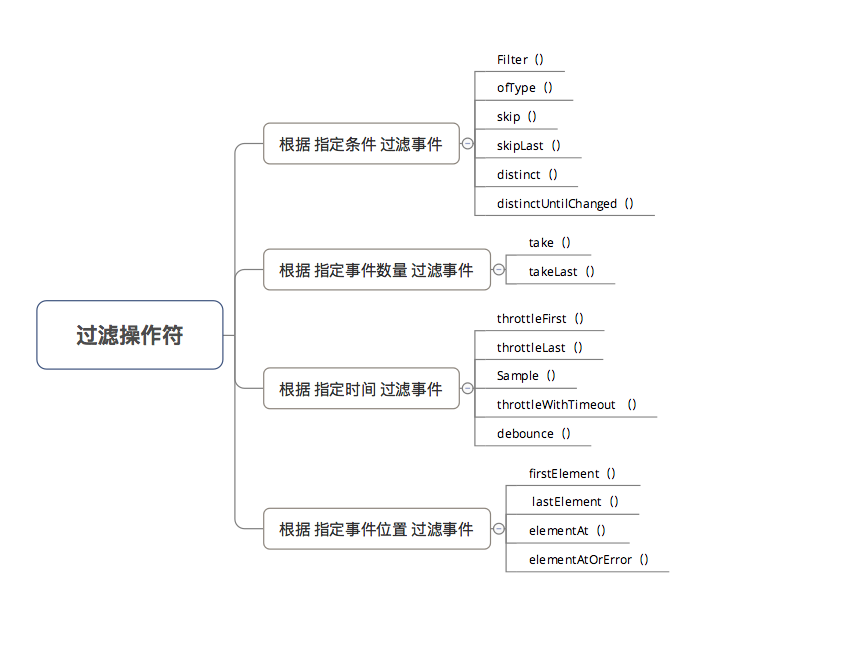
name = (EditText) findViewById(R.id.name);  
age = (EditText) findViewById(R.id.age);  
job = (EditText) findViewById(R.id.job);  
list = (Button) findViewById(R.id.list);  
  
/\*  
 \* 步骤2：为每个EditText设置被观察者，用于发送监听事件  
 \* 说明：  
 \* 1. 此处采用了RxBinding：RxTextView.textChanges(name) = 对对控件数据变更进行监听（功能类似TextWatcher），需要引入依赖：compile 'com.jakewharton.rxbinding2:rxbinding:2.0.0'  
 \* 2. 传入EditText控件，点击任1个EditText撰写时，都会发送数据事件 = Function3（）的返回值（下面会详细说明）  
 \* 3. 采用skip(1)原因：跳过 一开始EditText无任何输入时的空值  
 \*\*/  
Observable<CharSequence> nameObservable = RxTextView.textChanges(name).skip(1);  
Observable<CharSequence> ageObservable = RxTextView.textChanges(age).skip(1);  
Observable<CharSequence> jobObservable = RxTextView.textChanges(job).skip(1);  
  
/\*  
 \* 步骤3：通过combineLatest（）合并事件 & 联合判断  
 \*\*/  
Observable.combineLatest(nameObservable,ageObservable,jobObservable,new Function3<CharSequence, CharSequence, CharSequence,Boolean>() {  
 @Override  
 public Boolean apply(@NonNull CharSequence charSequence, @NonNull CharSequence charSequence2, @NonNull CharSequence charSequence3) throws Exception {  
  
 /\*  
 \* 步骤4：规定表单信息输入不能为空  
 \*\*/  
 // 1. 姓名信息  
 boolean isUserNameValid = !TextUtils.isEmpty(name.getText()) ;  
 // 除了设置为空，也可设置长度限制  
 // boolean isUserNameValid = !TextUtils.isEmpty(name.getText()) && (name.getText().toString().length() > 2 && name.getText().toString().length() < 9);  
  
 // 2. 年龄信息  
 boolean isUserAgeValid = !TextUtils.isEmpty(age.getText());  
 // 3. 职业信息  
 boolean isUserJobValid = !TextUtils.isEmpty(job.getText()) ;  
  
 /\*  
 \* 步骤5：返回信息 = 联合判断，即3个信息同时已填写，"提交按钮"才可点击  
 \*\*/  
 return isUserNameValid && isUserAgeValid && isUserJobValid;  
 }  
  
}).subscribe(new Consumer<Boolean>() {  
 @Override  
 public void accept(Boolean s) throws Exception {  
 /\*  
 \* 步骤6：返回结果 & 设置按钮可点击样式  
 \*\*/  
 Log.e(TAG, "提交按钮是否可点击： "+s);  
 list.setEnabled(s);  
 }  
});

最后，总结一张组合/操作符



##### 6.7.3.4 过滤操作符

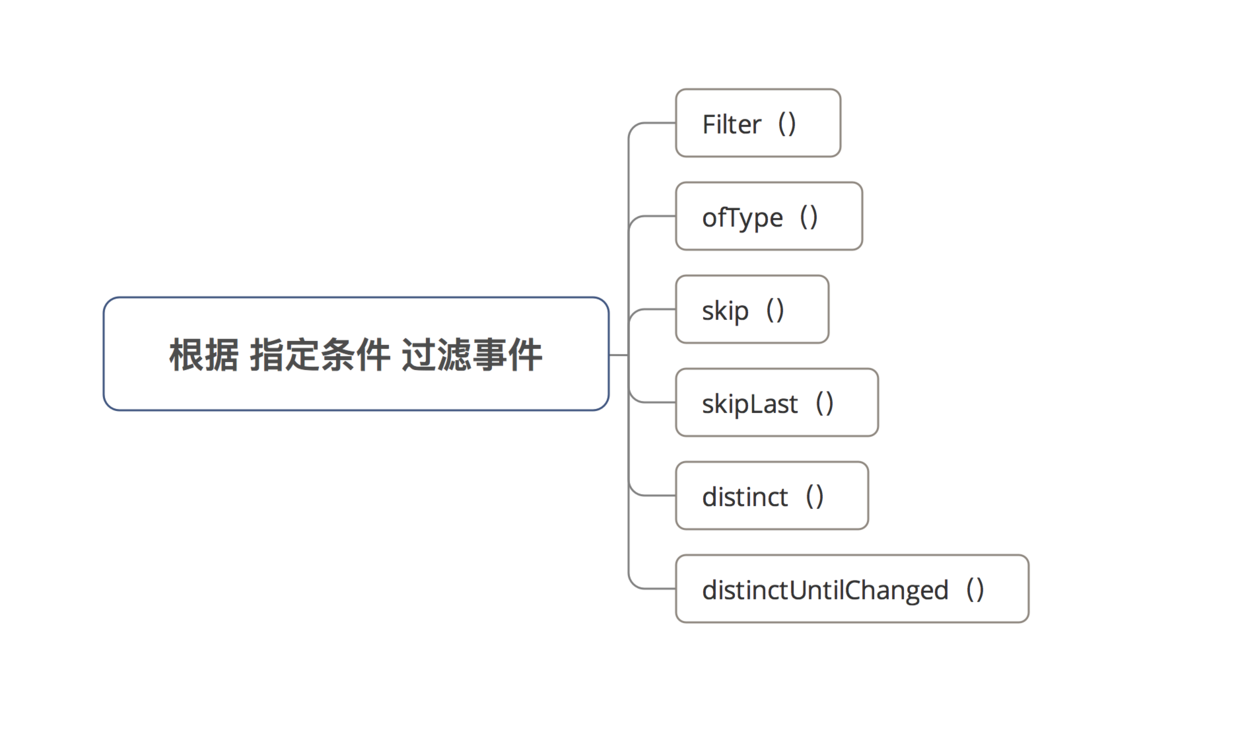
* 作用：过滤/筛选被观察者发送的事件&观察者接收的事件
* 应用场景：
* 根据 指定条件 过滤事件
* 根据 指定事件数量 过滤事件
* 根据 指定时间 过滤事件
* 根据 指定事件位置 过滤事件
* 类型：常用的过滤操作符如下图



**4.1 根据指定条件过滤事件**

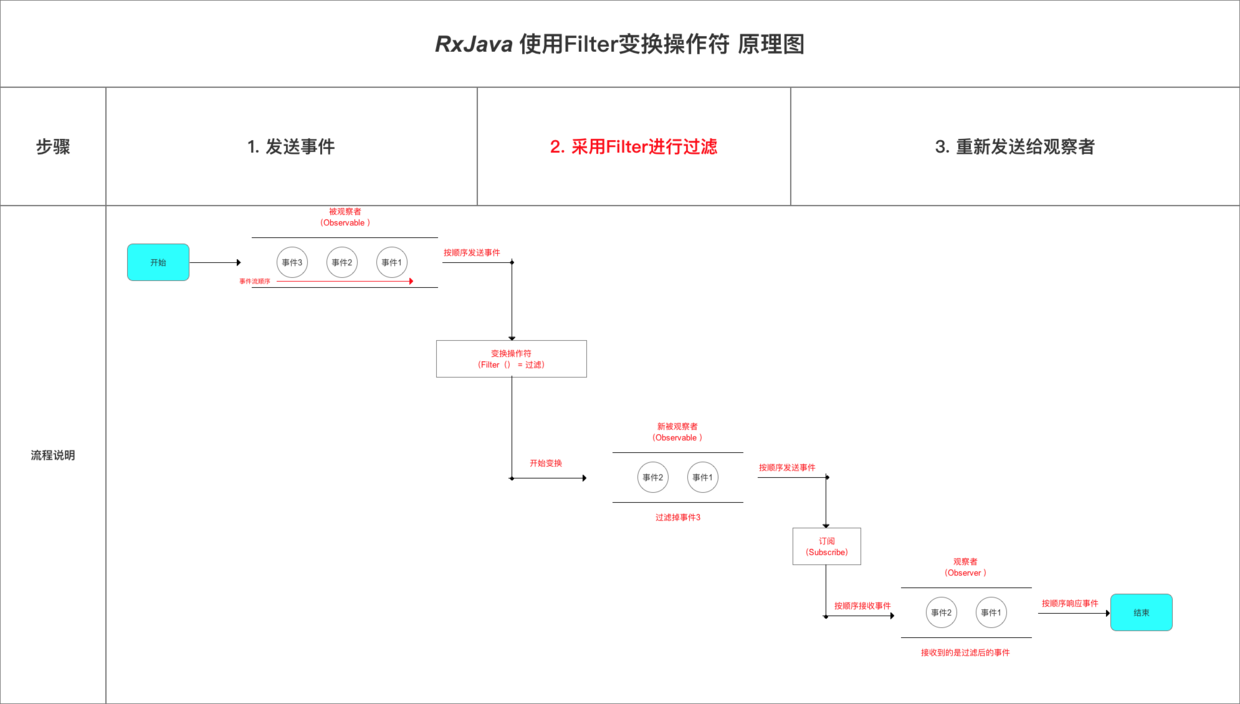
* 需求场景：通过设置指定的过滤条件，当且仅当该事件满足条件，就将该事件过滤（不发送）

