# RxJava 2.x 教程（完结版）

这可能是最好的 RxJava 2.x 入门教程系列专栏
文章链接：
这可能是最好的 RxJava 2.x 入门教程（一）
这可能是最好的 RxJava 2.x 入门教程（二）
这可能是最好的 RxJava 2.x 入门教程（三）
这可能是最好的 RxJava 2.x 入门教程（四）
这可能是最好的 RxJava 2.x 入门教程（五）
GitHub 代码同步更新：https://github.com/nanchen2251/RxJava2Examples
为了满足大家的饥渴难耐，GitHub 将同步更新代码，主要包含基本的代码封装，RxJava 2.x 所有操作符应用场景介绍和实际应用场景，后期除了 RxJava 可能还会增添其他东西，总之，GitHub 上的 Demo 专为大家倾心打造。传送门：
https://github.com/nanchen2251/RxJava2Examples

## 为什么要学 RxJava？

提升开发效率，降低维护成本一直是开发团队永恒不变的宗旨。近两年来国内的技术圈子中越来越多的开始提及 RxJava ，越来越多的应用和面试中都会有 RxJava ，而就目前的情况，Android 的网络库基本被 Retrofit + OkHttp 一统天下了，而配合上**响应式编程** RxJava 可谓如鱼得水。想必大家肯定被近期的 Kotlin 炸开了锅，笔者也在闲暇之时去了解了一番（作为一个与时俱进的有理想的青年怎么可能不与时俱进？），发现其中有个非常好的优点就是简洁，支持函数式编程。是的， RxJava 最大的优点也是简洁，但它不止是简洁，而且是** 随着程序逻辑变得越来越复杂，它依然能够保持简洁 **（这货洁身自好呀有木有）。

咳咳，要例子，猛戳这里：给 Android 开发者的 RxJava 详解

## 什么是响应式编程

上面我们提及了响应式编程，不少新司机对它可谓一脸懵逼，那什么是响应式编程呢？响应式编程是一种基于异步数据流概念的编程模式。数据流就像一条河：它可以被观测，被过滤，被操作，或者为新的消费者与另外一条流合并为一条新的流。

响应式编程的一个关键概念是事件。事件可以被等待，可以触发过程，也可以触发其它事件。事件是唯一的以合适的方式将我们的现实世界映射到我们的软件中：如果屋里太热了我们就打开一扇窗户。同样的，当我们的天气 app 从服务端获取到新的天气数据后，我们需要更新 app 上展示天气信息的 UI；汽车上的车道偏移系统探测到车辆偏移了正常路线就会提醒驾驶者纠正，就是是响应事件。

今天，响应式编程最通用的一个场景是 UI：我们的移动 App 必须做出对网络调用、用户触摸输入和系统弹框的响应。在这个世界上，软件之所以是事件驱动并响应的是因为现实生活也是如此。

# 为什么出了一个系列后还有完结版？

RxJava 这些年可谓越来越流行，而在去年的晚些时候发布了 2.0 正式版。大半年已过，虽然网上已经出现了大部分的 RxJava 教程（其实细心的你还是会发现 1.x 的超级多），前些日子，笔者花了大约两周的闲暇之时写了 RxJava 2.x 系列教程，也得到了不少反馈，其中就有不少读者觉得每一篇的教程太短，抑或是希望更多的侧重适用场景的介绍，在这样的大前提下，这篇完结版教程就此诞生，仅供各位新司机采纳。

# 开始

RxJava 2.x 已经按照 Reactive-Streams specification 规范完全的重写了，maven 也被放在了 `io.reactivex.rxjava2:rxjava:2.x.y` 下，所以 RxJava 2.x 独立于 RxJava 1.x 而存在，而随后官方宣布的将在一段时间后终止对 RxJava 1.x 的维护，所以对于熟悉 RxJava 1.x 的老司机自然可以直接看一下 2.x 的文档和异同就能轻松上手了，而对于不熟悉的年轻司机，不要慌，本酱带你装逼带你飞，马上就发车，坐稳了：
https://github.com/nanchen2251/RxJava2Examples

你只需要在 build.gradle 中加上：`compile 'io.reactivex.rxjava2:rxjava:2.1.1'`（2.1.1 为写此文章时的最新版本）

# 接口变化

RxJava 2.x 拥有了新的特性，其依赖于 4 个基础接口，它们分别是

- Publisher
- Subscriber
- Subscription
- Processor

其中最核心的莫过于 `Publisher` 和 `Subscriber`。`Publisher` 可以发出一系列的事件，
而 `Subscriber` 负责和处理这些事件。

其中用的比较多的自然是 `Publisher` 的 `Flowable`，它支持背压。关于背压给个简洁的定义就是：

背压是指在异步场景中，被观察者发送事件速度远快于观察者的处理速度的情况下，一种告诉上游的被观察者降低发送速度的策略。
简而言之，**背压是流速控制的一种策略**。有兴趣的可以看一下官方对于背压的讲解。

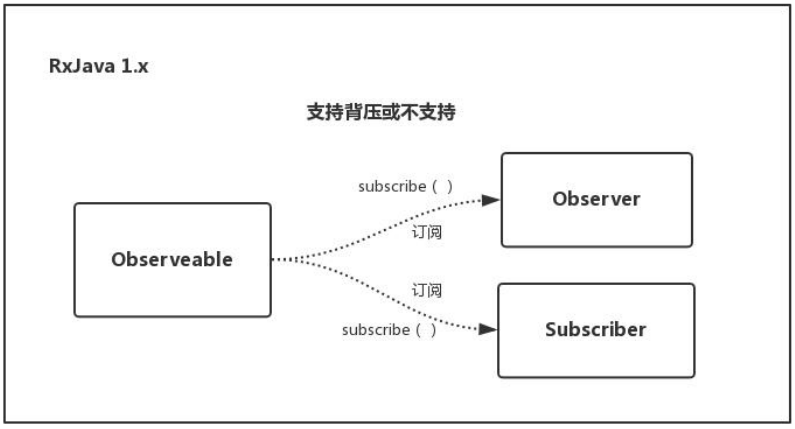
可以明显地发现，RxJava 2.x 最大的改动就是对于 `backpressure` 的处理，为此将原来的 `Observable` 拆分成了新的 `Observable` 和 `Flowable`，同时其他相关部分也同时进行了拆分，但令人庆幸的是，是它，是它，还是它，还是我们最熟悉和最喜欢的 RxJava。

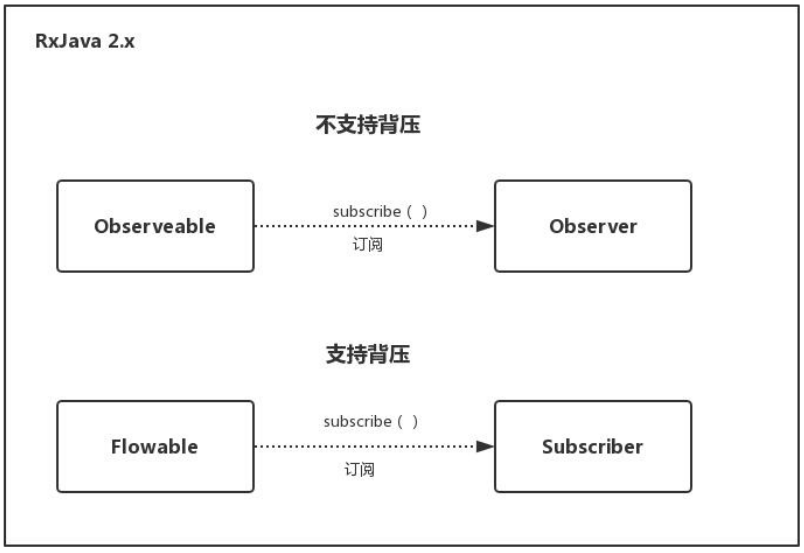
# 观察者模式

大家可能都知道， **RxJava 以观察者模式为骨架，在 2.0 中依旧如此**。

不过此次更新中，出现了两种观察者模式：

- Observable ( 被观察者 ) / Observer ( 观察者 )
- Flowable （被观察者）/ Subscriber （观察者）




在 RxJava 2.x 中，Observable 用于订阅 Observer，不再支持背压（1.x 中可以使用背压策略），而 Flowable 用于订阅 Subscriber ， 是支持背压（Backpressure）的。

# Observable

在 RxJava 1.x 中，我们最熟悉的莫过于 `Observable` 这个类了，笔者在刚刚使用 RxJava 2.x 的时候，创建了 一个 `Observable`，瞬间一脸懵逼有木有，居然连我们最最熟悉的 `Subscriber` 都没

了，取而代之的是 `ObservableEmmiter`，俗称发射器。此外，由于没有了 `Subscriber` 的踪影，我们创建观察者时需使用 `Observer`。而 `Observer` 也不是我们熟悉的那个 `Observer`，又出现了一个 `Disposable` 参数带你装逼带你飞。

废话不多说，从会用开始，还记得 RxJava 的三部曲吗？


** 第一步：初始化 Observable **
** 第二步：初始化 Observer **
** 第三步：建立订阅关系 **

```
Observable.create(new ObservableOnSubscribe<Integer>() { // 第一步：初始化 Observable

            @Override

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

                Log.e(TAG, "Observable emit 1" + "\n");

                e.onNext(1);

                Log.e(TAG, "Observable emit 2" + "\n");

                e.onNext(2);

                Log.e(TAG, "Observable emit 3" + "\n");

                e.onNext(3);

                e.onComplete();

                Log.e(TAG, "Observable emit 4" + "\n" );

                e.onNext(4);

            }

        }).subscribe(new Observer<Integer>() { // 第三步：订阅



            // 第二步：初始化 Observer

            private int i;

            private Disposable mDisposable;
```

```
            @Override
            public void onSubscribe(@NonNull Disposable d) {
                mDisposable = d;
            }

            @Override
            public void onNext(@NonNull Integer integer) {
                i++;
                if (i == 2) {
                    // 在 RxJava 2.x 中，新增的 Disposable 可以做到切断的操作，让 Observer 观察者不再接收上游事件
                    mDisposable.dispose();
                }
            }

            @Override
            public void onError(@NonNull Throwable e) {
                Log.e(TAG, "onError : value : " + e.getMessage() + "\n" );
            }

            @Override
            public void onComplete() {
                Log.e(TAG, "onComplete" + "\n" );
            }
        });
```

不难看出，RxJava 2.x 与 1.x 还是存在着一些区别的。首先，创建 `Observable` 时，回调的是 `ObservableEmitter`，字面意思即发射器，并且直接 throws Exception。其次，在创建的 Observer 中，也多了一个回调方法：`onSubscribe`，传递参数为 `Disposable`，`Disposable` 相当于

RxJava 1.x 中的 `Subscription`，用于解除订阅。可以看到示例代码中，在 i 自增到 2 的时候，订阅关系被切断。

```
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: onSubscribe : false
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 1
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: onNext : value : 1
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 2
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: onNext : value : 2
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: onNext : isDisposable : true
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 3
7-18467/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 4
```

当然，我们的 RxJava 2.x 也为我们保留了简化订阅方法，我们可以根据需求，进行相应的简化订阅，只不过传入对象改为了 `Consumer`。

`Consumer` 即消费者，用于接收单个值，`BiConsumer` 则是接收两个值，`Function` 用于变换对象，`Predicate` 用于判断。这些接口命名大多参照了 Java 8 ，熟悉 Java 8 新特性的应该都知道意思，这里也不再赘述。

# 线程调度

关于线程切换这点，RxJava 1.x 和 RxJava 2.x 的实现思路是一样的。这里简单的说一下，以便于我们的新司机入手。

## subScribeOn

同 RxJava 1.x 一样，`subscribeOn` 用于指定 `subscribe()` 时所发生的线程，从源码角度可以看出，内部线程调度是通过 `ObservableSubscribeOn` 来实现的。

```
@SchedulerSupport(SchedulerSupport.CUSTOM)
public final Observable<T> subscribeOn(Scheduler scheduler) {
    ObjectHelper.requireNonNull(scheduler, "scheduler is null");
    return RxJavaPlugins.onAssembly(new ObservableSubscribeOn<T>(this, scheduler));
}
```

`ObservableSubscribeOn` 的核心源码在 `subscribeActual` 方法中，通过代理的方式使用 `SubscribeOnObserver` 包装 `Observer` 后，设置 `Disposable` 来将 `subscribe` 切换到 `Scheduler` 线程中。

## observeOn

`observeOn` 方法用于指定下游 `Observer` 回调发生的线程。

```
   @SchedulerSupport(SchedulerSupport.CUSTOM)
    public final Observable<T> observeOn(Scheduler scheduler, boolean delayError, int
bufferSize) {
        ObjectHelper.requireNonNull(scheduler, "scheduler is null");
        ObjectHelper.verifyPositive(bufferSize, "bufferSize");
        return RxJavaPlugins.onAssembly(new ObservableObserveOn<T>(this, scheduler, delayError,
bufferSize));
    }
```

## 线程切换需要注意的

RxJava 内置的线程调度器的确可以让我们的线程切换得心应手，但其中也有些需要注意的地方。

- 简单地说，`subscribeOn()` 指定的就是发射事件的线程，`observerOn` 指定的就是订阅者接收事件的线程。
- 多次指定发射事件的线程只有第一次指定的有效，也就是说多次调用 `subscribeOn()`只有第一次的有效，其余的会被忽略。
- 但多次指定订阅者接收线程是可以的，也就是说每调用一次 `observerOn()`，下游的线程就会切换一次。

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {
                Log.e(TAG, "Observable thread is : " + Thread.currentThread().getName());
                e.onNext(1);
                e.onComplete();
            }
        }).subscribeOn(Schedulers.newThread())
                .subscribeOn(Schedulers.io())
```

```
                .observeOn(AndroidSchedulers.mainThread())
                .doOnNext(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        Log.e(TAG, "After observeOn(mainThread), Current thread is " +
Thread.currentThread().getName());
                    }
                })
                .observeOn(Schedulers.io())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        Log.e(TAG, "After observeOn(io), Current thread is " +
Thread.currentThread().getName());
                    }
                });
```

输出：

```
07-03 14:54:01.177 15121-15438/com.nanchen.rxjava2examples E/RxThreadActivity: Observable
thread is : RxNewThreadScheduler-1
07-03 14:54:01.178 15121-15121/com.nanchen.rxjava2examples E/RxThreadActivity: After
observeOn(mainThread), Current thread is main
07-03 14:54:01.179 15121-15439/com.nanchen.rxjava2examples E/RxThreadActivity: After
observeOn(io), Current thread is RxCachedThreadScheduler-2
```

实例代码中，分别用 `Schedulers.newThread()` 和 `Schedulers.io()` 对发射线程进行切换，并采用 `observeOn(AndroidSchedulers.mainThread()` 和 `Schedulers.io()` 进行了接收线程的切换。可以看到输出中发射线程仅仅响应了第一个 `newThread`，但每调用一次 `observeOn()` ，线程便会切换一次，因此如果我们有类似的需求时，便知道如何处理了。

RxJava 中，已经内置了很多线程选项供我们选择，例如有：

- `Schedulers.io()` 代表 io 操作的线程, 通常用于网络,读写文件等 io 密集型的操作；

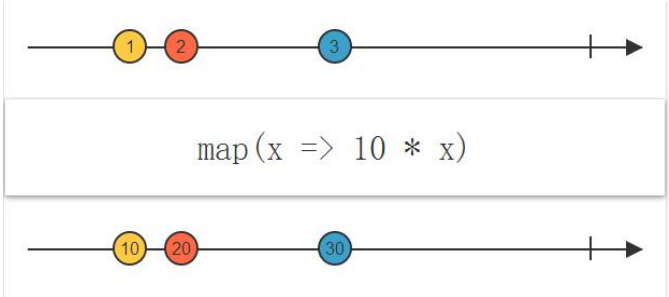
- `Schedulers.computation()` 代表 CPU 计算密集型的操作，例如需要大量计算的操作；
- `Schedulers.newThread()` 代表一个常规的新线程；
- `AndroidSchedulers.mainThread()` 代表 Android 的主线程

这些内置的 `Scheduler` 已经足够满足我们开发的需求，因此我们应该使用内置的这些选项，而 RxJava 内部使用的是线程池来维护这些线程，所以效率也比较高。

# 操作符

关于操作符，在官方文档中已经做了非常完善的讲解，并且笔者前面的系列教程中也着重讲解了绝大多数的操作符作用，这里受于篇幅限制，就不多做赘述，只挑选几个进行实际情景的讲解。

## map




`map` 操作符可以将一个 `Observable` 对象通过某种关系转换为另一个 `Observable` 对象。在 2.x 中和 1.x 中作用几乎一致，不同点在于：2.x 将 1.x 中的 `Func1` 和 `Func2` 改为了 `Function` 和 `BiFunction`。

## 采用 map 操作符进行网络数据解析

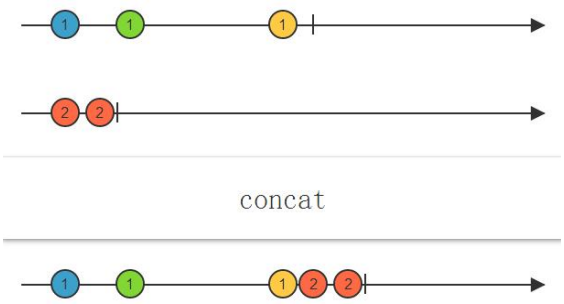
想必大家都知道，很多时候我们在使用 RxJava 的时候总是和 Retrofit 进行结合使用，而为了方便演示，这里我们就暂且采用 OkHttp3 进行演示，配合 `map`，`doOnNext`，线程切换进行简单的网络请求：
1）通过 Observable.create() 方法，调用 OkHttp 网络请求；
2）通过 map 操作符集合 gson，将 Response 转换为 bean 类；
3）通过 doOnNext() 方法，解析 bean 中的数据，并进行数据库存储等操作；
4）调度线程，在子线程中进行耗时操作任务，在主线程中更新 UI ；
5）通过 subscribe()，根据请求成功或者失败来更新 UI 。

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

          @Override

          public void subscribe(@NonNull ObservableEmitter<Response> e) throws Exception {
```

```
                Builder builder = new Builder()
                        .url("http://api.avatardata.cn/MobilePlace/LookUp?key=ec47b85086be4dc
8b5d941f5abd37a4e&mobileNumber=13021671512")
                        .get();
                Request request = builder.build();
                Call call = new OkHttpClient().newCall(request);
                Response response = call.execute();
                e.onNext(response);
            }
        }).map(new Function<Response, MobileAddress>() {
                @Override
                public MobileAddress apply(@NonNull Response response) throws Exception {
                    if (response.isSuccessful()) {
                        ResponseBody body = response.body();
                        if (body != null) {
                            Log.e(TAG, "map:转换前:" + response.body());
                            return new Gson().fromJson(body.string(), MobileAddress.class);
                        }
                    }
                    return null;
                }
            }).observeOn(AndroidSchedulers.mainThread())
            .doOnNext(new Consumer<MobileAddress>() {
                @Override
                public void accept(@NonNull MobileAddress s) throws Exception {
                    Log.e(TAG, "doOnNext: 保存成功：" + s.toString() + "\n");
                }
            }).subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<MobileAddress>() {
```

```
            @Override
            public void accept(@NonNull MobileAddress data) throws Exception {
                Log.e(TAG, "成功:" + data.toString() + "\n");
            }, new Consumer<Throwable>() {
                @Override
                public void accept(@NonNull Throwable throwable) throws Exception {
                    Log.e(TAG, "失败: " + throwable.getMessage() + "\n");
                }
            });
```

## Concat




`concat` 可以做到不交错的发射两个甚至多个 Observable 的发射事件，并且只有前一个 `Observable` 终止(`onComplete`) 后才会订阅下一个 `Observable`。

## 采用 concat 操作符先读取缓存再通过网络请求获取数据

想必在实际应用中，很多时候（对数据操作不敏感时）都需要我们先读取缓存的数据，如果缓存没有数据，再通过网络请求获取，随后在主线程更新我们的 UI。

`concat` 操作符简直就是为我们这种需求量身定做。

利用 `concat` 的必须调用 `onComplete` 后才能订阅下一个 `Observable` 的特性，我们就可以先读取缓存数据，倘若获取到的缓存数据不是我们想要的，再调用 `onComplete()` 以执行获取网络数据的 `Observable`，如果缓存数据能应我们所需，则直接调用 `onNext()`，防止过度的网络请求，浪费用户的流量。

```
Observable<FoodList> cache = Observable.create(new ObservableOnSubscribe<FoodList>() {
            @Override
```

```
        public void subscribe(@NonNull ObservableEmitter<FoodList> e) throws Exception {
            Log.e(TAG, "create 当前线程:"+Thread.currentThread().getName() );
            FoodList data = CacheManager.getInstance().getFoodListData();
            // 在操作符 concat 中，只有调用 onComplete 之后才会执行下一个 Observable
            if (data != null){ // 如果缓存数据不为空，则直接读取缓存数据，而不读取网络数据
                isFromNet = false;
                Log.e(TAG, "\nsubscribe: 读取缓存数据:" );
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        mRxOperatorsText.append("\nsubscribe: 读取缓存数据:\n");
                    }
                });
                e.onNext(data);
            }else {
                isFromNet = true;
                runOnUiThread(new Runnable() {
                    @Override
                    public void run() {
                        mRxOperatorsText.append("\nsubscribe: 读取网络数据:\n");
                    }
                });
                Log.e(TAG, "\nsubscribe: 读取网络数据:" );
                e.onComplete();
            }

        }
    });
```

```
        Observable<FoodList> network =
Rx2AndroidNetworking.get("http://www.tngou.net/api/food/list")
                .addQueryParameter("rows",10+"")
                .build()
                .getObjectObservable(FoodList.class);
        // 两个 Observable 的泛型应当保持一致
        Observable.concat(cache,network)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<FoodList>() {
                    @Override
                    public void accept(@NonNull FoodList tngouBeen) throws Exception {
                        Log.e(TAG, "subscribe 成功:"+Thread.currentThread().getName() );
                        if (isFromNet){
                            mRxOperatorsText.append("accept : 网络获取数据设置缓存: \n");
                            Log.e(TAG, "accept : 网络获取数据设置缓存:
\n"+tngouBeen.toString() );
                            CacheManager.getInstance().setFoodListData(tngouBeen);
                        }

                        mRxOperatorsText.append("accept: 读取数据成功:" +
tngouBeen.toString()+"\n");
                        Log.e(TAG, "accept: 读取数据成功:" + tngouBeen.toString());
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(@NonNull Throwable throwable) throws Exception {
                        Log.e(TAG, "subscribe 失败:"+Thread.currentThread().getName() );
                        Log.e(TAG, "accept: 读取数据失败："+throwable.getMessage() );
                        mRxOperatorsText.append("accept: 读取数据失败：
"+throwable.getMessage()+"\n");
```

```
                }
            });
```

有时候我们的缓存可能还会分为 memory 和 disk ，实际上都差不多，无非是多写点 `Observable` ，然后通过 `concat` 合并即可。

## flatMap 实现多个网络请求依次依赖

想必这种情况也在实际情况中比比皆是，例如用户注册成功后需要自动登录，我们只需要先通过注册接口注册用户信息，注册成功后马上调用登录接口进行自动登录即可。

我们的 `flatMap` 恰好解决了这种应用场景，`flatMap` 操作符可以将一个发射数据的 `Observable` 变换为多个 `Observables` ，然后将它们发射的数据合并后放到一个单独的 `Observable`，利用这个特性，我们很轻松地达到了我们的需求。

```
Rx2AndroidNetworking.get("http://www.tngou.net/api/food/list")
                .addQueryParameter("rows", 1 + "")
                .build()
                .getObjectObservable(FoodList.class) // 发起获取食品列表的请求，并解析到 FootList
                .subscribeOn(Schedulers.io())        // 在 io 线程进行网络请求
                .observeOn(AndroidSchedulers.mainThread()) // 在主线程处理获取食品列表的请求结果
                .doOnNext(new Consumer<FoodList>() {
                    @Override
                    public void accept(@NonNull FoodList foodList) throws Exception {
                        // 先根据获取食品列表的响应结果做一些操作
                        Log.e(TAG, "accept: doOnNext :" + foodList.toString());
                        mRxOperatorsText.append("accept: doOnNext :" +
foodList.toString()+"\n");
                    }
                })
                .observeOn(Schedulers.io()) // 回到 io 线程去处理获取食品详情的请求
                .flatMap(new Function<FoodList, ObservableSource<FoodDetail>>() {
                    @Override
```

```
                public ObservableSource<FoodDetail> apply(@NonNull FoodList foodList)
throws Exception {
                    if (foodList != null && foodList.getTngou() != null &&
foodList.getTngou().size() > 0) {
                        return
Rx2AndroidNetworking.post("http://www.tngou.net/api/food/show")
                                .addBodyParameter("id", foodList.getTngou().get(0).getId()
+ "")
                                .build()
                                .getObjectObservable(FoodDetail.class);
                    }
                    return null;
                }
            })
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<FoodDetail>() {
                @Override
                public void accept(@NonNull FoodDetail foodDetail) throws Exception {
                    Log.e(TAG, "accept: success : " + foodDetail.toString());
                    mRxOperatorsText.append("accept: success : " +
foodDetail.toString()+"\n");
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(@NonNull Throwable throwable) throws Exception {
                    Log.e(TAG, "accept: error :" + throwable.getMessage());
                    mRxOperatorsText.append("accept: error :" +
throwable.getMessage()+"\n");
                }
            });
```

## 善用 zip 操作符，实现多个接口数据共同更新 UI

在实际应用中，我们极有可能会在一个页面显示的数据来源于多个接口，这时候我们的 `zip` 操作符为我们排忧解难。

`zip` 操作符可以将多个 `Observable` 的数据结合为一个数据源再发射出去。

```
Observable<MobileAddress> observable1 =
Rx2AndroidNetworking.get("http://api.avatardata.cn/MobilePlace/LookUp?key=ec47b85086be4dc8b5d941f5abd37a4e&mobileNumber=13021671512")
                .build()
                .getObjectObservable(MobileAddress.class);
        Observable<CategoryResult> observable2 = Network.getGankApi()
                .getCategoryData("Android",1,1);
        Observable.zip(observable1, observable2, new BiFunction<MobileAddress, CategoryResult, String>() {
            @Override
            public String apply(@NonNull MobileAddress mobileAddress, @NonNull CategoryResult categoryResult) throws Exception {
                return "合并后的数据为：手机归属地："+mobileAddress.getResult().getMobilearea()+"人名："+categoryResult.results.get(0).who;
            }
        }).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        Log.e(TAG, "accept: 成功：" + s+"\n");
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(@NonNull Throwable throwable) throws Exception {
                        Log.e(TAG, "accept: 失败：" + throwable+"\n");
                    }
```

```
            });
```

## 采用 interval 操作符实现心跳间隔任务

想必即时通讯等需要轮训的任务在如今的 APP 中已是很常见，而 RxJava 2.x 的 `interval` 操作符可谓完美地解决了我们的疑惑。

这里就简单的意思一下轮训。

```
private Disposable mDisposable;

    @Override

    protected void doSomething() {

        mDisposable = Flowable.interval(1, TimeUnit.SECONDS)

                .doOnNext(new Consumer<Long>() {

                    @Override

                    public void accept(@NonNull Long aLong) throws Exception {

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

                    }

                })

                .observeOn(AndroidSchedulers.mainThread())

                .subscribe(new Consumer<Long>() {

                    @Override

                    public void accept(@NonNull Long aLong) throws Exception {

                        Log.e(TAG, "accept: 设置文本 : "+aLong );

                        mRxOperatorsText.append("accept: 设置文本 : "+aLong +"\n");

                    }

                });

    }


    /**

     * 销毁时停止心跳
```

```
     */
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mDisposable != null){
            mDisposable.dispose();
        }
    }
```