对应操作符如下：



**Filter()操作符**：过滤特定条件的事件

原理如图：



代码示例：

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

@Override

public void subscribe(ObservableEmitter<Integer> emitter)

throws Exception {

// 1. 发送5个事件

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

emitter.onNext(4);

emitter.onNext(5);

}

}).filter(new Predicate<Integer>() {

// 根据test()的返回值 对被观察者发送的事件进行过滤 & 筛选

// a. 返回true，则继续发送

// b. 返回false，则不发送（即过滤）

@Override

public boolean test(Integer integer) throws Exception {

return integer > 3;

}

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

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Integer value) {}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

**ofType()操作符**：过滤特定数据类型的数据

具体使用：

Observable.just(1, "Carson", 3, "Ho", 5)

.ofType(Integer.class) // 筛选出 整型数据

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"获取到的整型事件元素是： "+ integer);

}

});

**Skip()/skipLast()操作符**：跳过某个事件

具体使用：

// 使用1：根据顺序跳过数据项

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

.skip(1) // 跳过正序的前1项

.skipLast(2) // 跳过正序的后2项

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"获取到的整型事件元素是： "+ integer);

}

});

// 使用2：根据时间跳过数据项

// 发送事件特点：发送数据0-5，每隔1s发送一次，每次递增1；第1次发送延迟0s

Observable.intervalRange(0, 5, 0, 1, TimeUnit.SECONDS)

.skip(1, TimeUnit.SECONDS) // 跳过第1s发送的数据

.skipLast(1, TimeUnit.SECONDS) // 跳过最后1s发送的数据

.subscribe(new Consumer<Long>() {

@Override

public void accept( Long along ) throws Exception {

Log.d(TAG,"获取到的整型事件元素是： "+ along);

}

});

打印结果分别是2，3和1，2，3

**distinct（） / distinctUntilChanged（）操作符**：过滤事件序列中重复的事件/连续重复的事件

具体使用：

// 使用1：过滤事件序列中重复的事件

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

.distinct()

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"不重复的整型事件元素是： "+ integer);

}

});

// 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 Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"不连续重复的整型事件元素是： "+ integer);

}

});

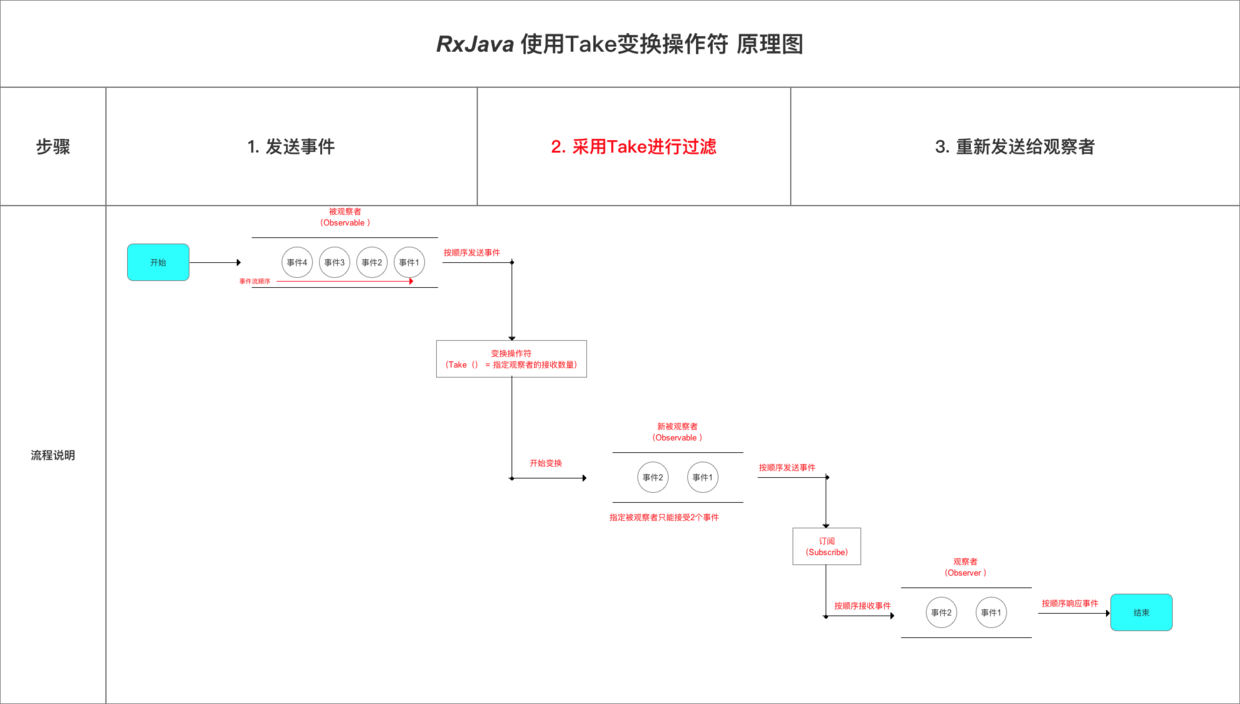
打印结果分别是1，2，3和1，2，3，1，2，3，4

**4.2 根据 指定事件数量 过滤条件**

* 应用场景：通过设置指定的事件数量，仅发送特定数量的事件
* 对应操作符类型：take() & takeLast()

**Take()操作符**：指定观察者最多能接收到的事件数量

原理：



具体使用：

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

@Override

public void subscribe(ObservableEmitter<Integer> emitter)

throws Exception {

// 1. 发送5个事件

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

emitter.onNext(4);

emitter.onNext(5);

}

// 采用take（）变换操作符

// 指定了观察者只能接收2个事件

}).take(2)

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Integer value) {

Log.d(TAG, "过滤后得到的事件是："+ value );

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

打印结果为：1，2

**takeLast操作符**：指定观察者只能接收到被观察者发送的最后几个事件

具体使用：

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

.takeLast(3) //指定观察者只能接受被观察者发送的3个事件

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Integer value) {

Log.d(TAG, "过滤后得到的事件是："+ value );

}

@Override

public void onError(Throwable e) {}

@Override

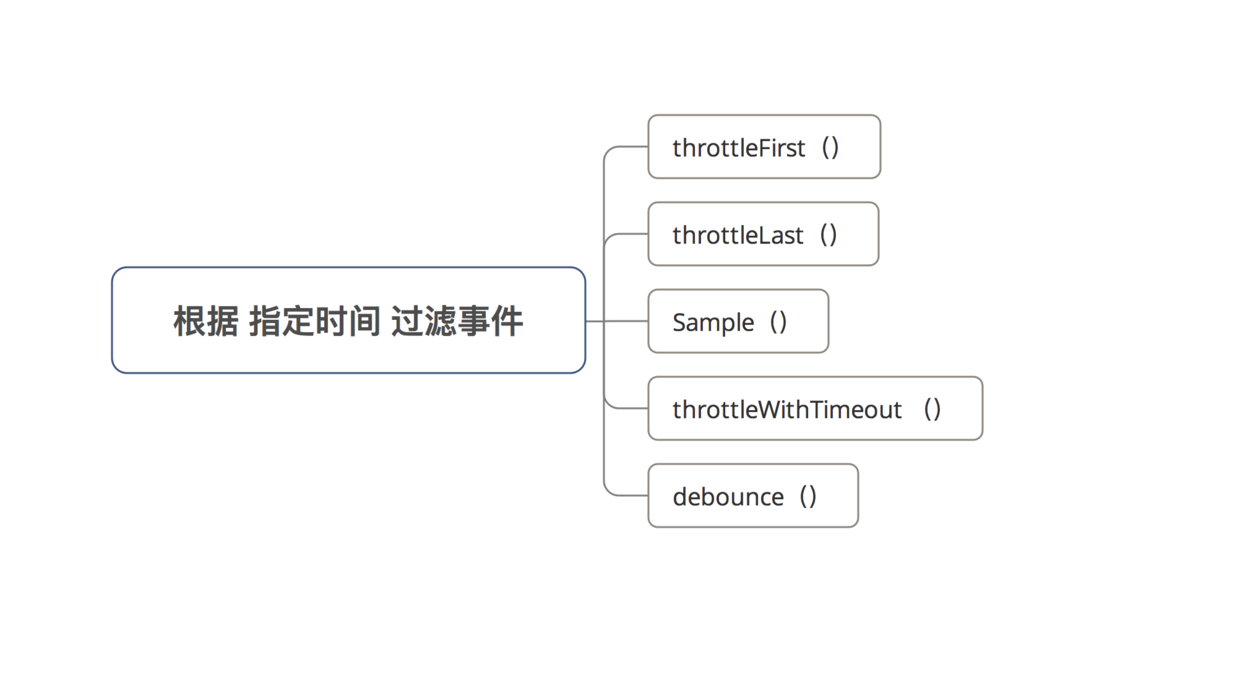
public void onComplete() {}

});

打印结果：3，4，5

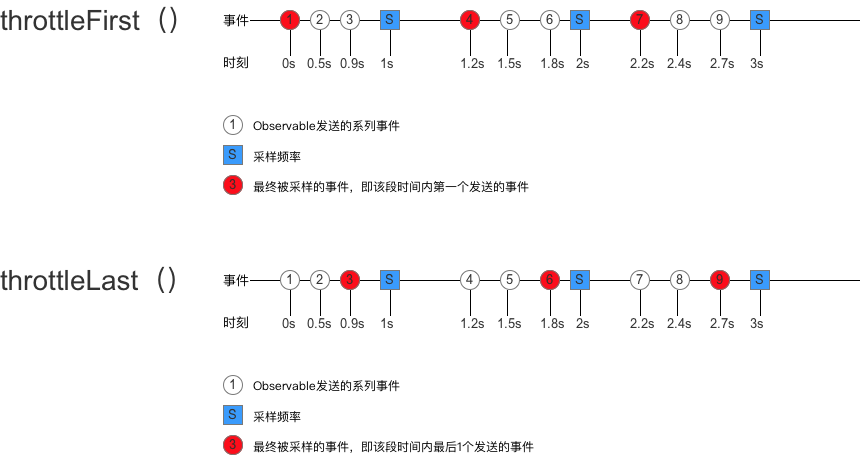
**4.3 根据 指定时间 过滤事件**

* 需求场景：通过设置指定的时间，仅发送在该时间内的事件
* 对应操作符类型：



**throttleFirst/ throttleLast操作符**：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

原理图：



具体使用：

<<- 在某段时间内，只发送该段时间内第1次事件 ->>

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

// 隔段事件发送时间

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) {}

@Override

public void onNext(Integer value) {

Log.d(TAG, "接收到了事件"+ value );