## RxJava 1.x 如何平滑升级到 RxJava 2.x？

由于 RxJava 2.x 变化较大无法直接升级，幸运的是，官方为我们提供了 RxJava2Interrop 这个库，可以方便地把 RxJava 1.x 升级到 RxJava 2.x，或者将 RxJava 2.x 转回到 RxJava 1.x。

# RxJava 2.x 入门教程（一）

## 前言

RxJava 对大家而言肯定不陌生，其受欢迎程度不言而喻。而在去年的早些时候，官方便宣布，将在一段时间后不再对 RxJava 1.x 进行维护，而在仓库中另辟蹊径，开始对 RxJava 2.x 进行推广起来，我原本是不想写这么一套教程的，因为 RxJava 受欢迎度这么高，而且这 2.x 也出来了这么久，我坚信网上一定有很多超级大牛早已为大家避雷。然而很难过的是，我搜索了些时间，能搜出来的基本都是对 RxJava 1.x 的讲解，或者是 Blog 标题就没说清楚是否是 2.x 系列（对于我们这种标题党来说很难受）。这不，我就来抛砖引玉了。

咱们先不提别的，先为大家带点可能你早已熟知的干货——来自扔物线大神的给 Android 开发者的 RxJava 详解。

该文详细地为大家讲解了 RxJava 的优势、原理以及使用方式和适用情景，一定被众多的 Android 开发者视为神器。可惜，文章历史比较久远，基本都是讲解的 RxJava 1.x 了。

那关注的小伙伴一定会问，那我没用过 RxJava 1.x ，还有必要先学习 1.x 的内容吗？

个人觉得不必要，因为 RxJava 2.x 是按照 **Reactive-Streams specification** 规范完全的重写的，完全独立于 RxJava 1.x 而存在，它改变了以往 RxJava 的用法。

额，由于个人能力水平有限，所以对于英文基础好的，大家可以去官网查阅相关 API 介绍，而对于英文不那么流畅的童鞋，我也为大家准备了干货：RxJava2Examples (正在更新）。

# 与 RxJava 1.x 的差异

其实，我标题为入门教程，按理说应该从简单入门开始讲的，原谅我突然偏题了，因为我觉得可能大多数人都了解或者使用过 RxJava 1.x（因为它真的太棒了）。虽然可能熟悉 1.x 的你可以直接扒文档就可以了，但这么大的变化，请原谅我还在这里瞎比比。

### Nulls

这是一个很大的变化，熟悉 RxJava 1.x 的童鞋一定都知道，1.x 是允许我们在发射事件的时候传入 null 值的，但现在我们的 2.x 不支持了，不信你试试？ 大大的 `NullPointerException` 教你做人。这意味着 `Observable<Void>` 不再发射任何值，而是正常结束或者抛出空指针。

### Flowable

在 RxJava 1.x 中关于介绍 `backpressure` 部分有一个小小的遗憾，那就是没有用一个单独的类，而是使用 `Observable` 。而在 2.x 中 `Observable` 不支持背压了，将用一个全新的 Flowable 来支持背压。

或许对于背压，有些小伙伴们还不是特别理解，这里简单说一下。大概就是指在异步场景中，被观察者发送事件的速度远快于观察者的处理速度的情况下，一种告诉上游的被观察者降低发送速度的策略。感兴趣的小伙伴可以模拟这种情况，在差距太大的时候，我们的内存会猛增，直到 OOM。而我们的 Flowable 一定意义上可以解决这样的问题，但其实并不能完全解决，这个后面可能会提到。

### Single/Completable/Maybe

其实这三者都差不多，`Single` 顾名思义，只能发送一个事件，和 `Observable` 接受可变参数完全不同。而 `Completable` 侧重于观察结果，而 `Maybe` 是上面两种的结合体。也就是说，当你只想要某个事件的结果（true or false）的时候，你可以使用这种观察者模式。

### 线程调度相关

这一块基本没什么改动，但细心的小伙伴一定会发现，RxJava 2.x 中已经没有了 `Schedulers.immediate()` 这个线程环境，还有 `Schedulers.test()`。

### Function 相关

熟悉 1.x 的小伙伴一定都知道，我们在 1.x 中是有 `Func1`，`Func2`.....`FuncN` 的，但 2.x 中将它们

移除，而采用 `Function` 替换了 `Func1`，采用 `BiFunction` 替换了 `Func 2..N`。并且，它们都增加了 throws Exception，也就是说，妈妈再也不用担心我们做某些操作还需要 try-catch 了。

**其他操作符相关**

如 `Func1...N` 的变化，现在同样用 `Consumer` 和 `BiConsumer` 对 `Action1` 和 `Action2` 进行了替换。后面的 `Action` 都被替换了，只保留了 `ActionN`。

# 附录

下面从官方截图展示 2.x 相对 1.x 的改动细节，仅供参考。

## 1.x Observable to 2.x Flowable

### Factory methods:

| 1.x | 2.x |
|---|---|
| `amb` | added `amb(ObservableSource...)` overload, 2-9 argument versions dropped |
| RxRingBuffer.SIZE | `bufferSize()` |
| `combineLatest` | added varargs overload, added overloads with `bufferSize` argument, `combineLatest(List)` dropped |
| `concat` | added overload with `prefetch` argument, 5-9 source overloads dropped, use `concatArray` instead |
| N/A | added `concatArray` and `concatArrayDelayError` |
| N/A | added `concatArrayEager` and `concatArrayEagerDelayError` |
| `concatDelayError` | added overloads with option to delay till the current ends or till the very end |
| `concatEagerDelayError` | added overloads with option to delay till the current ends or till the very end |
| `create(SyncOnSubscribe)` | replaced with `generate` + overloads (distinct interfaces, you can implement them all at once) |
| `create(AsnycOnSubscribe)` | not present |
| `create(OnSubscribe)` | repurposed with safe `create(FlowableOnSubscribe, BackpressureStrategy)`, raw support via `unsafeCreate()` |
| `from` | disambiguated into `fromArray`, `fromIterable`, `fromFuture` |
| N/A | added `fromPublisher` |
| `fromAsync` | renamed to `create()` |
| N/A | added `intervalRange()` |

| | |
|---|---|
| `limit` | dropped, use `take` |
| `merge` | added overloads with `prefetch` |
| `mergeDelayError` | added overloads with `prefetch` |
| `sequenceEqual` | added overload with `bufferSize` |
| `switchOnNext` | added overload with `prefetch` |
| `switchOnNextDelayError` | added overload with `prefetch` |
| `timer` | deprecated overloads dropped |
| `zip` | added overloads with `bufferSize` and `delayErrors` capabilities, disambiguated to `zipArray` and `zipIterable` |

### Instance methods:

| 1.x | 2.x |
|---|---|
| `all` | RC3 returns `Single<Boolean>` now |
| `any` | RC3 returns `Single<Boolean>` now |
| `asObservable` | renamed to `hide()`, hides all identities now |
| `buffer` | overloads with custom `Collection` supplier |
| `cache(int)` | deprecated and dropped |
| `collect` | RC3 returns `Single<U>` |
| `collect(U, Action2<U, T>)` | disambiguated to `collectInto` and RC3 returns `Single<U>` |
| `concatMap` | added overloads with `prefetch` |
| `concatMapDelayError` | added overloads with `prefetch`, option to delay till the current ends or till the very end |
| `concatMapEager` | added overloads with `prefetch` |
| `concatMapEagerDelayError` | added overloads with `prefetch`, option to delay till the current ends or till the very end |
| `count` | RC3 returns `Single<Long>` now |
| `countLong` | dropped, use `count` |
| `distinct` | overload with custom `Collection` supplier. |
| `doOnCompleted` | renamed to `doOnComplete`, note the missing `d`! |
| `doOnUnsubscribe` | renamed to `Flowable.doOnCancel` and `doOnDispose` for the others, additional info |

| | |
|---|---|
| N/A | added `doOnLifecylce` to handle `onSubscribe`, `request` and `cancel` peeking |
| `elementAt(int)` | **RC3** no longer signals NoSuchElementException if the source is shorter than the index |
| `elementAt(Func1, int)` | dropped, use `filter(predicate).elementAt(int)` |
| `elementAtOrDefault(int, T)` | renamed to `elementAt(int, T)` and **RC3** returns `Single<T>` |
| `elementAtOrDefault(Func1, int, T)` | dropped, use `filter(predicate).elementAt(int, T)` |
| `first()` | **RC3** renamed to `firstElement` and returns `Maybe<T>` |
| `first(Func1)` | dropped, use `filter(predicate).first()` |
| `firstOrDefault(T)` | renamed to `first(T)` and **RC3** returns `Single<T>` |
| `firstOrDefault(Func1, T)` | dropped, use `filter(predicate).first(T)` |
| `flatMap` | added overloads with `prefetch` |
| N/A | added `forEachWhile(Predicate<T>, [Consumer<Throwable>, [Action]])` for conditionally stopping consumption |
| `groupBy` | added overload with `bufferSize` and `delayError` option, *the custom internal map version didn't make it into RC1* |
| `ignoreElements` | **RC3** returns `Completable` |
| `isEmpty` | **RC3** returns `Single<Boolean>` |
| `last()` | **RC3** renamed to `lastElement` and returns `Maybe<T>` |
| `last(Func1)` | dropped, use `filter(predicate).last()` |
| `lastOrDefault(T)` | renamed to `last(T)` and **RC3** returns `Single<T>` |
| `lastOrDefault(Func1, T)` | dropped, use `filter(predicate).last(T)` |

| | |
|---|---|
| `nest` | dropped, use manual `just` |
| `publish(Func1)` | added overload with `prefetch` |
| `reduce(Func2)` | **RC3** returns `Maybe<T>` |
| N/A | added `reduceWith(Callable, BiFunction)` to reduce in a Subscriber-individual manner, returns `Single<T>` |
| N/A | added `repeatUntil(BooleanSupplier)` |
| `repeatWhen(Func1, Scheduler)` | dropped the overload, use `subscribeOn(Scheduler).repeatWhen(Function)` instead |
| `retry` | added `retry(Predicate)`, `retry(int, Predicate)` |
| N/A | added `retryUntil(BooleanSupplier)` |
| `retryWhen(Func1, Scheduler)` | dropped the overload, use `subscribeOn(Scheduler).retryWhen(Function)` instead |
| N/A | added `sampleWith(Callable, BiFunction)` to scan in a Subscriber-individual manner |
| `single()` | **RC3** renamed to `singleElement` and returns `Maybe<T>` |
| `single(Func1)` | dropped, use `filter(predicate).single()` |
| `singleOrDefault(T)` | renamed to `single(T)` and **RC3** returns `Single<T>` |
| `singleOrDefault(Func1, T)` | dropped, use `filter(predicate).single(T)` |
| `skipLast` | added overloads with `bufferSize` and `delayError` options |
| `startWith` | 2-9 argument version dropped, use `startWithArray` instead |
| N/A | added `startWithArray` to disambiguate |
| N/A | added `subscribeWith` that returns its input after subscription |
| `switchMap` | added overload with `prefetch` argument |
| `switchMapDelayError` | added overload with `prefetch` argument |

| | |
|---|---|
| `takeLastBuffer` | dropped |
| N/A | added `test()` (returns TestSubscriber subscribed to this) with overloads to fluently test |
| `timeout(Func0<Observable>, ...)` | signature changed to `timeout(Publisher, ...)` and dropped the function, use `defer(Callable<Publisher>>)` if necessary |
| `toBlocking().y` | inlined as `blockingY()` operators, except `toFuture` |
| `toCompletable` | RC3 dropped, use `ignoreElements` |
| `toList` | RC3 returns `Single<List<T>>` |
| `toMap` | RC3 returns `Single<Map<K, V>>` |
| `toMultimap` | RC3 returns `Single<Map<K, Collection<V>>>` |
| N/A | added `toFuture` |
| N/A | added `toObservable` |
| `toSingle` | RC3 dropped, use `single(T)` |
| `toSortedList` | RC3 returns `Single<List<T>>` |
| `withLatestFrom` | 5-9 source overloads dropped |
| `zipWith` | added overloads with `prefetch` and `delayErrors` options |

| Operator | Old return type | New return type | Remark |
|---|---|---|---|
| `all(Predicate)` | `Observable<Boolean>` | `Single<Boolean>` | Emits true if all elements match the predicate |
| `any(Predicate)` | `Observable<Boolean>` | `Single<Boolean>` | Emits true if any elements match the predicate |
| `count()` | `Observable<Long>` | `Single<Long>` | Counts the number of elements in the sequence |
| `elementAt(int)` | `Observable<T>` | `Maybe<T>` | Emits the element at the given index or completes |
| `elementAt(int, T)` | `Observable<T>` | `Single<T>` | Emits the element at the given index or the default |
| `first(T)` | `Observable<T>` | `Single<T>` | Emits the very first element or `NoSuchElementException` |
| `firstElement()` | `Observable<T>` | `Maybe<T>` | Emits the very first element or completes |
| `ignoreElements()` | `Observable<T>` | `Completable` | Ignore all but the terminal events |
| `isEmpty()` | `Observable<Boolean>` | `Single<Boolean>` | Emits true if the source is empty |
| `last(T)` | `Observable<T>` | `Single<T>` | Emits the very last element or the default item |

| | | | |
|---|---|---|---|
| `lastElement()` | `Observable<T>` | `Maybe<T>` | Emits the very last element or completes |
| `reduce(BiFunction)` | `Observable<T>` | `Maybe<T>` | Emits the reduced value or completes |
| `reduce(Callable, BiFunction)` | `Observable<U>` | `Single<U>` | Emits the reduced value (or the initial value) |
| `reduceWith(U, BiFunction)` | `Observable<U>` | `Single<U>` | Emits the reduced value (or the initial value) |
| `single(T)` | `Observable<T>` | `Single<T>` | Emits the only element or the default item |
| `singleElement()` | `Observable<T>` | `Maybe<T>` | Emits the only element or completes |
| `toList()` | `Observable<List<T>>` | `Single<List<T>>` | collects all elements into a `List` |
| `toMap()` | `Observable<Map<K, V>>` | `Single<Map<K, V>>` | collects all elements into a `Map` |
| `toMultimap()` | `Observable<Map<K, Collection<V>>>` | `Single<Map<K, Collection<V>>>` | collects all elements into a `Map` with collection |
| `toSortedList()` | `Observable<List<T>>` | `Single<List<T>>` | collects all elements into a `List` and sorts it |

# RxJava 2.x 入门教程（二）

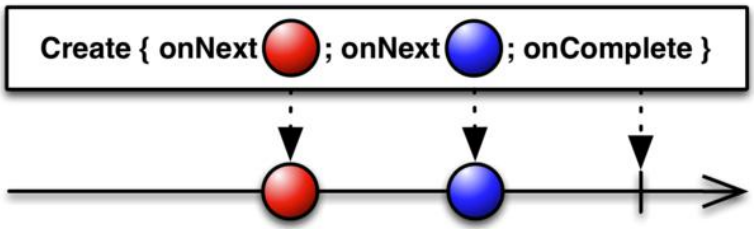
## 前言

很快我们就迎来了第二期，上一期我们主要讲解了 RxJava 1.x 到 2.x 的变化概览，相信各位熟练掌握 RxJava 1.x 的老司机们随便看一下变化概览就可以上手 RxJava 2.x 了，但为了满足更广大的年轻一代司机（未来也是老司机），在本节中，我们将学习 RxJava 2.x 强大的操作符章节。
【注】以下所有操作符标题都可直接点击进入官方 doc 查看。

## 正题

### Create

`create` 操作符应该是最常见的操作符了，主要用于产生一个 `Obserable` 被观察者对象，为了方便大家的认知，以后的教程中统一把被观察者 `Observable` 称为发射器（上游事件），观察者 `Observer` 称为接收器（下游事件）。




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

            @Override

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

                mRxOperatorsText.append("Observable emit 1" + "\n");

                Log.e(TAG, "Observable emit 1" + "\n");

                e.onNext(1);

                mRxOperatorsText.append("Observable emit 2" + "\n");

                Log.e(TAG, "Observable emit 2" + "\n");

                e.onNext(2);

                mRxOperatorsText.append("Observable emit 3" + "\n");

                Log.e(TAG, "Observable emit 3" + "\n");

                e.onNext(3);

                e.onComplete();

                mRxOperatorsText.append("Observable emit 4" + "\n");

                Log.e(TAG, "Observable emit 4" + "\n" );

                e.onNext(4);

            }

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

            private int i;

            private Disposable mDisposable;


            @Override

            public void onSubscribe(@NonNull Disposable d) {

                mRxOperatorsText.append("onSubscribe : " + d.isDisposed() + "\n");

                Log.e(TAG, "onSubscribe : " + d.isDisposed() + "\n" );

                mDisposable = d;
```

```
            }

            @Override
            public void onNext(@NonNull Integer integer) {
                mRxOperatorsText.append("onNext : value : " + integer + "\n");
                Log.e(TAG, "onNext : value : " + integer + "\n" );
                i++;
                if (i == 2) {
                    // 在 RxJava 2.x 中，新增的 Disposable 可以做到切断的操作，让 Observer 观察者不再接收上游事件
                    mDisposable.dispose();
                    mRxOperatorsText.append("onNext : isDisposable : " + mDisposable.isDisposed() + "\n");
                    Log.e(TAG, "onNext : isDisposable : " + mDisposable.isDisposed() + "\n");
                }
            }

            @Override
            public void onError(@NonNull Throwable e) {
                mRxOperatorsText.append("onError : value : " + e.getMessage() + "\n");
                Log.e(TAG, "onError : value : " + e.getMessage() + "\n" );
            }

            @Override
            public void onComplete() {
                mRxOperatorsText.append("onComplete" + "\n");
                Log.e(TAG, "onComplete" + "\n" );
            }
        });
```

输出：

```
06-22 16:14:18.052 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: onSubscribe : false
06-22 16:14:18.053 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 1
06-22 16:14:18.053 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: onNext : value : 1
06-22 16:14:18.053 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 2
06-22 16:14:18.054 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: onNext : value : 2
06-22 16:14:18.054 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: onNext : isDisposable : true
06-22 16:14:18.054 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 3
06-22 16:14:18.055 21489-21489/com.nanchen.rxjava2examples E/RxCreateActivity: Observable emit 4
```

需要注意的几点是：

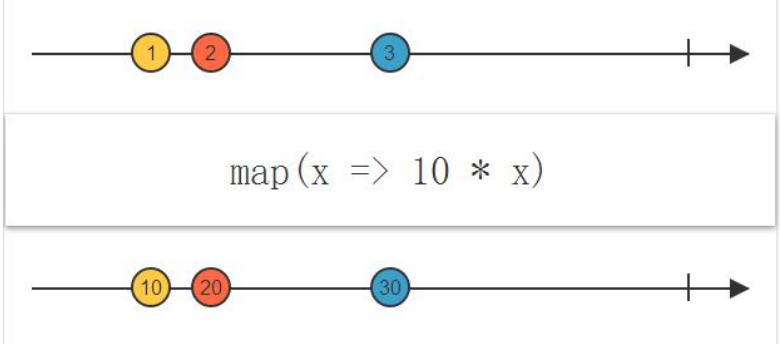
- 在发射事件中，我们在发射了数值 3 之后，直接调用了 `e.onComlete()`，虽然无法接收事件，但发送事件还是继续的。

- 另外一个值得注意的点是，在 RxJava 2.x 中，可以看到发射事件方法相比 1.x 多了一个 throws Excetion，意味着我们做一些特定操作再也不用 try-catch 了。

并且 2.x 中有一个 `Disposable` 概念，这个东西可以直接调用切断，可以看到，当它的 `isDisposed()` 返回为 false 的时候，接收器能正常接收事件，但当其为 true 的时候，接收器停止了接收。所以可以通过此参数动态控制接收事件了。

## Map

`Map` 基本算是 RxJava 中一个最简单的操作符了，熟悉 RxJava 1.x 的知道，它的作用是对发射时间发送的每一个事件应用一个函数，是的每一个事件都按照指定的函数去变化，而在 2.x 中它的作用几乎一致。




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

```
                e.onNext(2);

                e.onNext(3);

            }

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

            @Override

            public String apply(@NonNull Integer integer) throws Exception {

                return "This is result " + integer;

            }

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

            @Override

            public void accept(@NonNull String s) throws Exception {

                mRxOperatorsText.append("accept : " + s +"\n");

                Log.e(TAG, "accept : " + s +"\n" );

            }

        });
```

输出：

```
06-22 16:26:58.597 21489-21489/com.nanchen.rxjava2examples E/RxMapActivity: accept : This is result 1
06-22 16:26:58.598 21489-21489/com.nanchen.rxjava2examples E/RxMapActivity: accept : This is result 2
06-22 16:26:58.600 21489-21489/com.nanchen.rxjava2examples E/RxMapActivity: accept : This is result 3
```

是的，`map` 基本作用就是将一个 `Observable` 通过某种函数关系，转换为另一种 `Observable`，上面例子中就是把我们的 `Integer` 数据变成了 `String` 类型。从 Log 日志显而易见。

## Zip

`zip` 专用于合并事件，该合并不是连接（连接操作符后面会说），而是两两配对，也就意味着，最终配对出的 `Observable` 发射事件数目只和少的那个相同。

```
Observable.zip(getStringObservable(), getIntegerObservable(), new BiFunction<String, Integer,
String>() {

            @Override

            public String apply(@NonNull String s, @NonNull Integer integer) throws Exception
{
```

```
                return s + integer;
            }
        }).subscribe(new Consumer<String>() {
            @Override
            public void accept(@NonNull String s) throws Exception {
                mRxOperatorsText.append("zip : accept : " + s + "\n");
                Log.e(TAG, "zip : accept : " + s + "\n");
            }
        });
private Observable<String> getStringObservable() {
        return Observable.create(new ObservableOnSubscribe<String>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<String> e) throws Exception {
                if (!e.isDisposed()) {
                    e.onNext("A");
                    mRxOperatorsText.append("String emit : A \n");
                    Log.e(TAG, "String emit : A \n");
                    e.onNext("B");
                    mRxOperatorsText.append("String emit : B \n");
                    Log.e(TAG, "String emit : B \n");
                    e.onNext("C");
                    mRxOperatorsText.append("String emit : C \n");
                    Log.e(TAG, "String emit : C \n");
                }
            }
        });
    }

    private Observable<Integer> getIntegerObservable() {
```

```
        return Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {
                if (!e.isDisposed()) {
                    e.onNext(1);
                    mRxOperatorsText.append("Integer emit : 1 \n");
                    Log.e(TAG, "Integer emit : 1 \n");
                    e.onNext(2);
                    mRxOperatorsText.append("Integer emit : 2 \n");
                    Log.e(TAG, "Integer emit : 2 \n");
                    e.onNext(3);
                    mRxOperatorsText.append("Integer emit : 3 \n");
                    Log.e(TAG, "Integer emit : 3 \n");
                    e.onNext(4);
                    mRxOperatorsText.append("Integer emit : 4 \n");
                    Log.e(TAG, "Integer emit : 4 \n");
                    e.onNext(5);
                    mRxOperatorsText.append("Integer emit : 5 \n");
                    Log.e(TAG, "Integer emit : 5 \n");
                }
            }
        });
    }
```

输出：

```
06-22 16:41:42.607 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: String emit : A
06-22 16:41:42.608 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: String emit : B
06-22 16:41:42.609 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: String emit : C
06-22 16:41:42.610 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: zip : accept : A1
06-22 16:41:42.611 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: Integer emit : 1
06-22 16:41:42.612 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: zip : accept : B2
06-22 16:41:42.613 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: Integer emit : 2
06-22 16:41:42.614 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: zip : accept : C3
06-22 16:41:42.615 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: Integer emit : 3
06-22 16:41:42.615 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: Integer emit : 4
06-22 16:41:42.616 21489-21489/com.nanchen.rxjava2examples E/RxZipActivity: Integer emit : 5
```

需要注意的是：

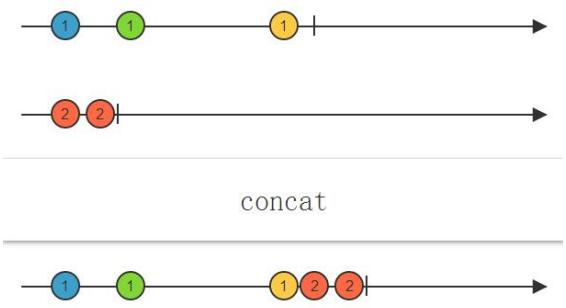
`zip` 组合事件的过程就是分别从发射器 A 和发射器 B 各取出一个事件来组合，并且一个事件只能被使用一次，组合的顺序是严格按照事件发送的顺序来进行的，所以上面截图中，可以看到，1 永远是和 A 结合的，2 永远是和 B 结合的。

最终接收器收到的事件数量是和发送器发送事件最少的那个发送器的发送事件数目相同，所以如截图中，5 很孤单，没有人愿意和它交往，孤独终老的单身狗。

- 

## Concat

对于单一的把两个发射器连接成一个发射器，虽然 `zip` 不能完成，但我们还是可以自力更生，官方提供的 `concat` 让我们的问题得到了完美解决。




```
Observable.concat(Observable.just(1,2,3), Observable.just(4,5,6))
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
```

```
                mRxOperatorsText.append("concat : "+ integer + "\n");

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

            }

        });
```

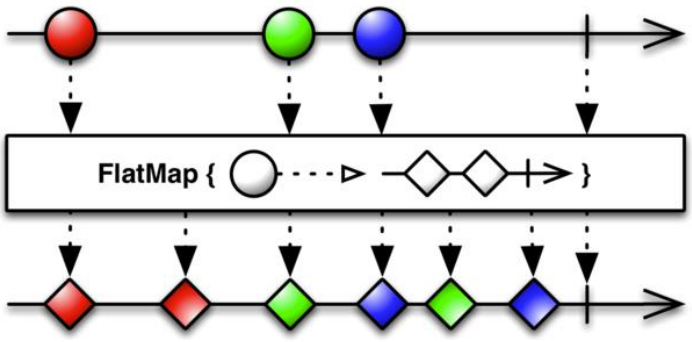
输出：

```
06-22 16:50:29.788 21489-21489/com.nanchen.rxjava2examples E/RxConcatActivity: concat : 1
06-22 16:50:29.789 21489-21489/com.nanchen.rxjava2examples E/RxConcatActivity: concat : 2
06-22 16:50:29.790 21489-21489/com.nanchen.rxjava2examples E/RxConcatActivity: concat : 3
06-22 16:50:29.791 21489-21489/com.nanchen.rxjava2examples E/RxConcatActivity: concat : 4
06-22 16:50:29.792 21489-21489/com.nanchen.rxjava2examples E/RxConcatActivity: concat : 5
06-22 16:50:29.794 21489-21489/com.nanchen.rxjava2examples E/RxConcatActivity: concat : 6
```

如图，可以看到。发射器 B 把自己的三个孩子送给了发射器 A，让他们组合成了一个新的发射器，非常懂事的孩子，有条不紊的排序接收。

## FlatMap

`FlatMap` 是一个很有趣的东西，我坚信你在实际开发中会经常用到。它可以把一个发射器 `Observable` 通过某种方法转换为多个 `Observables`，然后再把这些分散的 `Observables` 装进一个单一的发射器 `Observable`。但有个需要注意的是，`flatMap` 并不能保证事件的顺序，如果需要保证，需要用到我们下面要讲的 `ConcatMap`。




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

        @Override

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

            e.onNext(1);

            e.onNext(2);

            e.onNext(3);

        }
```

```
        }).flatMap(new Function<Integer, ObservableSource<String>>() {
            @Override
            public ObservableSource<String> apply(@NonNull Integer integer) throws Exception {
                List<String> list = new ArrayList<>();
                for (int i = 0; i < 3; i++) {
                    list.add("I am value " + integer);
                }
                int delayTime = (int) (1 + Math.random() * 10);
                return Observable.fromIterable(list).delay(delayTime, TimeUnit.MILLISECONDS);
            }
        }).subscribeOn(Schedulers.newThread())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        Log.e(TAG, "flatMap : accept : " + s + "\n");
                        mRxOperatorsText.append("flatMap : accept : " + s + "\n");
                    }
                });
```