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

<<- 在某段时间内，只发送该段时间内最后1次事件 ->>

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

// 隔段事件发送时间

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) {}

@Override

public void onNext(Integer value) {

Log.d(TAG, "接收到了事件"+ value );

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

打印结果为1，4，7和3，6，9

**Sample()操作符**：某段时间内，只发送该时间段内最新(最后)1次事件，与throttleLast操作符使用一样。

**throttleWithTimeout / debounce操作符**：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

具体使用：

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

// 隔段事件发送时间

e.onNext(1);

Thread.sleep(500);

e.onNext(2);

// 1和2之间的间隔小于指定时间1s，所以前1次数据（1）会被抛弃，2会被保留

Thread.sleep(1500);

// 因为2和3之间的间隔大于指定时间1s，所以之前被保留的2事件将发出

e.onNext(3);

Thread.sleep(1500);

// 因为3和4之间的间隔大于指定时间1s，所以3事件将发出

e.onNext(4);

Thread.sleep(500);

// 因为4和5之间的间隔小于指定时间1s，所以前1次数据（4）会被抛弃，5会被保留

e.onNext(5);

Thread.sleep(500);

// 因为5和6之间的间隔小于指定时间1s，所以前1次数据（5）会被抛弃，6会被保留

e.onNext(6);

Thread.sleep(1500);

// 因为6和Complete实践之间的间隔大于指定时间1s，所以之前被保留的6事件将发出

e.onComplete();

}

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

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Integer value) {

Log.d(TAG, "接收到了事件"+ value );

}

@Override

public void onError(Throwable e) {}

@Override

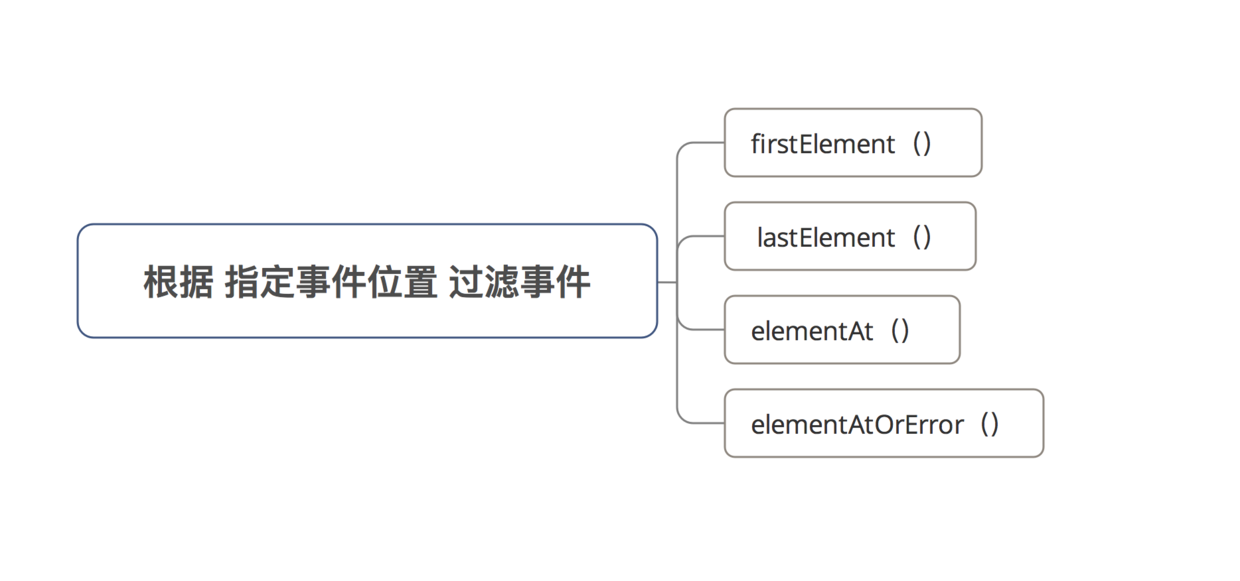
public void onComplete() {}

});

打印结果：2，3，6

**4.4 根据 指定事件位置 过滤事件**

* 需求场景：通过设置指定的位置，过滤在该位置的事件
* 对应操作符类型



**firstElement / lastElement操作符**：仅选取第1个元素 / 最后一个元素

具体使用：

// 获取第1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);

}

});

// 获取最后1个元素

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

.lastElement()

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"获取到的最后1个事件是： "+ integer);

}

});

**elementAt操作符**：指定接收某个元素（通过 索引值 确定）

注：允许越界，即获取的位置索引 ＞ 发送事件序列长度

具体使用：

// 使用1：获取位置索引 = 2的 元素

// 位置索引从0开始

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

.elementAt(2)

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"获取到的事件元素是： "+ integer);

}

});

// 使用2：获取的位置索引 ＞ 发送事件序列长度时，设置默认参数

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

.elementAt(6,10)

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

Log.d(TAG,"获取到的事件元素是： "+ integer);

}

});

打印结果：3，10

**elementAtOrErro操作符**：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

具体使用：

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

.elementAtOrError(6)

.subscribe(new Consumer<Integer>() {

@Override

public void accept( Integer integer) throws Exception {

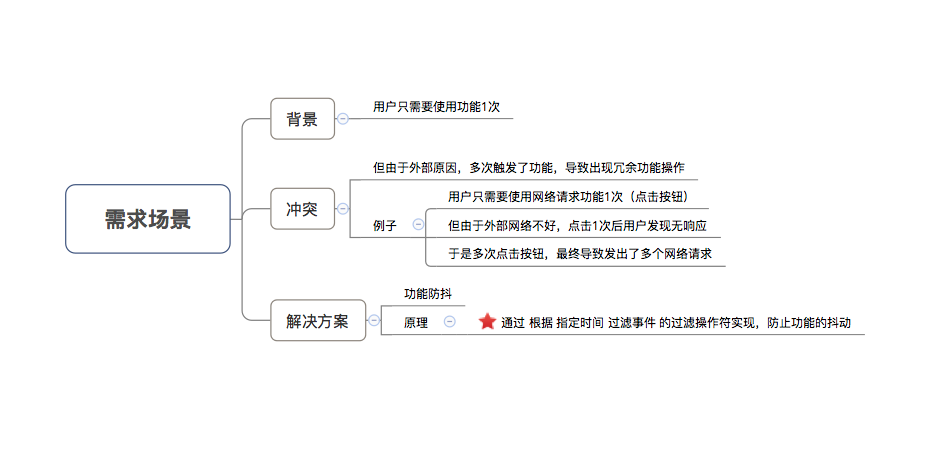
Log.d(TAG,"获取到的事件元素是： "+ integer);

}

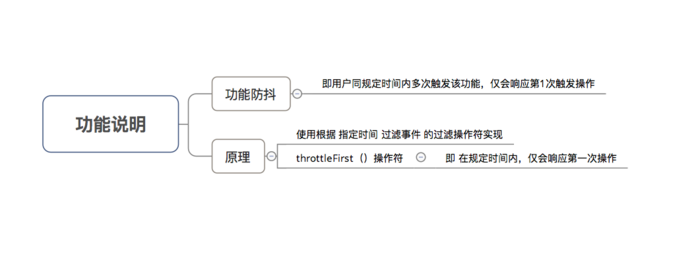
});

##### 过滤操作符在实际开发中的使用

* 功能防抖



原理：

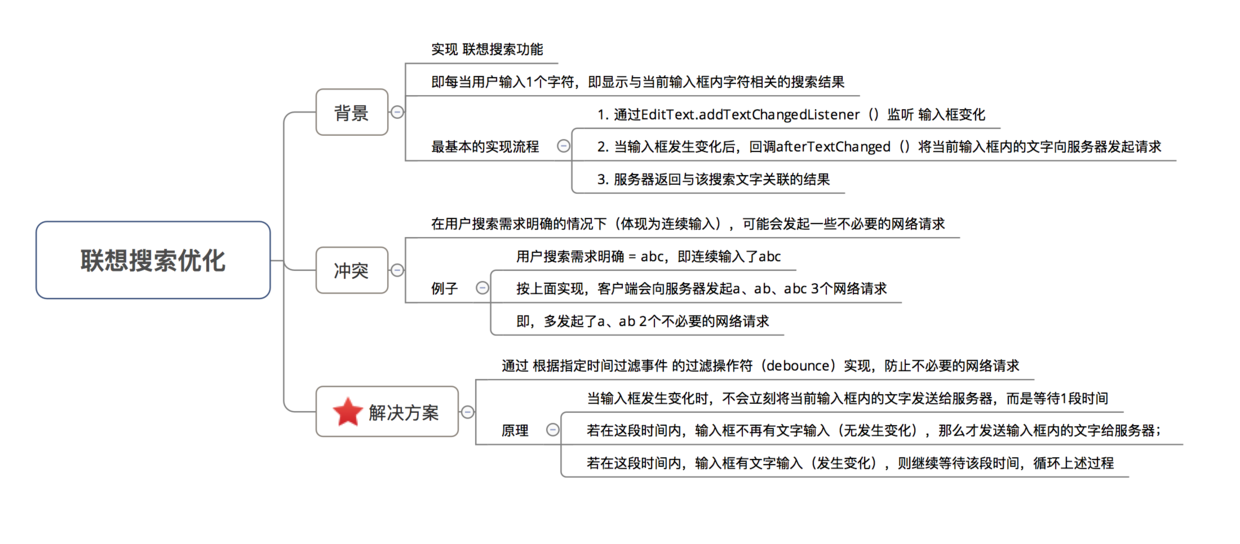


具体实现：

RxView.clicks(button)  
 .throttleFirst(2, TimeUnit.SECONDS)  
 // 才发送 2s内第1次点击按钮的事件  
 .subscribe(new Observer<Object>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 }  
  
 @Override  
 public void onNext(Object value) {  
 Log.d(TAG, "发送了网络请求");  
 }  
  
 @Override  
 public void onError(Throwable e) {}  
  
 @Override  
 public void onComplete() {}  
 });

* 联想搜索优化

场景说明：



具体实现：

/\*  
\* 说明  
\* 1. 此处采用了RxBinding：RxTextView.textChanges(name) = 对对控件数据变更进行监听（功能类似TextWatcher）  
\* 2. 传入EditText控件，输入字符时都会发送数据事件（此处不会马上发送，因为使用了debounce（））   
\* 3. 采用skip(1)原因：跳过 第1次请求 = 初始输入框的空字符状态   
\* \*/   
RxTextView.textChanges(ed)   
 .debounce(1, TimeUnit.SECONDS)  
 .skip(1)   
 .observeOn(AndroidSchedulers.mainThread())   
 .subscribe(new Observer<CharSequence>() {   
 @Override   
 public void onSubscribe(Disposable d) { }   
 @Override   
 public void onNext(CharSequence charSequence) {   
 tv.setText("发送给服务器的字符 = " + charSequence.toString());   
 }   
 @Override   
 public void onError(Throwable e) {}   
 @Override   
 public void onComplete() {}   
 });