输出：

```
06-22 16:53:54.050 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 2
06-22 16:53:54.051 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 3
06-22 16:53:54.052 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 3
06-22 16:53:54.053 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 3
06-22 16:53:54.054 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 2
06-22 16:53:54.054 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 2
06-22 16:53:54.055 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 1
06-22 16:53:54.056 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 1
06-22 16:53:54.056 21489-21489/com.nanchen.rxjava2examples E/RxFlatMapActivity: flatMap : accept : I am value 1
```

一切都如我们预期中的有意思，为了区分 `concatMap`（下一个会讲），我在代码中特意动了一点小手脚，我采用一个随机数，生成一个时间，然后通过 `delay`（后面会讲）操作符，做一个小延时操作，而查看 Log 日志也确认验证了我们上面的说法，它是无序的。

## concatMap

上面其实就说了，`concatMap` 与 `FlatMap` 的唯一区别就是 `concatMap` 保证了顺序，所以，我们就直接把 `flatMap` 替换为 `concatMap` 验证吧。

```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Integer> e) throws Exception {
                e.onNext(1);
                e.onNext(2);
                e.onNext(3);
            }
        }).concatMap(new Function<Integer, ObservableSource<String>>() {
            @Override
            public ObservableSource<String> apply(@NonNull Integer integer) throws Exception {
                List<String> list = new ArrayList<>();
                for (int i = 0; i < 3; i++) {
                    list.add("I am value " + integer);
                }
                int delayTime = (int) (1 + Math.random() * 10);
                return Observable.fromIterable(list).delay(delayTime, TimeUnit.MILLISECONDS);
            }
        }).subscribeOn(Schedulers.newThread())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        Log.e(TAG, "flatMap : accept : " + s + "\n");
                        mRxOperatorsText.append("flatMap : accept : " + s + "\n");
                    }
                });
```

输出：

```
06-22 17:07:03.089 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 1
06-22 17:07:03.090 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 1
06-22 17:07:03.090 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 1
06-22 17:07:03.101 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 2
06-22 17:07:03.102 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 2
06-22 17:07:03.103 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 2
06-22 17:07:03.103 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 3
06-22 17:07:03.105 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 3
06-22 17:07:03.106 20049-20049/com.nanchen.rxjava2examples E/RxConcatMapActivity: concatMap : accept : I am value 3
```

结果的确和我们预想的一样。

# RxJava 2.x 入门教程（三）

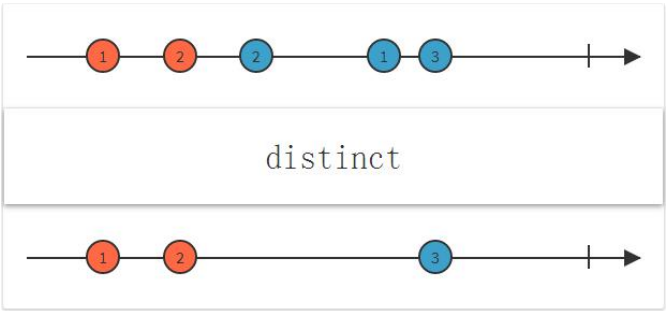
## 前言

年轻的老司机们，我这么勤的为大家分享，却少有催更的，好吧。其实写这个系列不是为了吸睛，那咱们继续写我们的 RxJava 2.x 的操作符。

## 正题

### distinct

这个操作符非常的简单、通俗、易懂，就是简单的去重嘛，我甚至都不想贴代码，但人嘛，总得持之以恒。




```
Observable.just(1, 1, 1, 2, 2, 3, 4, 5)
                .distinct()
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        mRxOperatorsText.append("distinct : " + integer + "\n");
```

```
                    Log.e(TAG, "distinct : " + integer + "\n");
                }
            });
```

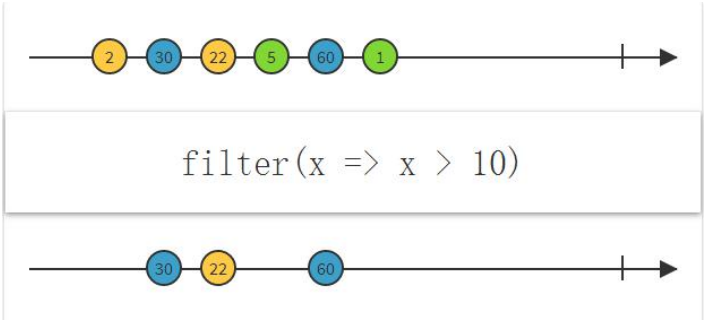
输出：

```
06-23 11:19:06.324 21033-21033/com.nanchen.rxjava2examples E/RxDistinctActivity: distinct : 1
06-23 11:19:06.324 21033-21033/com.nanchen.rxjava2examples E/RxDistinctActivity: distinct : 2
06-23 11:19:06.324 21033-21033/com.nanchen.rxjava2examples E/RxDistinctActivity: distinct : 3
06-23 11:19:06.324 21033-21033/com.nanchen.rxjava2examples E/RxDistinctActivity: distinct : 4
06-23 11:19:06.324 21033-21033/com.nanchen.rxjava2examples E/RxDistinctActivity: distinct : 5
```

Log 日志显而易见，我们在经过 `dinstinct()` 后接收器接收到的事件只有 1,2,3,4,5 了。

## Filter

信我，`Filter` 你会很常用的，它的作用也很简单，过滤器嘛。可以接受一个参数，让其过滤掉不符合我们条件的值




```
Observable.just(1, 20, 65, -5, 7, 19)
                .filter(new Predicate<Integer>() {
                    @Override
                    public boolean test(@NonNull Integer integer) throws Exception {
                        return integer >= 10;
                    }
                }).subscribe(new Consumer<Integer>() {
            @Override
            public void accept(@NonNull Integer integer) throws Exception {
                mRxOperatorsText.append("filter : " + integer + "\n");
                Log.e(TAG, "filter : " + integer + "\n");
```

```
            }
        });
```

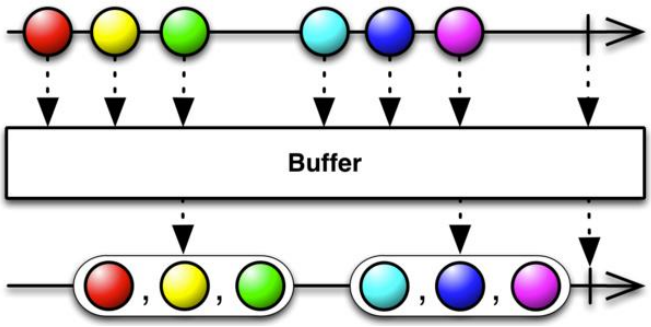
输出：

```
06-23 11:22:41.934 21033-21033/com.nanchen.rxjava2examples E/RxFilterActivity: filter : 20
06-23 11:22:41.944 21033-21033/com.nanchen.rxjava2examples E/RxFilterActivity: filter : 65
06-23 11:22:41.944 21033-21033/com.nanchen.rxjava2examples E/RxFilterActivity: filter : 19
```

可以看到，我们过滤器舍去了小于 10 的值，所以最好的输出只有 20, 65, 19。

## buffer

`buffer` 操作符接受两个参数，`buffer(count,skip)`，作用是将 `Observable` 中的数据按 `skip` (步长)分成最大不超过 count 的 `buffer` ，然后生成一个 `Observable` 。也许你还不太理解，我们可以通过我们的示例图和示例代码来进一步深化它。




```
Observable.just(1, 2, 3, 4, 5)
                .buffer(3, 2)
                .subscribe(new Consumer<List<Integer>>() {
                    @Override
                    public void accept(@NonNull List<Integer> integers) throws Exception {
                        mRxOperatorsText.append("buffer size : " + integers.size() + "\n");
                        Log.e(TAG, "buffer size : " + integers.size() + "\n");
                        mRxOperatorsText.append("buffer value : ");
                        Log.e(TAG, "buffer value : " );
                        for (Integer i : integers) {
                            mRxOperatorsText.append(i + "");
                            Log.e(TAG, i + "");
```

```
                    }

                    mRxOperatorsText.append("\n");

                    Log.e(TAG, "\n");

                }

            });
```

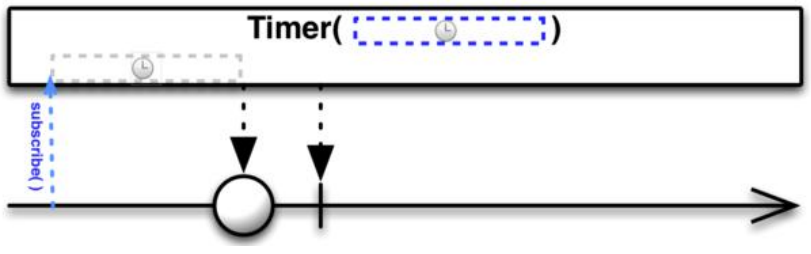
输出：

```
06-23 11:31:01.284 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: buffer size : 3
06-23 11:31:01.284 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: buffer value :
06-23 11:31:01.284 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 1
06-23 11:31:01.284 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 2
06-23 11:31:01.284 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 3
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: buffer size : 3
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: buffer value :
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 3
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 4
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 5
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: buffer size : 1
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: buffer value :
06-23 11:31:01.294 3320-3320/com.nanchen.rxjava2examples E/RxBufferActivity: 5
```

如图，我们把 1, 2, 3, 4, 5 依次发射出来，经过 `buffer` 操作符，其中参数 `skip` 为 2， `count` 为 3，而我们的输出 依次是 123，345，5。显而易见，我们 `buffer` 的第一个参数是 `count`，代表最大取值，在事件足够的时候，一般都是取 `count` 个值，然后每次跳过 `skip` 个事件。其实看 Log 日志，我相信大家都明白了。

## timer

`timer` 很有意思，相当于一个定时任务。在 1.x 中它还可以执行间隔逻辑，但在 2.x 中此功能被交给了 `interval`， 下一个会介绍。但需要注意的是， `timer` 和 `interval` 均默认在新线程。




```
mRxOperatorsText.append("timer start : " + TimeUtil.getNowStrTime() + "\n");

        Log.e(TAG, "timer start : " + TimeUtil.getNowStrTime() + "\n");

        Observable.timer(2, TimeUnit.SECONDS)
```

```
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread()) // timer 默认在新线程，所以需要切换回主线程
                .subscribe(new Consumer<Long>() {
                    @Override
                    public void accept(@NonNull Long aLong) throws Exception {
                        mRxOperatorsText.append("timer :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");
                        Log.e(TAG, "timer :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");
                    }
                });
```

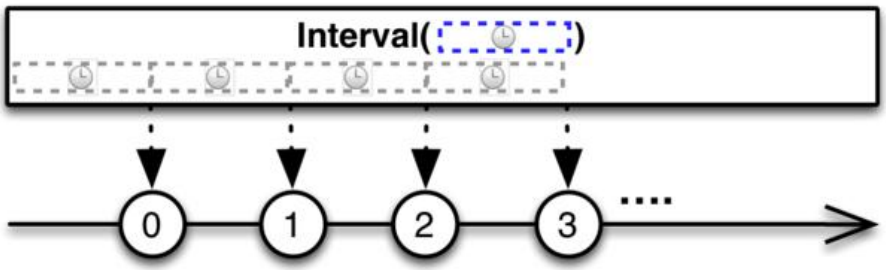
输出：

```
06-23 13:24:53.994 18169-18169/com.nanchen.rxjava2examples E/RxTimerActivity: timer start : 2017-06-23 13:24:54
06-23 13:24:56.244 18169-18169/com.nanchen.rxjava2examples E/RxTimerActivity: timer :0 at 2017-06-23 13:24:56
```

显而易见，当我们两次点击按钮触发这个事件的时候，接收被延迟了 2 秒。

## interval

如同我们上面可说，`interval` 操作符用于间隔时间执行某个操作，其接受三个参数，分别是第一次发送延迟，间隔时间，时间单位。




```
mRxOperatorsText.append("interval start : " + TimeUtil.getNowStrTime() + "\n");
        Log.e(TAG, "interval start : " + TimeUtil.getNowStrTime() + "\n");
        Observable.interval(3,2, TimeUnit.SECONDS)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread()) // 由于 interval 默认在新线程，所以我们应该切回主线程
```

```
                .subscribe(new Consumer<Long>() {

                    @Override

                    public void accept(@NonNull Long aLong) throws Exception {

                        mRxOperatorsText.append("interval :" + aLong + " at " +
TimeUtil.getNowStrTime() + "\n");

                        Log.e(TAG, "interval :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");

                    }

                });
```

输出：

```
06-23 13:31:56.724 25949-25949/com.nanchen.rxjava2examples E/RxIntervalActivity: interval start : 2017-06-23 13:31:56
06-23 13:31:59.974 25949-25949/com.nanchen.rxjava2examples E/RxIntervalActivity: interval :0 at 2017-06-23 13:31:59
06-23 13:32:01.974 25949-25949/com.nanchen.rxjava2examples E/RxIntervalActivity: interval :1 at 2017-06-23 13:32:01
06-23 13:32:03.974 25949-25949/com.nanchen.rxjava2examples E/RxIntervalActivity: interval :2 at 2017-06-23 13:32:03
06-23 13:32:05.974 25949-25949/com.nanchen.rxjava2examples E/RxIntervalActivity: interval :3 at 2017-06-23 13:32:05
06-23 13:32:07.974 25949-25949/com.nanchen.rxjava2examples E/RxIntervalActivity: interval :4 at 2017-06-23 13:32:07
```

如同 Log 日志一样，第一次延迟了 3 秒后接收到，后面每次间隔了 2 秒。
然而，心细的小伙伴可能会发现，由于我们这个是间隔执行，所以当我们的 Activity 都销毁的时候，实际上这个操作还依然在进行，所以，我们得花点小心思让我们在不需要它的时候干掉它。查看源码发现，我们 subscribe(Cousumer<? super T> onNext)返回的是 Disposable，我们可以在这上面做文章。

```
@CheckReturnValue
@SchedulerSupport(SchedulerSupport.NONE)
public final Disposable subscribe(Consumer<? super T> onNext) {
    return subscribe(onNext, Functions.ON_ERROR_MISSING, Functions.EMPTY_ACTION, Functions.emp
}
```

```
@Override

   protected void doSomething() {

       mRxOperatorsText.append("interval start : " + TimeUtil.getNowStrTime() + "\n");

       Log.e(TAG, "interval start : " + TimeUtil.getNowStrTime() + "\n");

       mDisposable = Observable.interval(3, 2, TimeUnit.SECONDS)

               .subscribeOn(Schedulers.io())

               .observeOn(AndroidSchedulers.mainThread()) // 由于 interval 默认在新线程，所以我
们应该切回主线程

               .subscribe(new Consumer<Long>() {
```

```
                @Override
                public void accept(@NonNull Long aLong) throws Exception {
                    mRxOperatorsText.append("interval :" + aLong + " at " +
TimeUtil.getNowStrTime() + "\n");
                    Log.e(TAG, "interval :" + aLong + " at " + TimeUtil.getNowStrTime() + "\n");
                }
            });
    }

    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mDisposable != null && !mDisposable.isDisposed()) {
            mDisposable.dispose();
        }
    }
```

哈哈，再次验证，解决了我们的疑惑。

## doOnNext

其实觉得 `doOnNext` 应该不算一个操作符，但考虑到其常用性，我们还是咬咬牙将它放在了这里。它的作用是让订阅者在接收到数据之前干点有意思的事情。假如我们在获取到数据之前想先保存一下它，无疑我们可以这样实现。

```
Observable.just(1, 2, 3, 4)
            .doOnNext(new Consumer<Integer>() {
                @Override
                public void accept(@NonNull Integer integer) throws Exception {
                    mRxOperatorsText.append("doOnNext 保存 " + integer + "成功" + "\n");
                    Log.e(TAG, "doOnNext 保存 " + integer + "成功" + "\n");
                }
```

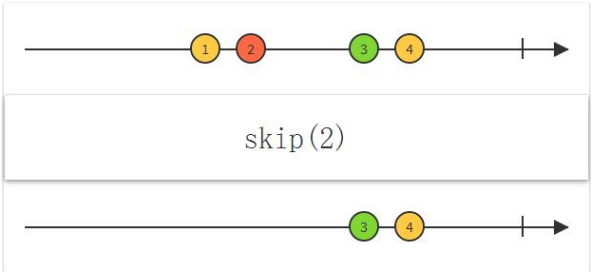
```
            }).subscribe(new Consumer<Integer>() {
        @Override
        public void accept(@NonNull Integer integer) throws Exception {
            mRxOperatorsText.append("doOnNext :" + integer + "\n");
            Log.e(TAG, "doOnNext :" + integer + "\n");
        }
    });
```

输出：

```
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext 保存 1成功
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext :1
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext 保存 2成功
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext :2
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext 保存 3成功
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext :3
06-23 13:51:37.854 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext 保存 4成功
06-23 13:51:37.864 32733-32733/com.nanchen.rxjava2examples E/RxDoOnNextActivity: doOnNext :4
```

## skip

`skip` 很有意思，其实作用就和字面意思一样，接受一个 long 型参数 count ，代表跳过 count 个数目开始接收。




```
Observable.just(1,2,3,4,5)
            .skip(2)
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(@NonNull Integer integer) throws Exception {
                    mRxOperatorsText.append("skip : "+integer + "\n");
```

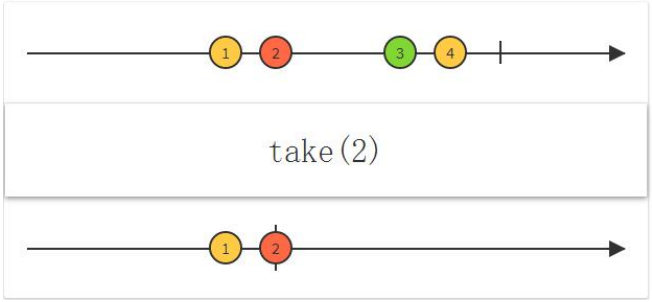
```
                    Log.e(TAG, "skip : "+integer + "\n");
                }
            });
```

输出：

```
06-23 13:54:53.794 32733-32733/com.nanchen.rxjava2examples E/RxSkipActivity: skip : 3
06-23 13:54:53.794 32733-32733/com.nanchen.rxjava2examples E/RxSkipActivity: skip : 4
06-23 13:54:53.794 32733-32733/com.nanchen.rxjava2examples E/RxSkipActivity: skip : 5
```

## take

take，接受一个 long 型参数 count ，代表至多接收 count 个数据。



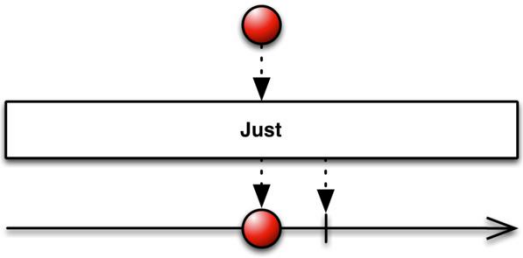

```
Flowable.fromArray(1,2,3,4,5)
            .take(2)
            .subscribe(new Consumer<Integer>() {
                @Override
                public void accept(@NonNull Integer integer) throws Exception {
                    mRxOperatorsText.append("take : "+integer + "\n");
                    Log.e(TAG, "accept: take : "+integer + "\n" );
                }
            });
```

输出：

```
06-23 13:59:10.484 32733-32733/com.nanchen.rxjava2examples E/RxTakeActivity: accept: take : 1
06-23 13:59:10.484 32733-32733/com.nanchen.rxjava2examples E/RxTakeActivity: accept: take : 2
```

## just

just，没什么好说的，其实在前面各种例子都说明了，就是一个简单的发射器依次调用 onNext() 方法。




```
Observable.just("1", "2", "3")
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        mRxOperatorsText.append("accept : onNext : " + s + "\n");
                        Log.e(TAG,"accept : onNext : " + s + "\n" );
                    }
                });
```

输出：

```
06-23 14:01:30.014 32733-32733/com.nanchen.rxjava2examples E/RxJustActivity: accept : onNext : 1
06-23 14:01:30.014 32733-32733/com.nanchen.rxjava2examples E/RxJustActivity: accept : onNext : 2
06-23 14:01:30.014 32733-32733/com.nanchen.rxjava2examples E/RxJustActivity: accept : onNext : 3
```

# RxJava 2.x 入门教程（四）

## 前言

最近很多小伙伴私信我，说自己很懊恼，对于 RxJava 2.x 系列一看就能明白，但自己写却又写不出来。如果 LZ 能放上实战情景教程就最好不过了。也是哈，单讲我们的操作符，也让我们的教程不温不火，但 LZ 自己选择的路，那跪着也要走完呀。所以，也就让我可怜的小伙伴们忍忍了，操作符马上就讲完了。

# 正题

## Single

顾名思义，`Single` 只会接收一个参数，而 `SingleObserver` 只会调用 `onError()` 或者 `onSuccess()`。

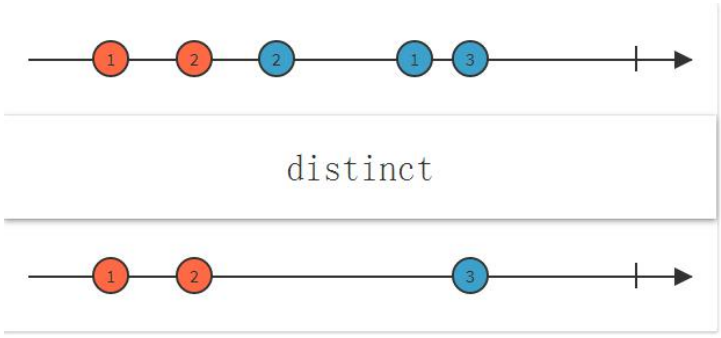
```
Single.just(new Random().nextInt())
                .subscribe(new SingleObserver<Integer>() {
                    @Override
                    public void onSubscribe(@NonNull Disposable d) {


                    }


                    @Override
                    public void onSuccess(@NonNull Integer integer) {
                        mRxOperatorsText.append("single : onSuccess : "+integer+"\n");
                        Log.e(TAG, "single : onSuccess : "+integer+"\n" );
                    }


                    @Override
                    public void onError(@NonNull Throwable e) {
                        mRxOperatorsText.append("single : onError : "+e.getMessage()+"\n");
                        Log.e(TAG, "single : onError : "+e.getMessage()+"\n");
                    }
                });
```

输出：

```
06-26 17:06:35.085 1999-1999/com.nanchen.rxjava2examples E/RxSingleActivity: single : onSuccess : -430368405
```

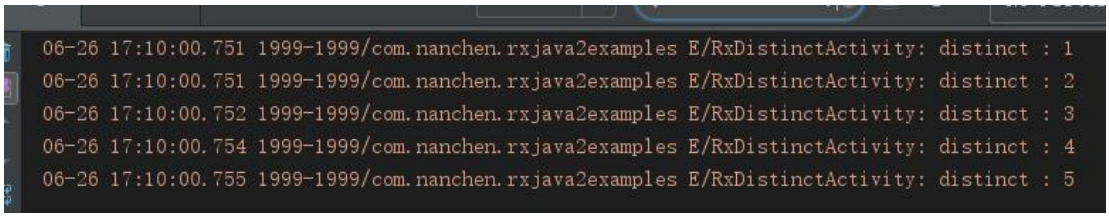
## distinct

去重操作符，简单的作用就是去重。




```
Observable.just(1, 1, 1, 2, 2, 3, 4, 5)
                .distinct()
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        mRxOperatorsText.append("distinct : " + integer + "\n");
                        Log.e(TAG, "distinct : " + integer + "\n");
                    }
                });
```

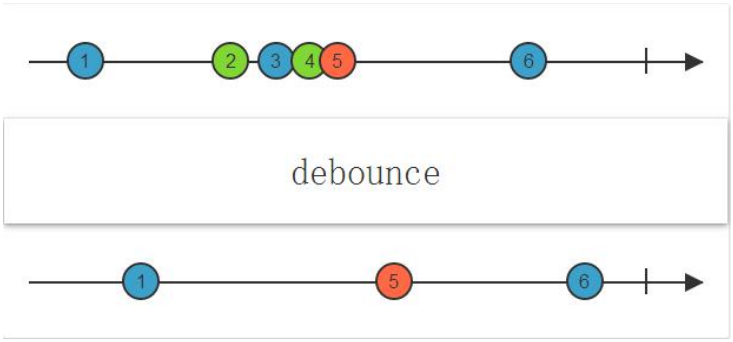
输出：




很明显，发射器发送的事件，在接收的时候被去重了。

## debounce

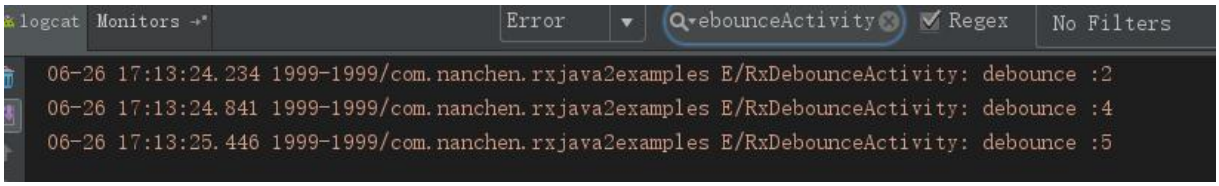
去除发送频率过快的项，看起来好像没啥用处，但你信我，后面绝对有地方很有用武之地。




```
Observable.create(new ObservableOnSubscribe<Integer>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Integer> emitter) throws Exception
{
                // send events with simulated time wait
                emitter.onNext(1); // skip
                Thread.sleep(400);
                emitter.onNext(2); // deliver
                Thread.sleep(505);
                emitter.onNext(3); // skip
                Thread.sleep(100);
                emitter.onNext(4); // deliver
                Thread.sleep(605);
                emitter.onNext(5); // deliver
                Thread.sleep(510);
                emitter.onComplete();
            }
        }).debounce(500, TimeUnit.MILLISECONDS)
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        mRxOperatorsText.append("debounce :" + integer + "\n");
```

```
                    Log.e(TAG,"debounce :" + integer + "\n");
                }
            });
```

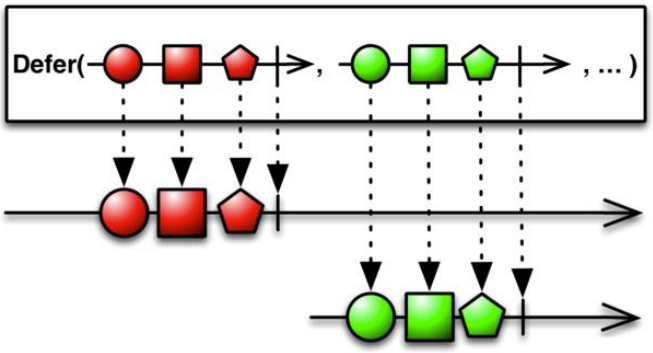
输出：




代码很清晰，去除发送间隔时间小于 500 毫秒的发射事件，所以 1 和 3 被去掉了。

## defer

简单地时候就是每次订阅都会创建一个新的 `Observable`，并且如果没有被订阅，就不会产生新的 `Observable`。




```
Observable<Integer> observable = Observable.defer(new Callable<ObservableSource<Integer>>() {
            @Override
            public ObservableSource<Integer> call() throws Exception {
                return Observable.just(1, 2, 3);
            }
        });


        observable.subscribe(new Observer<Integer>() {
            @Override
```