##### 6.7.3.5 条件/布尔操作符

* 作用：通过设置函数，判断被观察者（Observable）发送的事件是否符合条件
* 类型：



**all（）操作符**：判断发送的每项数据是否都满足 设置的函数条件

具体代码：

Observable.just(1,2,3,4,5,6)

.all(new Predicate<Integer>(){

@Override

public boolean test( Integer integer) throws Exception {

return (integer<=10);

// 该函数用于判断Observable发送的10个数据是否都满足integer<=10

}

}).subscribe(new Consumer<Boolean>() {

@Override

public void accept(Boolean aBoolean) throws Exception {

Log.d(TAG,"result is "+ aBoolean);

}

});

**takeWhile（）操作符**：判断发送的每项数据是否满足 设置函数条件

具体代码：

// 1. 每1s发送1个数据 = 从0开始，递增1，即0、1、2、3

Observable.interval(1, TimeUnit.SECONDS)

// 2. 通过takeWhile传入一个判断条件

.takeWhile(new Predicate<Long>(){

@Override

public boolean test( Long integer) throws Exception {

return (integer<3);

}

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

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Long value) {

Log.d(TAG,"发送了事件 "+ value);

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

**skipWhile（）操作符**：判断发送的每项数据是否满足 设置函数条件，直到该判断条件为false时，才开始发送Observable的数据

代码实现：

// 1. 每隔1s发送1个数据 = 从0开始，每次递增1

Observable.interval(1, TimeUnit.SECONDS)

.skipWhile(new Predicate<Long>(){

@Override

public boolean test( Long aLong) throws Exception {

return (aLong<5);

// 直到判断条件不成立 = false = 发射的数据≥5，才开始发送数据

}

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

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Long value) {

Log.d(TAG,"发送了事件 "+ value);

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

打印结果：5，6，7...

**takeUntil（）操作符**：执行到某个条件时，停止发送事件

具体代码：

// 1. 每1s发送1个数据 = 从0开始，递增1，即0、1、2、3

Observable.interval(1, TimeUnit.SECONDS)

.takeUntil(new Predicate<Long>(){

@Override

public boolean test( Long integer) throws Exception {

return (integer>3);

// 返回true时，就停止发送事件

// 当发送的数据满足>3时，就停止发送Observable的数据

}

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

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Long value) {

Log.d(TAG,"发送了事件 "+ value);

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

打印结果：0，1，2，3，4

**skipUntil（）操作符**：等到 skipUntil（） 传入的Observable开始发送数据，（原始）第1个Observable的数据才开始发送数据

具体代码：

// （原始）第1个Observable：每隔1s发送1个数据 = 从0开始，每次递增1

Observable.interval(1, TimeUnit.SECONDS)

// 第2个Observable：延迟5s后开始发送1个Long型数据

.skipUntil(Observable.timer(5, TimeUnit.SECONDS))

.subscribe(new Observer<Long>() {

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Long value) {

Log.d(TAG, "接收到了事件"+ value );

}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

测试结果：5s后（ skipUntil（） 传入的Observable开始发送数据），（原始）第1个Observable的数据才开始发送 4，5，6...

**SequenceEqual（）操作符**：判定两个Observables需要发送的数据是否相同

具体实现：

Observable.sequenceEqual(

Observable.just(4,5,6),

Observable.just(4,5,6)

)

.subscribe(new Consumer<Boolean>() {

@Override

public void accept( Boolean aBoolean) throws Exception {

Log.d(TAG,"2个Observable是否相同："+ aBoolean);

// 输出返回结果

}

});

**contains（）操作符**：判断发送的数据中是否包含指定数据

代码实现：

Observable.just(1,2,3,4,5,6)

.contains(4)

.subscribe(new Consumer<Boolean>() {

@Override

public void accept(Boolean aBoolean) throws Exception {

Log.d(TAG,"result is "+ aBoolean);

// 输出返回结果

}

});

**isEmpty（）操作符**：判断发送的数据是否为空

代码实现：

Observable.just(1,2,3,4,5,6)

.isEmpty() // 判断发送的数据中是否为空

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

@Override

public void call(Boolean aBoolean) {

Log.d(TAG,"result is "+ aBoolean);

// 输出返回结果

}

});

**amb（）操作符**：当需要发送多个 Observable时，只发送 先发送数据的Observable的数据，而其余 Observable则被丢弃

代码实现：

// 设置2个需要发送的Observable & 放入到集合中

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

// 第1个Observable延迟1秒发射数据

list.add( Observable.just(1,2,3).delay(1,TimeUnit.SECONDS));

// 第2个Observable正常发送数据

list.add( Observable.just(4,5,6));

// 一共需要发送2个Observable的数据

// 但由于使用了amba（）,所以仅发送先发送数据的Observable

// 即第二个（因为第1个延时了）

Observable.amb(list).subscribe(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.e(TAG, "接收到了事件 "+integer);

}

});

打印结果：4，5，6

**defaultIfEmpty（）操作符**：在不发送任何有效事件（ Next事件）、仅发送了 Complete 事件的前提下，发送一个默认值

代码实现：

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

// 仅发送Complete事件

e.onComplete();

}

}).defaultIfEmpty(10) // 若仅发送了Complete事件，默认发送 值 = 10

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {}

@Override

public void onNext(Integer value) {}

@Override

public void onError(Throwable e) {}

@Override

public void onComplete() {}

});

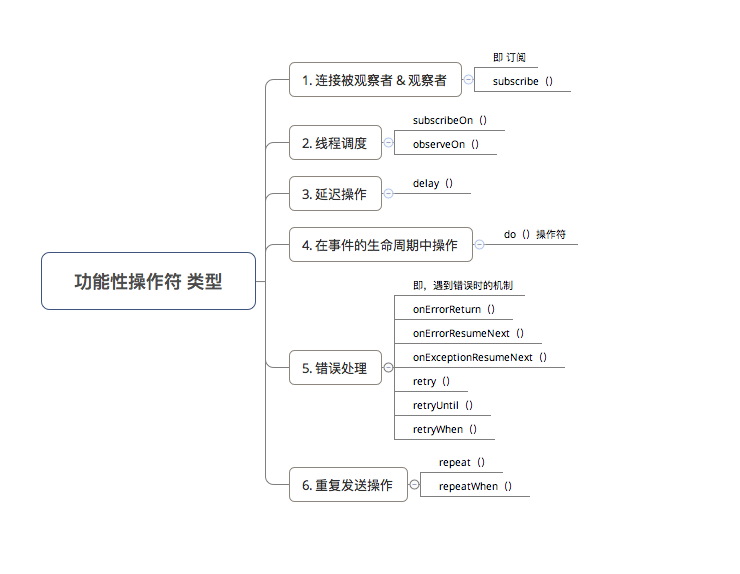
总结所有的条件/布尔操作符：



##### 6.7.3.5 功能性操作符

作用：辅助被观察者在发送事件时实现一些功能性需求，如错误处理、线程调度等

类型：常见的功能性操作符主要有



**5.1 线程调度**

需求场景：快速、方便指定 & 控制被观察者 & 观察者的工作线程

作用：指定被观察者/观察者的工作线程类型

解决问题：为了实现异步操作，需要对Rxjava进行线程控制(调度/切换)

实现方式：采用Rxjava内置的线程调度器(Scheduler)，即通过功能性操作符subscribeOn()&ObserveOn()实现

线程类型：

* Schedulers.immediate()：当前线程 = 不指定线程，默认
* AndroidSchedulers.mainThread()：Android主线程，操作UI
* Schedulers.newThread()：常规新线程，耗时等操作
* Schedulers.io()：io操作线程，网络请求、读写文件等io密集型操作
* Schedulers.computation()：CPU计算操作线程，大量计算操作

实现过程如下：

observable  
 .subscribeOn(Schedulers.newThread())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(observer);

特别注意：

* 若observable.subscribeOn()多次指定被观察者执行线程，则只有第一次指定有效，其余的指定线程无效
* 若observeOn多次指定观察者接受和响应的线程，则每次指定都是有效的，即每指定一次都会进行一次线程的切换

实际应用：

**应用程序崩溃问题**

* 背景：在发送网络请求时 退出当前Activity
* 冲突：此时如果回到主线程更新 UI，App会崩溃
* 解决方案：当 Activity退出时，调用 Disposable.dispose()切断观察者和被观察者的连接，使得观察者无法收到事件 & 响应事件

当出现多个Disposable时，可采用RxJava内置容器CompositeDisposable进行统一管理

// 添加Disposable到CompositeDisposable容器

CompositeDisposable.add()

// 清空CompositeDisposable容器

CompositeDisposable.clear()

**5.2 延迟操作**

需求场景：在被观察者发送事件前，进行一些延迟的操作

对应操作符：delay()

delay()操作符的多个重载方法：

// 1. 指定延迟时间// 参数1 = 时间；参数2 = 时间单位

delay(long delay,TimeUnit unit)

// 2. 指定延迟时间 & 调度器// 参数1 = 时间；参数2 = 时间单位；参数3 = 线程调度器

delay(long delay,TimeUnit unit,mScheduler scheduler)

// 3. 指定延迟时间 & 错误延迟// 错误延迟，即：若存在Error事件，则如常执行，执行后再抛出错误异常// 参数1 = 时间；参数2 = 时间单位；参数3 = 错误延迟参数

delay(long delay,TimeUnit unit,boolean delayError)

// 4. 指定延迟时间 & 调度器 & 错误延迟// 参数1 = 时间；参数2 = 时间单位；参数3 = 线程调度器；参数4 = 错误延迟参数

delay(long delay,TimeUnit unit,mScheduler scheduler,boolean delayError)

使用实例：

Observable.just(1, 2, 3)

.delay(3, TimeUnit.SECONDS)