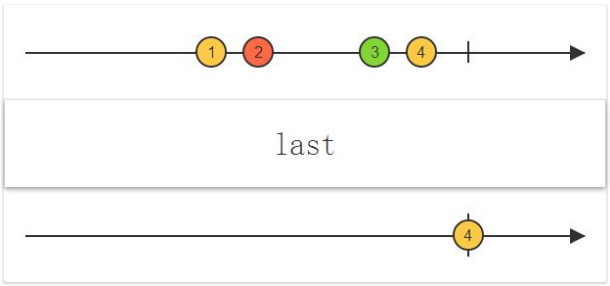
```
            public void onSubscribe(@NonNull Disposable d) {


            }


            @Override
            public void onNext(@NonNull Integer integer) {
                mRxOperatorsText.append("defer : " + integer + "\n");
                Log.e(TAG, "defer : " + integer + "\n");
            }


            @Override
            public void onError(@NonNull Throwable e) {
                mRxOperatorsText.append("defer : onError : " + e.getMessage() + "\n");
                Log.e(TAG, "defer : onError : " + e.getMessage() + "\n");
            }


            @Override
            public void onComplete() {
                mRxOperatorsText.append("defer : onComplete\n");
                Log.e(TAG, "defer : onComplete\n");
            }
        });
```

输出：

```
06-26 17:28:13.268 27689-27689/com.nanchen.rxjava2examples E/RxDeferActivity: defer : 1
06-26 17:28:13.269 27689-27689/com.nanchen.rxjava2examples E/RxDeferActivity: defer : 2
06-26 17:28:13.270 27689-27689/com.nanchen.rxjava2examples E/RxDeferActivity: defer : 3
06-26 17:28:13.273 27689-27689/com.nanchen.rxjava2examples E/RxDeferActivity: defer : onComplete
```

## last

`last` 操作符仅取出可观察到的最后一个值，或者是满足某些条件的最后一项。



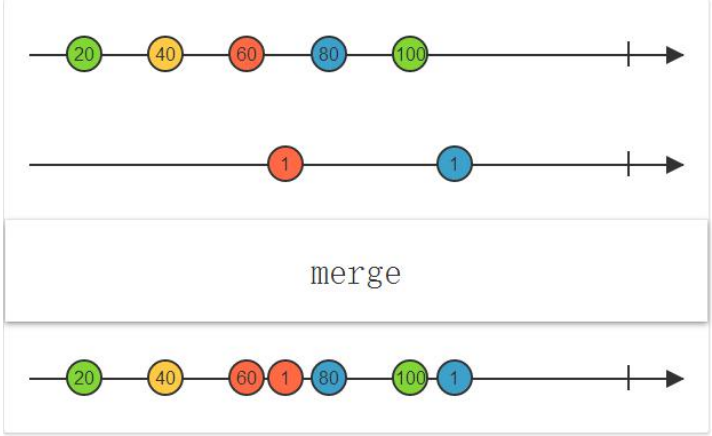

```
Observable.just(1, 2, 3)
                .last(4)
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        mRxOperatorsText.append("last : " + integer + "\n");
                        Log.e(TAG, "last : " + integer + "\n");
                    }
                });
```

输出：

```
06-26 17:32:27.371 27689-27689/com.nanchen.rxjava2examples E/RxLastActivity: last : 3
```

## merge

`merge` 顾名思义，熟悉版本控制工具的你一定不会不知道 merge 命令，而在 Rx 操作符中，`merge` 的作用是把多个 `Observable` 结合起来，接受可变参数，也支持迭代器集合。注意它和 `concat` 的区别在于，不用等到 发射器 A 发送完所有的事件再进行发射器 B 的发送。



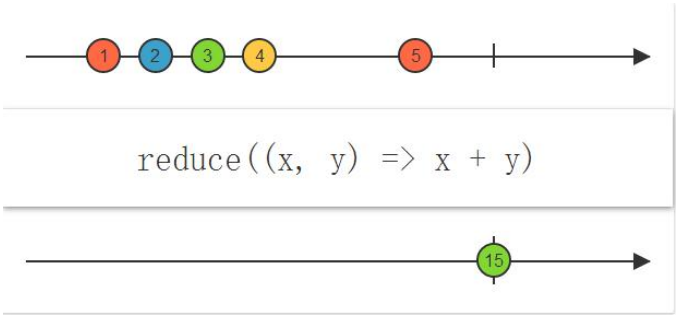
```
Observable.merge(Observable.just(1, 2), Observable.just(3, 4, 5))
                .subscribe(new Consumer<Integer>() {
                    @Override
                    public void accept(@NonNull Integer integer) throws Exception {
                        mRxOperatorsText.append("merge :" + integer + "\n");
                        Log.e(TAG, "accept: merge :" + integer + "\n" );
                    }
                });
```

输出：

```
06-26 17:38:06.529 27689-27689/com.nanchen.rxjava2examples E/RxMergeActivity: accept: merge :1
06-26 17:38:06.530 27689-27689/com.nanchen.rxjava2examples E/RxMergeActivity: accept: merge :2
06-26 17:38:06.531 27689-27689/com.nanchen.rxjava2examples E/RxMergeActivity: accept: merge :3
06-26 17:38:06.532 27689-27689/com.nanchen.rxjava2examples E/RxMergeActivity: accept: merge :4
06-26 17:38:06.533 27689-27689/com.nanchen.rxjava2examples E/RxMergeActivity: accept: merge :5
```

## reduce

`reduce` 操作符每次用一个方法处理一个值，可以有一个 `seed` 作为初始值。




```
Observable.just(1, 2, 3)
                .reduce(new BiFunction<Integer, Integer, Integer>() {
                    @Override
                    public Integer apply(@NonNull Integer integer, @NonNull Integer integer2)
throws Exception {
                        return integer + integer2;
                    }
```

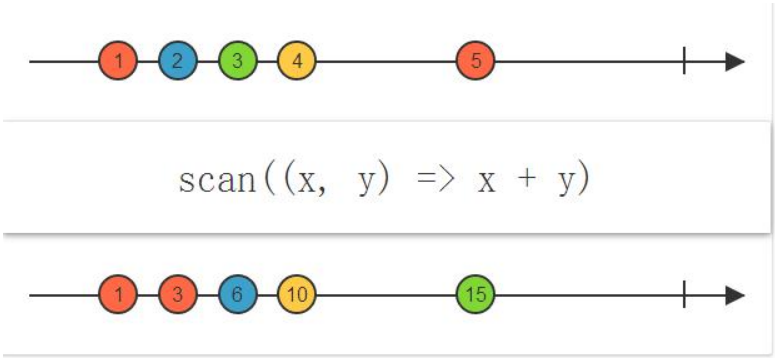
```
            }).subscribe(new Consumer<Integer>() {
        @Override
        public void accept(@NonNull Integer integer) throws Exception {
            mRxOperatorsText.append("reduce : " + integer + "\n");
            Log.e(TAG, "accept: reduce : " + integer + "\n");
        }
    });
```

输出：

可以看到，代码中，我们中间采用 reduce ，支持一个 function 为两数值相加，所以应该最后的值是：1 + 2 = 3 + 3 = 6 ， 而 Log 日志完美解决了我们的问题。

## scan

`scan` 操作符作用和上面的 `reduce` 一致，唯一区别是 `reduce` 是个只追求结果的坏人，而 `scan` 会始终如一地把每一个步骤都输出。




```
Observable.just(1, 2, 3)
                .scan(new BiFunction<Integer, Integer, Integer>() {
                    @Override
                    public Integer apply(@NonNull Integer integer, @NonNull Integer integer2)
throws Exception {
                        return integer + integer2;
                    }
```

```
            }).subscribe(new Consumer<Integer>() {
        @Override
        public void accept(@NonNull Integer integer) throws Exception {
            mRxOperatorsText.append("scan " + integer + "\n");
            Log.e(TAG, "accept: scan " + integer + "\n");
        }
    });
```

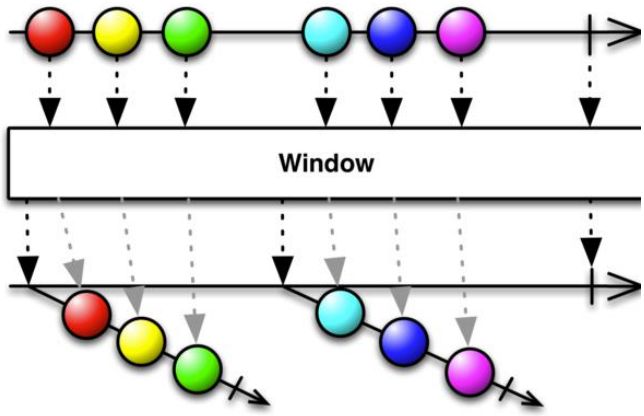
输出：

```
06-26 17:45:40.674 13224-13224/com.nanchen.rxjava2examples E/RxScanActivity: accept: scan 1
06-26 17:45:40.675 13224-13224/com.nanchen.rxjava2examples E/RxScanActivity: accept: scan 3
06-26 17:45:40.676 13224-13224/com.nanchen.rxjava2examples E/RxScanActivity: accept: scan 6
```

看日志，没毛病。

## window

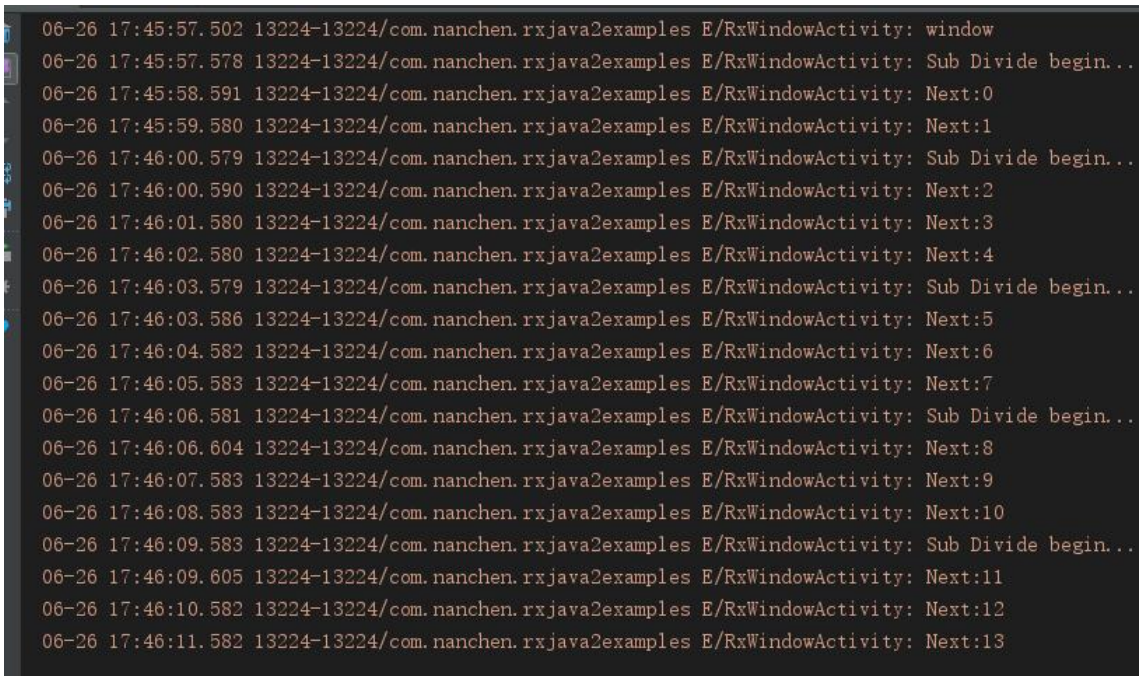
按照实际划分窗口，将数据发送给不同的 `Observable`




```
mRxOperatorsText.append("window\n");
    Log.e(TAG, "window\n");
    Observable.interval(1, TimeUnit.SECONDS) // 间隔一秒发一次
            .take(15) // 最多接收 15 个
            .window(3, TimeUnit.SECONDS)
            .subscribeOn(Schedulers.io())
```

```
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<Observable<Long>>() {
                    @Override
                    public void accept(@NonNull Observable<Long> longObservable) throws Exception
{
                        mRxOperatorsText.append("Sub Divide begin...\n");
                        Log.e(TAG, "Sub Divide begin...\n");
                        longObservable.subscribeOn(Schedulers.io())
                                .observeOn(AndroidSchedulers.mainThread())
                                .subscribe(new Consumer<Long>() {
                                    @Override
                                    public void accept(@NonNull Long aLong) throws Exception {
                                        mRxOperatorsText.append("Next:" + aLong + "\n");
                                        Log.e(TAG, "Next:" + aLong + "\n");
                                    }
                                });
                    }
                });
```

输出：




# RxJava 2.x 入门教程（五）

## 前言

终于如愿来到让我小伙伴们亢奋的 RxJava 2 使用场景举例了，前面几章中我们讲解完了 RxJava 1.x 到 RxJava 2.x 的异同以及 RxJava 2.x 的各种操作符使用，如有疑问，欢迎点击上方的链接进入你想要的环节。

## 正题

### 简单的网络请求

想必大家都知道，很多时候我们在使用 RxJava 的时候总是和 Retrofit 进行结合使用，而为了方便演示，这里我们就暂且采用 OkHttp3 进行演示，配合 `map`，`doOnNext`，线程切换进行简单的网络请求：