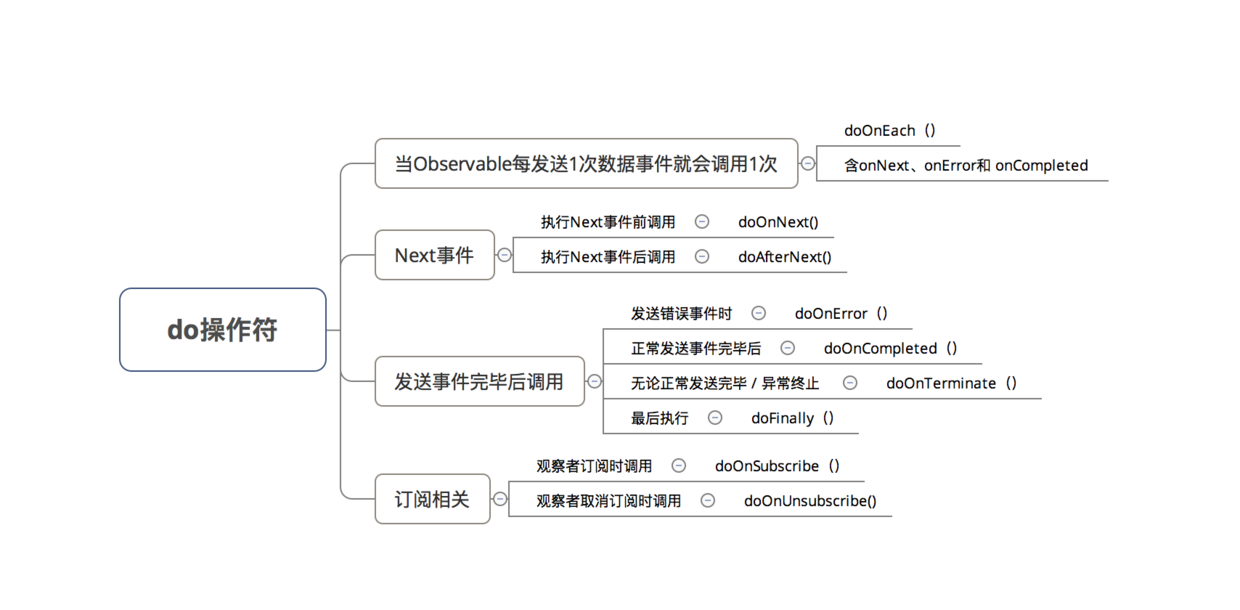
.subscribe(observer);

**5.3 在事件的生命周期中使用**

需求场景：在事件发送&接受的整个生命周期中进行操作，如发送事件前的初始化和事件发送后的回调请求等

do()操作符：在某个事件的生命周期中使用

do()操作符有很多：



具体使用实例：

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 Throwable("发生错误了"));

}

})

// 1. 当Observable每发送1次数据事件就会调用1次

.doOnEach(new Consumer<Notification<Integer>>() {

@Override

public void accept(Notification<Integer>

integerNotification) throws Exception {

Log.d(TAG, "doOnEach: " + integerNotification.getValue());

}

})

// 2. 执行Next事件前调用

.doOnNext(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.d(TAG, "doOnNext: " + integer);

}

})

// 3. 执行Next事件后调用

.doAfterNext(new Consumer<Integer>() {

@Override

public void accept(Integer integer) throws Exception {

Log.d(TAG, "doAfterNext: " + integer);

}

})

// 4. Observable正常发送事件完毕后调用

.doOnComplete(new Action() {

@Override

public void run() throws Exception {

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

}

})

// 5. Observable发送错误事件时调用

.doOnError(new Consumer<Throwable>() {

@Override

public void accept(Throwable throwable) throws Exception {

Log.d(TAG, "doOnError: " + throwable.getMessage());

}

})

// 6. 观察者订阅时调用

.doOnSubscribe(new Consumer<Disposable>() {

@Override

public void accept(@NonNull Disposable disposable)

throws Exception {

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

}

})

// 7. Observable发送事件完毕后调用，无论正常发送完毕 / 异常终止

.doAfterTerminate(new Action() {

@Override

public void run() throws Exception {

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

}

})

// 8. 最后执行

.doFinally(new Action() {

@Override

public void run() throws Exception {

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

}

})

.subscribe(observer);

打印结果如下：

doOnSubscribe

doOnEach ：1

doOnNext ：1

接收到事件1

doAfterNext ：1

doOnEach ：2

doOnNext ：2

接收到事件2

doAfterNext ：2

doOnEach ：3

doOnNext ：3

接收到事件3

doAfterNext ：3

doOnEach ：null

doOnError : 发生错误了

对error事件做出响应

doFinally

doAfterTerminate

**5.4 错误处理**

需求场景：发送事件过程中，遇到错误时的处理机制

对应操作符类型：



**OnErrorReturn()操作符**：遇到错误时，发送一个特殊事件，正常终止，可捕获在它之前发生的异常

具体操作：

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

e.onNext(1);

e.onNext(2);

e.onError(new Throwable("发生错误了"));

}

})

.onErrorReturn(new Function<Throwable, Integer>() {

@Override

public Integer apply(@NonNull Throwable throwable)

throws Exception {

// 捕捉错误异常

Log.e(TAG, "错误: "+throwable.toString() );

return 666;

}

})

.subscribe(observer);

**onErrorResumeNext（）操作符**：遇到错误时，发送1个新的Observable

注：

* onErrorResumeNext()拦截的错误 = Throwable
* 若需拦截Exception请用onExceptionResumeNext()
* 若onErrorResumeNext()拦截的错误 = Exception，则会将错误传递给观察者的onError方法

具体使用：

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

e.onNext(1);

e.onNext(2);

e.onError(new Throwable("发生错误了"));

}

})

.onErrorResumeNext(new Function<Throwable,

ObservableSource<? extends Integer>>() {

@Override

public ObservableSource<? extends Integer> apply(

@NonNull Throwable throwable) throws Exception {

// 1. 捕捉错误异常

Log.e(TAG, “错误: "+throwable.toString() );

// 2. 发生错误事件后，发送一个新的被观察者 & 发送事件序列

return Observable.just(11,22);

}

})

.subscribe(observer);

**onExceptionResumeNext操作符**：遇到错误时，发送1个新的Observable

注：

* onExceptionResumeNext（）拦截的错误 = Exception；
* 若需拦截Throwable请用onErrorResumeNext（）
* 若onExceptionResumeNext（）拦截的错误 = Throwable，则会将错误传递给观察者的onError方法

具体使用：

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

}

})

.onExceptionResumeNext(new Observable<Integer>() {

@Override

protected void subscribeActual(

Observer<? super Integer> observer) {

observer.onNext(11);

observer.onNext(22);

observer.onComplete();

}

})

.subscribe(observer);

**retry（）操作符**：重试，即当出现错误时，让被观察者（Observable）重新发射数据

* 接收到 onError（）时，重新订阅 & 发送事件
* Throwable 和 Exception都可拦截

重载类型：

<-- 1. retry（） -->

// 作用：出现错误时，让被观察者重新发送数据

// 注：若一直错误，则一直重新发送

<-- 2. retry（long time） -->

// 作用：出现错误时，让被观察者重新发送数据（具备重试次数限制

// 参数 = 重试次数

<-- 3. retry（Predicate predicate） -->

// 作用：出现错误后，判断是否需要重新发送数据（若需要重新发送& 持续遇到错误，则持续重试）

// 参数 = 判断逻辑

<-- 4. retry（new BiPredicate<Integer, Throwable>） -->

// 作用：出现错误后，判断是否需要重新发送数据（若需要重新发送 & 持续遇到错误，则持续重试

// 参数 = 判断逻辑（传入当前重试次数 & 异常错误信息）

<-- 5. retry（long time,Predicate predicate） -->

// 作用：出现错误后，判断是否需要重新发送数据（具备重试次数限制

// 参数 = 设置重试次数 & 判断逻辑

具体使用：

<-- 1. retry（） -->// 作用：出现错误时，让被观察者重新发送数据// 注：若一直错误，则一直重新发送

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

@Override

public void subscribe(ObservableEmitter<Integer> e)

throws Exception {

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

e.onNext(3);

}

})

.retry()

// 遇到错误时，让被观察者重新发射数据（若一直错误，则一直重新发送)

.subscribe(observer);

<-- 2. retry（long time） -->// 作用：出现错误时，让被观察者重新发送数据（具备重试次数限制// 参数 = 重试次数

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

@Override

public void subscribe(ObservableEmitter<Integer> e) {

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

e.onNext(3);

}

})

.retry(3) // 设置重试次数 = 3次

.subscribe(observer);

<-- 3. retry（Predicate predicate） -->// 作用：出现错误后，判断是否需要重新发送数据（若需要重新发送& 持续遇到错误，则持续重试）// 参数 = 判断逻辑

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

@Override

public void subscribe(ObservableEmitter<Integer> e) {

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

e.onNext(3);

}

})

// 拦截错误后，判断是否需要重新发送请求

.retry(new Predicate<Throwable>() {

@Override

public boolean test(@NonNull Throwable throwable){

// 捕获异常

Log.e(TAG, "retry错误: "+throwable.toString());

//返回false = 不重新重新发送数据 & 调用观察者的onError结束

//返回true = 重新发送请求（若持续遇到错误，就持续重新发送）

return true;

}

})

.subscribe(observer);

<-- 4. retry（new BiPredicate<Integer, Throwable>） -->// 作用：出现错误后，判断是否需要重新发送数据（若需要重新发送 & 持续遇到错误，则持续重试// 参数 = 判断逻辑（传入当前重试次数 & 异常错误信息）

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

@Override

public void subscribe(ObservableEmitter<Integer> e){

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

e.onNext(3);

}

})

// 拦截错误后，判断是否需要重新发送请求

.retry(new BiPredicate<Integer, Throwable>() {

@Override

public boolean test(@NonNull Integer integer,

@NonNull Throwable throwable) throws Exception {

// 捕获异常

Log.e(TAG, "异常错误 = "+throwable.toString());

// 获取当前重试次数

Log.e(TAG, "当前重试次数 = "+integer);

//返回false = 不重新重新发送数据 & 调用观察者的onError结束

//返回true = 重新发送请求（若持续遇到错误，就持续重新发送）

return true;

}

})

.subscribe(observer);

<-- 5. retry（long time,Predicate predicate） -->// 作用：出现错误后，判断是否需要重新发送数据（具备重试次数限制// 参数 = 设置重试次数 & 判断逻辑

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

@Override

public void subscribe(ObservableEmitter<Integer> e){

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

e.onNext(3);

}

})

// 拦截错误后，判断是否需要重新发送请求

.retry(3, new Predicate<Throwable>() {

@Override

public boolean test(@NonNull Throwable throwable){

// 捕获异常

Log.e(TAG, "retry错误: "+throwable.toString());

//返回false = 不重新重新发送数据 & 调用观察者的onError（）结束

//返回true = 重新发送请求（最多重新发送3次）

return true;

}

})

.subscribe(observer);

**retryUntil（）操作符**：出现错误后，判断是否需要重新发送数据

* 若需要重新发送 & 持续遇到错误，则持续重试
* 作用类似于retry（Predicate predicate）

具体使用   
具体使用类似于retry（Predicate predicate），唯一区别：返回 true 则不重新发送数据事件。此处不作过多描述

**retryWhen操作符**：遇到错误时，将发生的错误传递给一个新的被观察者（Observable），并决定是否需要重新订阅原始被观察者（Observable）& 发送事件

具体使用：

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

@Override

public void subscribe(ObservableEmitter<Integer> e){

e.onNext(1);

e.onNext(2);

e.onError(new Exception("发生错误了"));

e.onNext(3);

}

})

// 遇到error事件才会回调

.retryWhen(new Function

<Observable<Throwable>, ObservableSource<?>>() {

@Override

public ObservableSource<?> apply(

@NonNull Observable<Throwable> throwableObservable){

// 参数Observable<Throwable>中的泛型 = 上游操作符抛出的异常，可通过该条件来判断异常的类型

// 返回Observable<?> = 新的被观察者 Observable（任意类型）

// 此处有两种情况：

// 1. 若 新的被观察者 Observable发送的事件 = Error事件，那么 原始Observable则不重新发送事件：

// 2. 若 新的被观察者 Observable发送的事件 = Next事件 ，那么原始的Observable则重新发送事件：

return throwableObservable.flatMap(

new Function<Throwable, ObservableSource<?>>() {

@Override

public ObservableSource<?> apply(

@NonNull Throwable throwable) throws Exception {

// 1. 若返回的Observable发送的事件 = Error事件，则原始的Observable不重新发送事件

// 该异常错误信息可在观察者中的onError（）中获得

return Observable.error(

new Throwable("retryWhen终止啦"));

// 2. 若返回的Observable发送的事件 = Next事件，则原始的Observable重新发送事件（若持续遇到错误，则持续重试）

// return Observable.just(1);

}

});

}

})

.subscribe(observer);

**5.5 重复发送**

需求场景：需要不断地发送被观察者事件

对应操作符：repeat() & repeatWhen()

**repeat()操作符**：无条件地、重复地发送被观察者事件，重载方法可以设置重复创建次数

代码实现：

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

.repeat(3) // 重复创建次数 = 3次

.subscribe(new Observer<Integer>() {

@Override

public void onSubscribe(Disposable d) {

Log.d(TAG, "开始采用subscribe连接");

}

@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");

}

});

打印结果：1，2，3，4，1，2，3，4，1，2，3，4

**repeatWhen操作符**：有条件地、重复发送 被观察者事件

原理：   
将原始 Observable 停止发送事件的标识（Complete（） / Error（））转换成1个 Object 类型数据传递给1个新被观察者（Observable），以此决定是否重新订阅 & 发送原来的 Observable

* 若新被观察者（Observable）返回1个Complete / Error事件，则不重新订阅 & 发送原来的 Observable
* 若新被观察者（Observable）返回其余事件时，则重新订阅 & 发送原来的 Observable

具体使用：

Observable.just(1,2,4)

.repeatWhen(new Function<Observable<Object>

, ObservableSource<?>>() {

@Override

public ObservableSource<?> apply(

Observable<Object> objectObservable) throws Exception {

return objectObservable.flatMap(

new Function<Object, ObservableSource<?>>() {

@Override

public ObservableSource<?> apply(

@NonNull Object throwable) throws Exception {

// 情况1

return Observable.empty();

// Observable.empty() = 发送Complete事件，但不会回调观察者的onComplete（）

// return Observable.error(new Throwable(""));

// 返回Error事件 = 回调onError（）事件，并接收传过去的错误信息。

// 情况2

// return Observable.just(1);

}

});

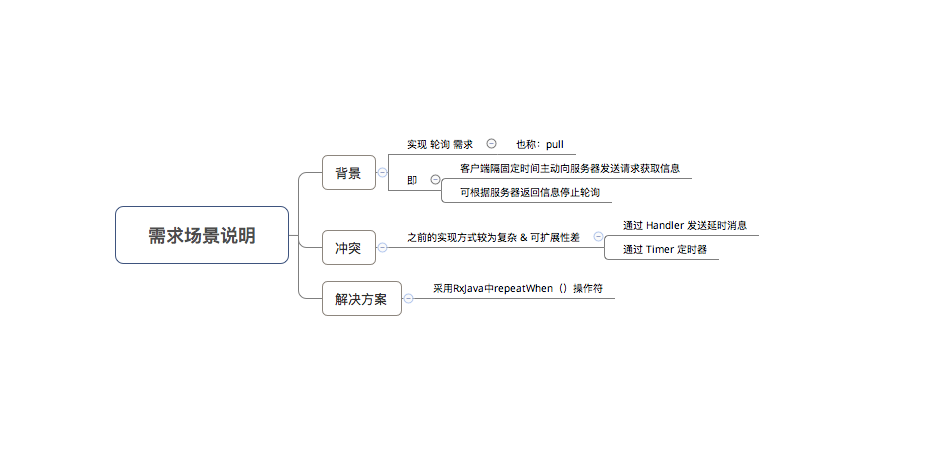
}

})

##### 实际开发中的使用

1. 轮询

需求场景：



具体实现：

observable.repeatWhen(  
 new Function<Observable<Object>, ObservableSource<?>>() {  
 @Override  
 public ObservableSource<?> apply(

Observable<Object> objectObservable) throws Exception {  
   
 return objectObservable.flatMap(

new Function<Object, ObservableSource<?>>() {  
 @Override  
 public ObservableSource<?> apply(

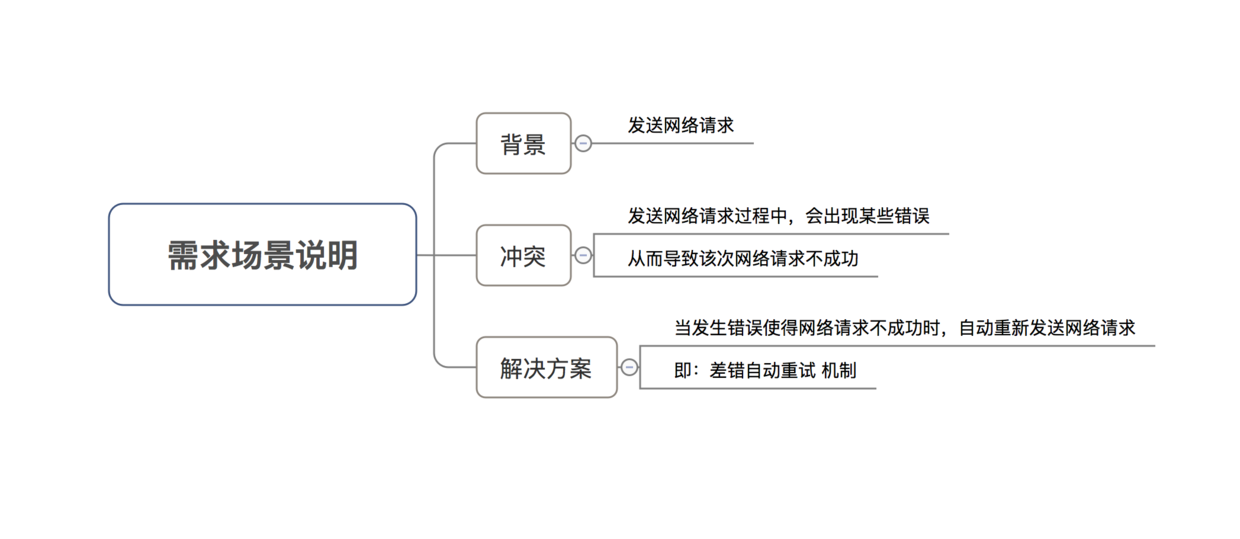
@NonNull Object throwable) throws Exception {  
  
 if (i > 3) {  
 return Observable.error(new Throwable("轮询结束"));  
 }  
 return Observable

.just(1)

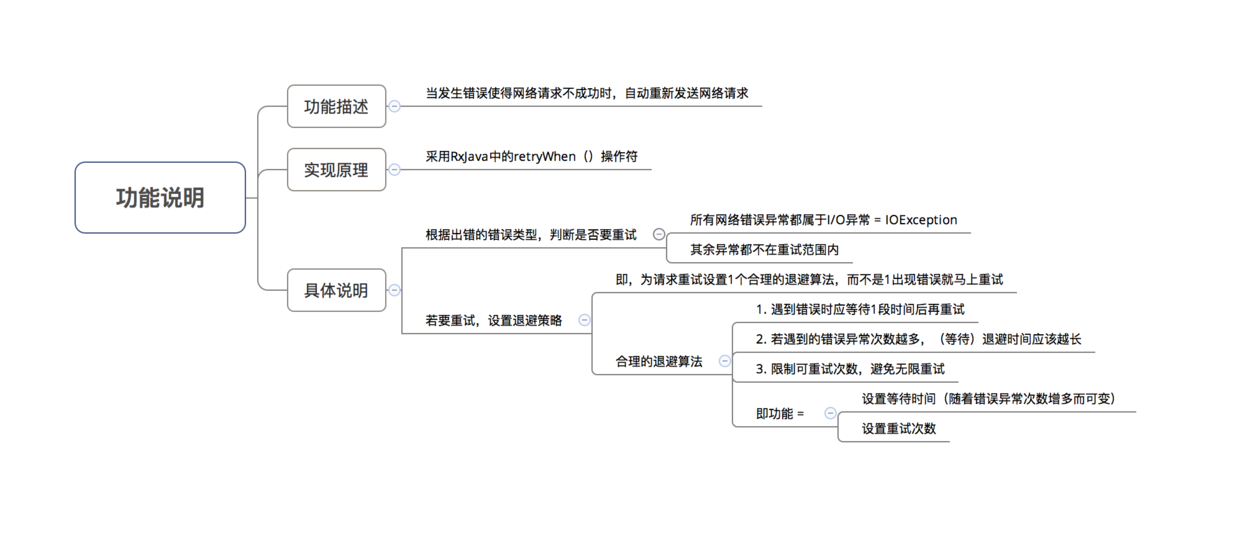
.delay(2000, TimeUnit.MILLISECONDS);  
 }  
 });  
 }  
 })  
 .subscribeOn(Schedulers.io())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Observer<Translation>() {  
 @Override  
 public void onSubscribe(Disposable d) {  
 }  
  
 @Override  
 public void onNext(Translation result) {  
 result.show();  
 i++;  
 }  
  
 @Override  
 public void onError(Throwable e) {  
 Log.d(TAG, e.toString());  
 }  
  
 @Override  
 public void onComplete() {  
 }  
 });

1. 发送网络请求时的差错重试机制

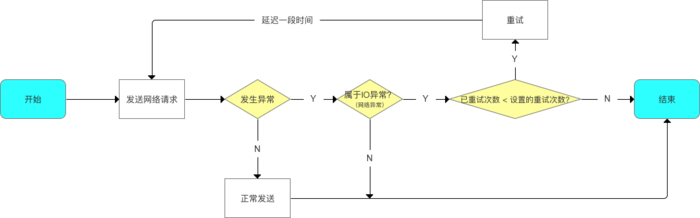
需求场景：



功能说明：



功能逻辑：



具体实现：

observable.retryWhen(

new Function<Observable<Throwable>, ObservableSource<?>>() {  
 @Override  
 public ObservableSource<?> apply(