- 1）通过 Observable.create() 方法，调用 OkHttp 网络请求；
- 2）通过 map 操作符集合 gson，将 Response 转换为 bean 类；
- 3）通过 doOnNext() 方法，解析 bean 中的数据，并进行数据库存储等操作；
- 4）调度线程，在子线程中进行耗时操作任务，在主线程中更新 UI ；
- 5）通过 subscribe()，根据请求成功或者失败来更新 UI 。

```
Observable.create(new ObservableOnSubscribe<Response>() {
            @Override
            public void subscribe(@NonNull ObservableEmitter<Response> e) throws Exception {
                Builder builder = new Builder()
                        .url("http://api.avatardata.cn/MobilePlace/LookUp?key=ec47b85086be4dc
8b5d941f5abd37a4e&mobileNumber=13021671512")
                        .get();
                Request request = builder.build();
                Call call = new OkHttpClient().newCall(request);
                Response response = call.execute();
                e.onNext(response);
            }
        }).map(new Function<Response, MobileAddress>() {
                    @Override
                    public MobileAddress apply(@NonNull Response response) throws Exception {

                        Log.e(TAG, "map 线程:" + Thread.currentThread().getName() + "\n");
                        if (response.isSuccessful()) {
                            ResponseBody body = response.body();
                            if (body != null) {
                                Log.e(TAG, "map:转换前:" + response.body());
                                return new Gson().fromJson(body.string(), MobileAddress.class);
                            }
                        }
                        return null;
                    }
                }).observeOn(AndroidSchedulers.mainThread())
                .doOnNext(new Consumer<MobileAddress>() {
                    @Override
                    public void accept(@NonNull MobileAddress s) throws Exception {
```

```
                    Log.e(TAG, "doOnNext 线程:" + Thread.currentThread().getName() + "\n");
                    mRxOperatorsText.append("\ndoOnNext 线程:" +
Thread.currentThread().getName() + "\n");
                    Log.e(TAG, "doOnNext: 保存成功：" + s.toString() + "\n");
                    mRxOperatorsText.append("doOnNext: 保存成功：" + s.toString() + "\n");


                }
            }).subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<MobileAddress>() {
                @Override
                public void accept(@NonNull MobileAddress data) throws Exception {
                    Log.e(TAG, "subscribe 线程:" + Thread.currentThread().getName() + "\n");
                    mRxOperatorsText.append("\nsubscribe 线程:" +
Thread.currentThread().getName() + "\n");
                    Log.e(TAG, "成功:" + data.toString() + "\n");
                    mRxOperatorsText.append("成功:" + data.toString() + "\n");
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(@NonNull Throwable throwable) throws Exception {
                    Log.e(TAG, "subscribe 线程:" + Thread.currentThread().getName() + "\n");
                    mRxOperatorsText.append("\nsubscribe 线程:" +
Thread.currentThread().getName() + "\n");


                    Log.e(TAG, "失败：" + throwable.getMessage() + "\n");
                    mRxOperatorsText.append("失败：" + throwable.getMessage() + "\n");
                }
            });
```

为了方便，我们后面的讲解大部分采用开源的 Rx2AndroidNetworking 来处理，数据来源于天狗网等多个公共 API 接口。

```
mRxOperatorsText.append("RxNetworkActivity\n");

Rx2AndroidNetworking.get("http://api.avatardata.cn/MobilePlace/LookUp?key=ec47b85086be4dc8b5d941f5abd37a4e&mobileNumber=13021671512")
                .build()
                .getObjectObservable(MobileAddress.class)
                .observeOn(AndroidSchedulers.mainThread()) // 为 doOnNext() 指定在主线程，否则报错
                .doOnNext(new Consumer<MobileAddress>() {
                    @Override
                    public void accept(@NonNull MobileAddress data) throws Exception {
                        Log.e(TAG, "doOnNext:"+Thread.currentThread().getName()+"\n" );

mRxOperatorsText.append("\ndoOnNext:"+Thread.currentThread().getName()+"\n" );
                        Log.e(TAG,"doOnNext:"+data.toString()+"\n");
                        mRxOperatorsText.append("doOnNext:"+data.toString()+"\n");
                    }
                })
                .map(new Function<MobileAddress, ResultBean>() {
                    @Override
                    public ResultBean apply(@NonNull MobileAddress mobileAddress) throws Exception {
                        Log.e(TAG, "\n" );
                        mRxOperatorsText.append("\n");
                        Log.e(TAG, "map:"+Thread.currentThread().getName()+"\n" );

mRxOperatorsText.append("map:"+Thread.currentThread().getName()+"\n" );
                        return mobileAddress.getResult();
                    }
                })
```

```
                .subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<ResultBean>() {
                    @Override
                    public void accept(@NonNull ResultBean data) throws Exception {
                        Log.e(TAG, "subscribe 成功:"+Thread.currentThread().getName()+"\n" );
                        mRxOperatorsText.append("\nsubscribe 成
功:"+Thread.currentThread().getName()+"\n" );
                        Log.e(TAG, "成功:" + data.toString() + "\n");
                        mRxOperatorsText.append("成功:" + data.toString() + "\n");
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(@NonNull Throwable throwable) throws Exception {
                        Log.e(TAG, "subscribe 失败:"+Thread.currentThread().getName()+"\n" );
                        mRxOperatorsText.append("\nsubscribe 失
败:"+Thread.currentThread().getName()+"\n" );
                        Log.e(TAG, "失败: "+ throwable.getMessage()+"\n" );
                        mRxOperatorsText.append("失败: "+ throwable.getMessage()+"\n");
                    }
                });
```

## 先读取缓存，如果缓存没数据再通过网络请求获取数据后更新 UI

想必在实际应用中，很多时候（对数据操作不敏感时）都需要我们先读取缓存的数据，如果缓存没有数据，再通过网络请求获取，随后在主线程更新我们的 UI。

`concat` 操作符简直就是为我们这种需求量身定做。

`concat` 可以做到不交错的发射两个甚至多个 `Observable` 的发射事件，并且只有前一个 `Observable` 终止( `onComplete()` ) 后才会定义下一个 `Observable`。

利用这个特性，我们就可以先读取缓存数据，倘若获取到的缓存数据不是我们想要的，再调用 `onComplete()` 以执行获取网络数据的 `Observable`，如果缓存数据能应我们所需，则直接调用 `onNext()`，防止过度的网络请求，浪费用户的流量。

```
Observable<FoodList> cache = Observable.create(new ObservableOnSubscribe<FoodList>() {

            @Override

            public void subscribe(@NonNull ObservableEmitter<FoodList> e) throws Exception {

                Log.e(TAG, "create 当前线程:"+Thread.currentThread().getName() );

                FoodList data = CacheManager.getInstance().getFoodListData();



                // 在操作符 concat 中，只有调用 onComplete 之后才会执行下一个 Observable

                if (data != null){ // 如果缓存数据不为空，则直接读取缓存数据，而不读取网络数据

                    isFromNet = false;

                    Log.e(TAG, "\nsubscribe: 读取缓存数据:" );

                    runOnUiThread(new Runnable() {

                        @Override

                        public void run() {

                            mRxOperatorsText.append("\nsubscribe: 读取缓存数据:\n");

                        }

                    });



                    e.onNext(data);

                }else {

                    isFromNet = true;

                    runOnUiThread(new Runnable() {

                        @Override

                        public void run() {

                            mRxOperatorsText.append("\nsubscribe: 读取网络数据:\n");

                        }

                    });

                    Log.e(TAG, "\nsubscribe: 读取网络数据:" );
```

```
                e.onComplete();
            }



        }
    });

    Observable<FoodList> network =
Rx2AndroidNetworking.get("http://www.tngou.net/api/food/list")
            .addQueryParameter("rows",10+"")
            .build()
            .getObjectObservable(FoodList.class);



    // 两个 Observable 的泛型应当保持一致

    Observable.concat(cache,network)
            .subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<FoodList>() {
                @Override
                public void accept(@NonNull FoodList tngouBeen) throws Exception {
                    Log.e(TAG, "subscribe 成功:"+Thread.currentThread().getName() );
                    if (isFromNet){
                        mRxOperatorsText.append("accept : 网络获取数据设置缓存: \n");
                        Log.e(TAG, "accept : 网络获取数据设置缓存:
\n"+tngouBeen.toString() );
                        CacheManager.getInstance().setFoodListData(tngouBeen);
                    }
```

```
                    mRxOperatorsText.append("accept: 读取数据成功:" +
tngouBeen.toString()+"\n");
                    Log.e(TAG, "accept: 读取数据成功:" + tngouBeen.toString());
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(@NonNull Throwable throwable) throws Exception {
                    Log.e(TAG, "subscribe 失败:"+Thread.currentThread().getName() );
                    Log.e(TAG, "accept: 读取数据失败："+throwable.getMessage() );
                    mRxOperatorsText.append("accept: 读取数据失败：
"+throwable.getMessage()+"\n");
                }
            });
```

有时候我们的缓存可能还会分为 memory 和 disk ，实际上都差不多，无非是多写点 `Observable` ，然后通过 `concat` 合并即可。

## 多个网络请求依次依赖

想必这种情况也在实际情况中比比皆是，例如用户注册成功后需要自动登录，我们只需要先通过注册接口注册用户信息，注册成功后马上调用登录接口进行自动登录即可。

我们的 `flatMap` 恰好解决了这种应用场景，`flatMap` 操作符可以将一个发射数据的 `Observable` 变换为多个 `Observables` ，然后将它们发射的数据合并后放到一个单独的 `Observable`，利用这个特性，我们很轻松地达到了我们的需求。

```
Rx2AndroidNetworking.get("http://www.tngou.net/api/food/list")
            .addQueryParameter("rows", 1 + "")
            .build()
            .getObjectObservable(FoodList.class) // 发起获取食品列表的请求，并解析到 FootList
            .subscribeOn(Schedulers.io())        // 在 io 线程进行网络请求
            .observeOn(AndroidSchedulers.mainThread()) // 在主线程处理获取食品列表的请求结果
            .doOnNext(new Consumer<FoodList>() {
                @Override
```

```
                public void accept(@NonNull FoodList foodList) throws Exception {
                    // 先根据获取食品列表的响应结果做一些操作
                    Log.e(TAG, "accept: doOnNext :" + foodList.toString());
                    mRxOperatorsText.append("accept: doOnNext :" +
foodList.toString()+"\n");
                }
            })
            .observeOn(Schedulers.io()) // 回到 io 线程去处理获取食品详情的请求
            .flatMap(new Function<FoodList, ObservableSource<FoodDetail>>() {
                @Override
                public ObservableSource<FoodDetail> apply(@NonNull FoodList foodList)
throws Exception {
                    if (foodList != null && foodList.getTngou() != null &&
foodList.getTngou().size() > 0) {
                        return
Rx2AndroidNetworking.post("http://www.tngou.net/api/food/show")
                                .addBodyParameter("id", foodList.getTngou().get(0).getId()
+ "")
                                .build()
                                .getObjectObservable(FoodDetail.class);
                    }
                    return null;

                }
            })
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<FoodDetail>() {
                @Override
                public void accept(@NonNull FoodDetail foodDetail) throws Exception {
                    Log.e(TAG, "accept: success : " + foodDetail.toString());
                    mRxOperatorsText.append("accept: success : " +
foodDetail.toString()+"\n");
```

```
                }
            }, new Consumer<Throwable>() {
                @Override
                public void accept(@NonNull Throwable throwable) throws Exception {
                    Log.e(TAG, "accept: error :" + throwable.getMessage());
                    mRxOperatorsText.append("accept: error :" +
throwable.getMessage()+"\n");
                }
            });
```

## 结合多个接口的数据更新 UI

在实际应用中，我们极有可能会在一个页面显示的数据来源于多个接口，这时候我们的 `zip` 操作符为我们排忧解难。

`zip` 操作符可以将多个 `Observable` 的数据结合为一个数据源再发射出去。

```
Observable<MobileAddress> observable1 =
Rx2AndroidNetworking.get("http://api.avatardata.cn/MobilePlace/LookUp?key=ec47b85086be4dc8b
5d941f5abd37a4e&mobileNumber=13021671512")
            .build()
            .getObjectObservable(MobileAddress.class);


       Observable<CategoryResult> observable2 = Network.getGankApi()
               .getCategoryData("Android",1,1);


       Observable.zip(observable1, observable2, new BiFunction<MobileAddress, CategoryResult,
String>() {
           @Override
           public String apply(@NonNull MobileAddress mobileAddress, @NonNull CategoryResult
categoryResult) throws Exception {
               return "合并后的数据为：手机归属地："+mobileAddress.getResult().getMobilearea()+"
人名："+categoryResult.results.get(0).who;
```

```
            }
        }).subscribeOn(Schedulers.io())
                .observeOn(AndroidSchedulers.mainThread())
                .subscribe(new Consumer<String>() {
                    @Override
                    public void accept(@NonNull String s) throws Exception {
                        Log.e(TAG, "accept: 成功：" + s+"\n");
                    }
                }, new Consumer<Throwable>() {
                    @Override
                    public void accept(@NonNull Throwable throwable) throws Exception {
                        Log.e(TAG, "accept: 失败：" + throwable+"\n");
                    }
                });
```

## 间隔任务实现心跳

想必即时通讯等需要轮训的任务在如今的 APP 中已是很常见，而 RxJava 2.x 的 `interval` 操作符可谓完美地解决了我们的疑惑。

这里就简单的意思一下轮训。

```
private Disposable mDisposable;

    @Override
    protected void doSomething() {
        mDisposable = Flowable.interval(1, TimeUnit.SECONDS)
                .doOnNext(new Consumer<Long>() {
                    @Override
                    public void accept(@NonNull Long aLong) throws Exception {
                        Log.e(TAG, "accept: doOnNext : "+aLong );
                    }
```

```
            })
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe(new Consumer<Long>() {
                @Override
                public void accept(@NonNull Long aLong) throws Exception {
                    Log.e(TAG, "accept: 设置文本 : "+aLong );
                    mRxOperatorsText.append("accept: 设置文本 : "+aLong +"\n");
                }
            });
    }


    /**
     * 销毁时停止心跳
     */
    @Override
    protected void onDestroy() {
        super.onDestroy();
        if (mDisposable != null){
            mDisposable.dispose();
        }
    }
```