@NonNull Observable<Throwable> throwableObservable){  
 // 参数Observable<Throwable>中的泛型 = 上游操作符抛出的异常，可通过该条件来判断异常的类型  
 return throwableObservable.flatMap(

new Function<Throwable, ObservableSource<?>>() {  
 @Override  
 public ObservableSource<?> apply(

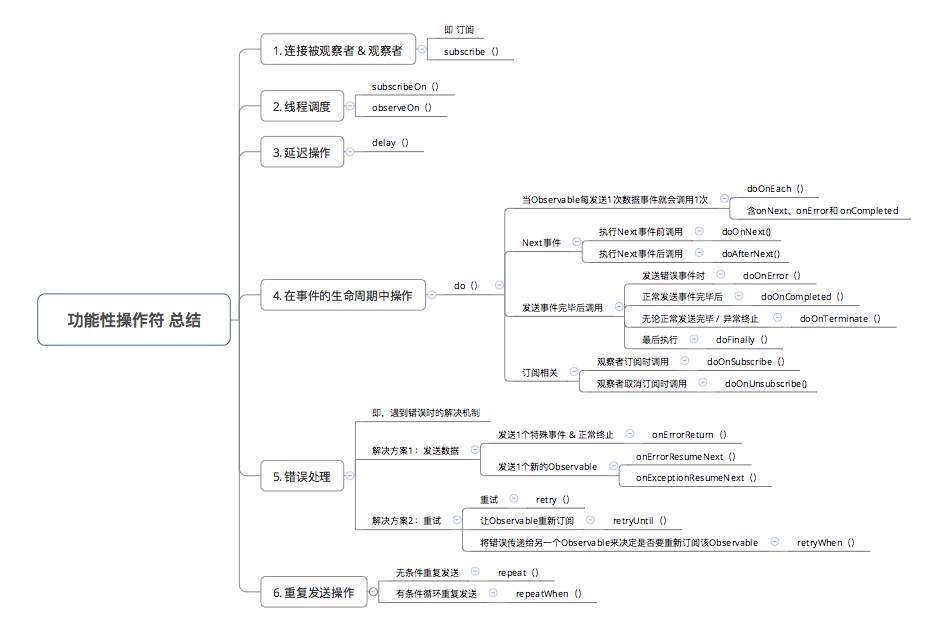
@NonNull Throwable throwable) throws Exception {  
 // 输出异常信息  
 Log.d(TAG, "发生异常 = " + throwable.toString());  
 /\*\*  
 \* 需求1：根据异常类型选择是否重试  
 \* 即，当发生的异常 = 网络异常 = IO异常 才选择重试  
 \*  
 \* \*/  
 if (throwable instanceof IOException) {  
 Log.d(TAG, "属于IO异常，需重试");  
 /\*\*  
 \*  
 \* 需求2：限制重试次数  
 \* 即，当已重试次数 < 设置的重试次数，才选择重试  
 \*/  
 if (currentRetryCount < maxConnectCount) {  
 // 记录重试次数  
 currentRetryCount++;  
 Log.d(TAG, "重试次数 = " + currentRetryCount);  
 /\*\*  
 \* 需求2：实现重试  
 \* 通过返回的Observable发送的事件 = Next事件，从而使得retryWhen（）重订阅，最终实现重试功能  
 \* 需求3：延迟1段时间再重试  
 \* 采用delay操作符 = 延迟一段时间发送，以实现重试间隔设置  
 \* 需求4：遇到的异常越多，时间越长  
 \* 在delay操作符的等待时间内设置 = 每重试1次，增多延迟重试时间1s  
 \*/  
 // 设置等待时间  
 waitRetryTime = 1000 + currentRetryCount \* 1000;  
 Log.d(TAG, "等待时间 =" + waitRetryTime);  
 return Observable

.just(1)

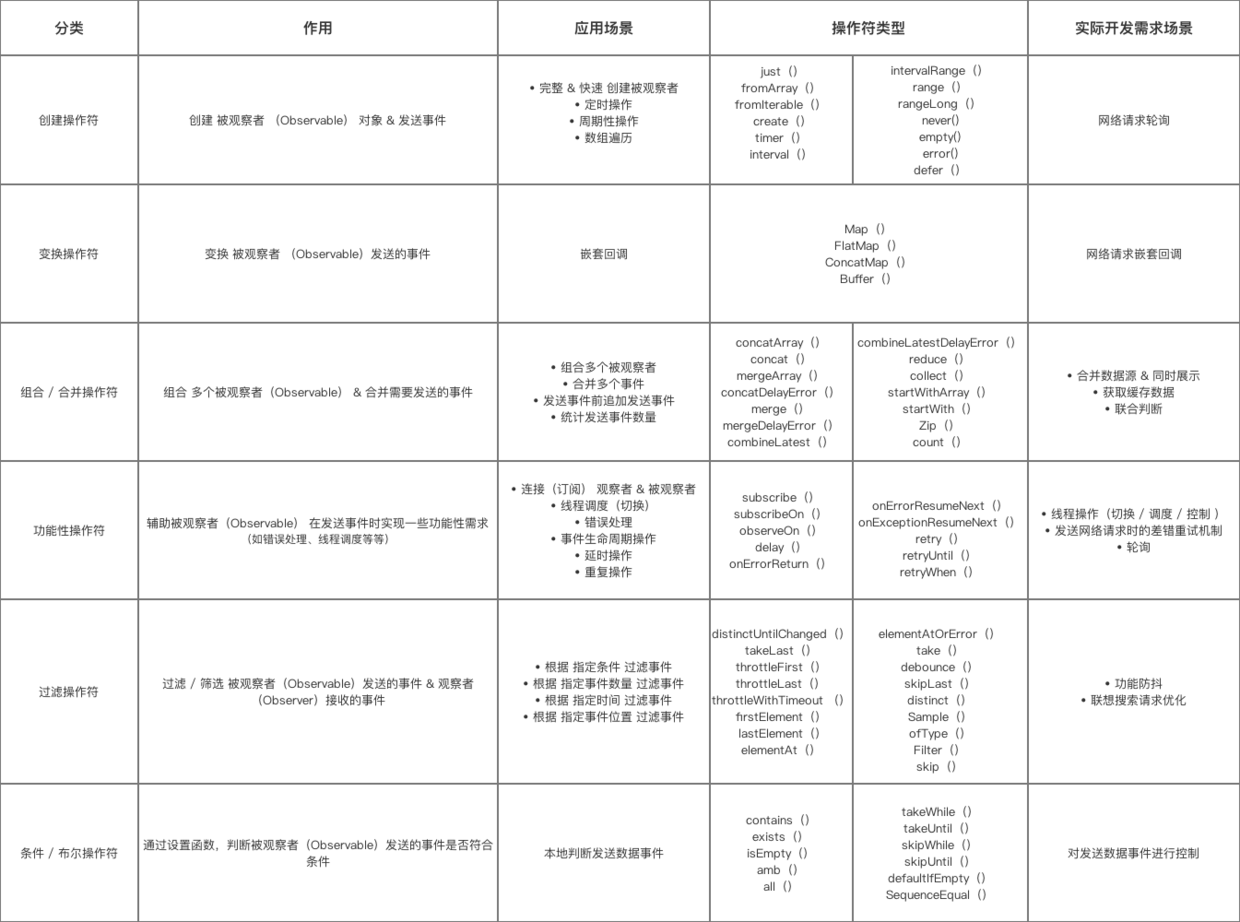
.delay(waitRetryTime, TimeUnit.MILLISECONDS);  
 } else {  
 // 若重试次数已 > 设置重试次数，则不重试  
 // 通过发送error来停止重试（可在观察者的onError（）中获取信息）  
 return Observable.error(

new Throwable("重试次数已超过设置次数 = " + currentRetryCount + "，即 不再重试"));  
 }  
 }  
 // 若发生的异常不属于I/O异常，则不重试  
 // 通过返回的Observable发送的事件 = Error事件 实现（可在观察者的onError（）中获取信息）  
 else {  
 return Observable.error(new Throwable("发生了非网络异常（非I/O异常）"));  
 }  
 }  
 });  
 }  
}).subscribeOn(observable);

##### 功能性操作符总结



##### 总结所有Rxjava操作符：



1. **Backpressure背压**

介绍：RxJava 2.x 最大的改动就是对于 backpressure 的处理，为此将原来的 Observable 拆分成了新的 Observable 和 Flowable

背景：观察者和被观察者之间存在两种订阅关系，同步&&异步

* 同步：两者在同一线程，被观察者发送一个事件，必须等到观察者接受并处理完成后，才能继续发送下一个事件
* 异步：两者在不同线程，被观察者不断发送事件，但是此时的事件不会直接发送到观察者，而是会先发送到缓冲区，等观察者从缓冲区取出事件来进行处理

对于异步订阅关系，存在 被观察者发送事件速度 与 观察者接受事件速度 不匹配 的情况，大多数情况下是 被观察者速度>观察者速度

被观察者 发送事件速度太快，而观察者 来不及接收所有事件，从而导致观察者无法及时响应 / 处理所有发送过来事件的问题，最终导致缓存区溢出、事件丢失 & OOM

如，点击按钮事件：连续过快的点击按钮10次，则只会造成点击2次的效果；

解释：因为点击速度太快了，所以按钮来不及响应

解决办法：采用背压策略

背压策略简介：一种控制事件流速的策略

背压的作用：在异步订阅关系中，控制事件发送&接受的速度

原理：

* 可以提前避免出现事件发送&接受流速不匹配的情况
* 控制观察者接受事件的速度

原理：响应式拉取

即，观察者根据自身实际情况按需接受事件

* 控制被观察者发送事件的速度

原理：反馈控制

即，被观察者根据观察者的接受事件能力从而控制发送事件速度

* 当出现事件发送&接受不匹配时，我们再解决也行
* 采用背压策略模式

面向对象：缓冲区

原理：对超出缓冲区大小的事件进行丢弃、保留、报错的措施

背压策略的具体实现：Flowable(非阻塞式背压策略)

**Flowable的介绍**

1. 对应的观察者变成Subscriber，即Flowable----Subscriber
2. 所有的操作符强制支持背压，Flowable中 大多数操作符与旧的Observable类似
3. 缓冲区策略

* 事件按照发送顺序进入缓冲区，即先发送--想进入缓冲区
* 想进入缓冲区的事件先取出
* 类似队列，实际在ZIP内部的实现=队列
* 默认的缓冲区大小=128

背压的基本实现案例：

Flowable.create(new FlowableOnSubscribe<Integer>() {

@Override

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

emitter.onNext(1);

emitter.onNext(2);

emitter.onNext(3);

emitter.onComplete();

}

}, BackpressureStrategy.ERROR)

.subscribe(new Subscriber<Integer>() {

// 步骤2：创建观察者 = Subscriber & 建立订阅关系

@Override

public void onSubscribe(Subscription s) {

Log.d(TAG, "onSubscribe");

s.request(3);

}

@Override

public void onNext(Integer integer) {}

@Override

public void onError(Throwable t) {}

@Override

public void onComplete() {}

});

##### 具体缓存策略

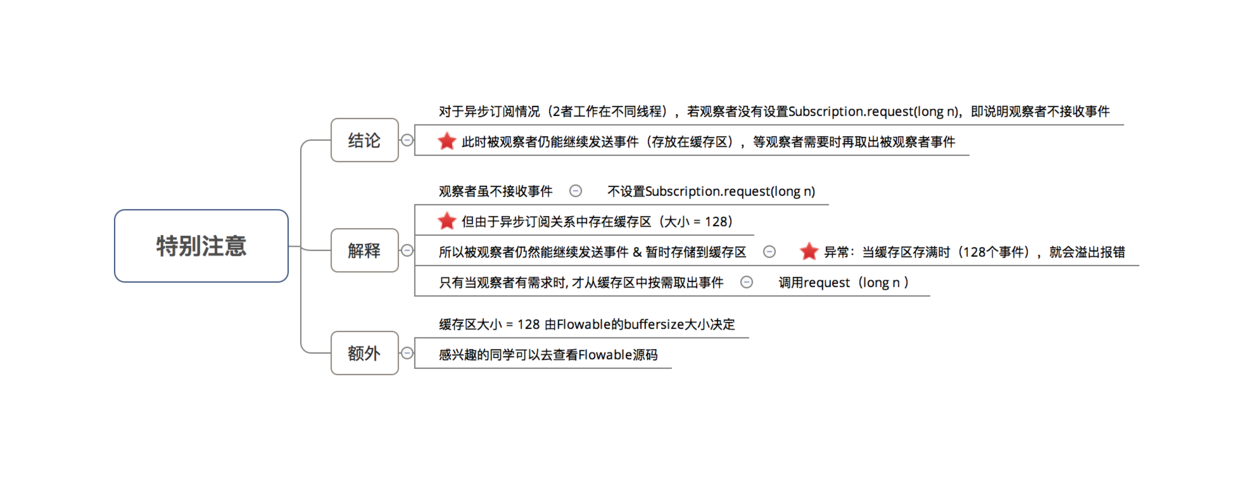
1. **控制观察者接受事件的速度**

1.1 原理：响应式拉取

1.2 实现方式：通过观察者中onSubscribe的Subscription参数的request()方法控制流速

1.3 具体细节：被观察者异步将事件发送到缓冲区，观察者按需接收事件，通过request从缓冲区取出需要的事件数量

1.4 关于拉取过程需要有几个方面注意的，下面图



注意：观察者不接受时，被观察者一直往缓冲区发送事件，直到超出缓冲区大小报错

**2 控制被观察者发送事件的速度**

2.1 原理：反馈控制

2.2 实现方式：通过被观察者中FlowableEmitter类的requested()方法控制流速

public interface FlowableEmitter<T> extends Emitter<T> {

// FlowableEmitter = 1个接口，继承自Emitter

// Emitter接口方法包括：onNext(),onComplete() & onError

long requested();

// 作用：返回当前线程中request（a）中的a值

// 该request（a）则是措施1中讲解的方法，作用 = 设置

....// 仅贴出关键代码

}

2.3 注意：通过Rxjava内部固定调用的被观察者线程中的request(n)，来方向控制被观察者的发送事件速度

被观察者 long requested()的返回值 = 被观察者线程中Rxjava内部调用的request(n)中的n值，n=128、96、0

观察者中request()的返回值 = 观察者线程中request(n)的n值

实际案例：

Flowable.create(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull FlowableEmitter<Integer> emitter) throws Exception {  
  
 Log.e("RxJava", "可接受数量= " + emitter.requested());  
 }  
}, BackpressureStrategy.ERROR)  
 .subscribeOn(Schedulers.io())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 s.request(200);  
 }  
  
 @Override  
 public void onNext(Integer integer) {}  
  
 @Override  
 public void onError(Throwable t) {}  
  
 @Override  
 public void onComplete() {}  
 });

打印结果：可接受数量= 128

修改s.request(200)里n的值，可接受的数量仍然是128

再看一个更复杂的例子

public class MainActivity extends AppCompatActivity {  
  
 Subscription subscription;  
 @Override  
 protected void onCreate(Bundle savedInstanceState) {  
 super.onCreate(savedInstanceState);  
 setContentView(R.layout.activity\_main);  
 flowableMethod();  
 findViewById(R.id.btn).setOnClickListener(new View.OnClickListener() {  
 @Override  
 public void onClick(View v) {  
 subscription.request(40);  
 }  
 });  
 }  
  
 private void flowableMethod() {  
  
 Flowable.create(new FlowableOnSubscribe<Integer>() {  
 @Override  
 public void subscribe(@NonNull FlowableEmitter<Integer> emitter) {  
  
 Log.d("RxJava", "观察者可接收事件数量 = " + emitter.requested());  
 boolean flag; //设置标记位控制  
  
 // 被观察者一共需要发送500个事件  
 for (int i = 0; i < 500; i++) {  
 flag = false;  
  
 // 若requested() == 0则不发送  
 while (emitter.requested() == 0) {  
 if (!flag) {  
 Log.d("RxJava", "不再发送");  
 flag = true;  
 }  
 }  
 Log.d("RxJava", "发送了事件" + i + "，观察者可接收事件数量 = " + emitter.requested());  
 emitter.onNext(i);  
  
 }  
 }  
 }, BackpressureStrategy.ERROR)  
 .subscribeOn(Schedulers.io())  
 .observeOn(AndroidSchedulers.mainThread())  
 .subscribe(new Subscriber<Integer>() {  
 @Override  
 public void onSubscribe(Subscription s) {  
 subscription = s;  
 }  
  
 @Override  
 public void onNext(Integer integer) {  
 Log.d("观察者", "onNext: " + integer);  
 }  
  
 @Override  
 public void onError(Throwable t) {}  
  
 @Override  
 public void onComplete() {}  
 });  
 }  
}

##### 分析结果

* 第一次运行，被观察者直接发送128条数据到缓冲区

观察者可接收事件数量 = 128

发送了事件0，观察者可接收事件数量 = 128

发送了事件1，观察者可接收事件数量 = 127

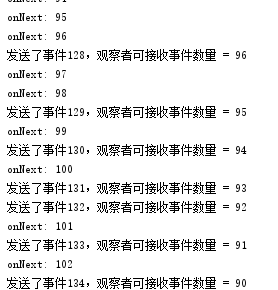
...

发送了事件126，观察者可接收事件数量 = 2

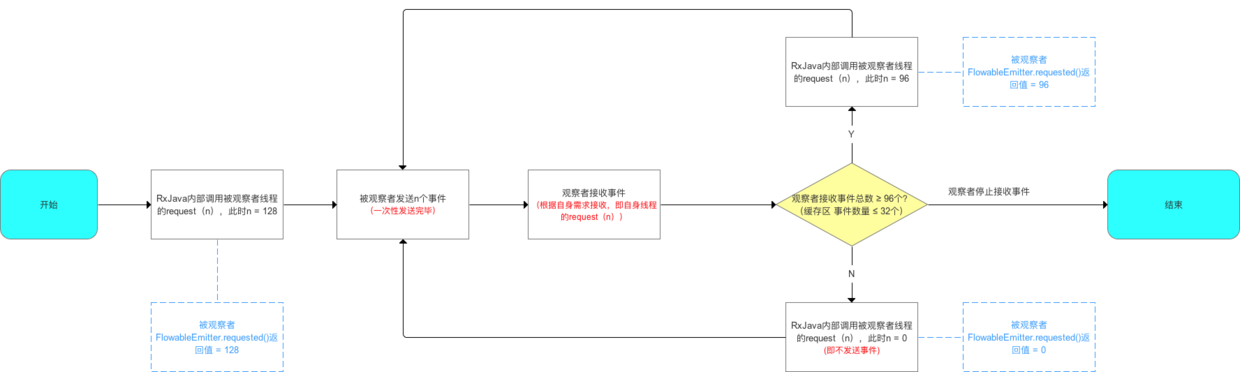
发送了事件127，观察者可接收事件数量 = 1

不再发送

* 点击按钮，观察者开始从缓冲区获取40条数据
* 再点击，继续获取40条数据
* 再点击，继续获取40条数据，此时观察者接受的事件 >= 96，被观察者会继续发送96条数据，而且内部就会继续调用request(96)



可以用下面一张图总结整个过程



##### 3 采用背压策略

3.1 面向对象：缓存区

3.2 作用：当缓存区已经被存满，但是被观察者仍然发送事件，该如何处理的策略

3.3 背压模式类型

* **BackpressureStrategy.ERROR**

问题：发送事件速度 ＞ 接收事件 速度，即流速不匹配

具体表现：出现当缓存区大小存满（默认缓存区大小 = 128）、被观察者仍然继续发送下1个事件时

处理方式：直接抛出异常MissingBackpressureException

* **BackpressureStrategy.MISSING**

问题：发送事件速度 ＞ 接收事件 速度，即流速不匹配

具体表现是：出现当缓存区大小存满（默认缓存区大小 = 128）、被观察者仍然继续发送下1个事件时

处理方式：友好提示：缓存区满了，在onError里提示

onerror = io.reactivex.exceptions.MissingBackpressureException: Queue is full?!

* **BackpressureStrategy.BUFFER**

问题：发送事件速度 ＞ 接收事件 速度，即流速不匹配

具体表现是：出现当缓存区大小存满（默认缓存区大小 = 128）、被观察者仍然续发送下1个事件时

处理方式：将缓存区大小设置成无限大

* 被观察者可无限发送事件 观察者，但实际上是存放在缓存区
* 但要注意内存情况，防止出现OOM
* **BackpressureStrategy.DROP**

处理方式：超过缓存区大小（128）的事件丢弃

如发送了150个事件，被观察者一下子发送150个事件，缓冲区仅保存第1 - 第128个事件，第129 -第150事件将被丢弃

* **BackpressureStrategy.LATEST**

处理方式：只保存最新（最后）事件，超过缓存区大小（128）的事件丢弃

即如果发送了150个事件，缓存区里会保存129个事件（第1到第128个 + 第150事件）

**注意：**

对于自身手动创建FLowable的情况，可通过传入背压模式参数选择背压策略

RxJava 2.0内部提供 封装了背压策略模式的方法   
- onBackpressureBuffer()   
- onBackpressureDrop()   
- onBackpressureLatest()

默认采用BackpressureStrategy.ERROR模式

实例如下：

Flowable.interval(1, TimeUnit.MILLISECONDS)

.onBackpressureBuffer()

// 添加背压策略封装好的方法，此处选择Buffer模式，即缓存区大小无限制

.observeOn(Schedulers.newThread())

.subscribe(new Subscriber<Long>() {

##### 总结：背压完整总结如下图：

