关于 RxJava 最友好的文章—— RxJava 2.0 全新来袭

Dreawer · 2 个月前

> **来源：关于 *RxJava* 最友好的文章—— *RxJava 2.0* 全新来袭**
>
> **作者：拉丁吴（转载已获得作者许可）**

- **文中图片点击一次就能看见大图了！**
- **关于RxJava最友好的文章（初级篇）**
- 关于RxJava最友好的文章（进阶篇）
- 关于RxJava最友好的文章——背压（Backpressure）

前言

之前写RxJava相关文章的时候，就有人想让我谈谈RxJava2.0的新特性，说实话，一开始我是拒绝的。因为在我看来，RxJava2.0虽然是版本的重大升级，但总归还是RxJava，升级一个版本还能上天是咋的？了解一下它的更新文档不就好了么？真的有必要单出一篇文章来谈这个么？

但是详细的了解了RxJava2.0以及部分源码之后，我觉得还是有必要对RxJava2.0做一个说明，帮助大家对于RxJava有更好的认识。

铺垫

假如你还不是很熟悉RxJava，或者对于背压这个概念（2.0更新中会涉及到背压的概念）很模

糊，希望你也能读一读下面两篇铺垫的文章：

- 关于RxJava最友好的文章
- 关于RxJava最友好的文章----背压

关于背压的那篇文章本来是本文的一部分，因为篇幅过大，被剥离出去了，所以建议大家有时间也一并阅读。

正文

RxJava2.0有很多的更新，一些改动甚至冲击了我之前的文章里的内容，这也是我想写这篇文章的原因之一。不过想要写这篇文章其实也是有难度的，因为相关的资料去其实是很少的，还得自己硬着头皮上....不过俗话说得好，有困难要上，没有困难创造困难也要上。

在这里，我会按照我们之前关于RxJava的文章的讲述顺序：观察者模式，操作符，线程调度，这三个方面依次看有哪些更新。

添加依赖

这个估计得放在最前面。

Android端使用RxJava需要依赖新的包名：

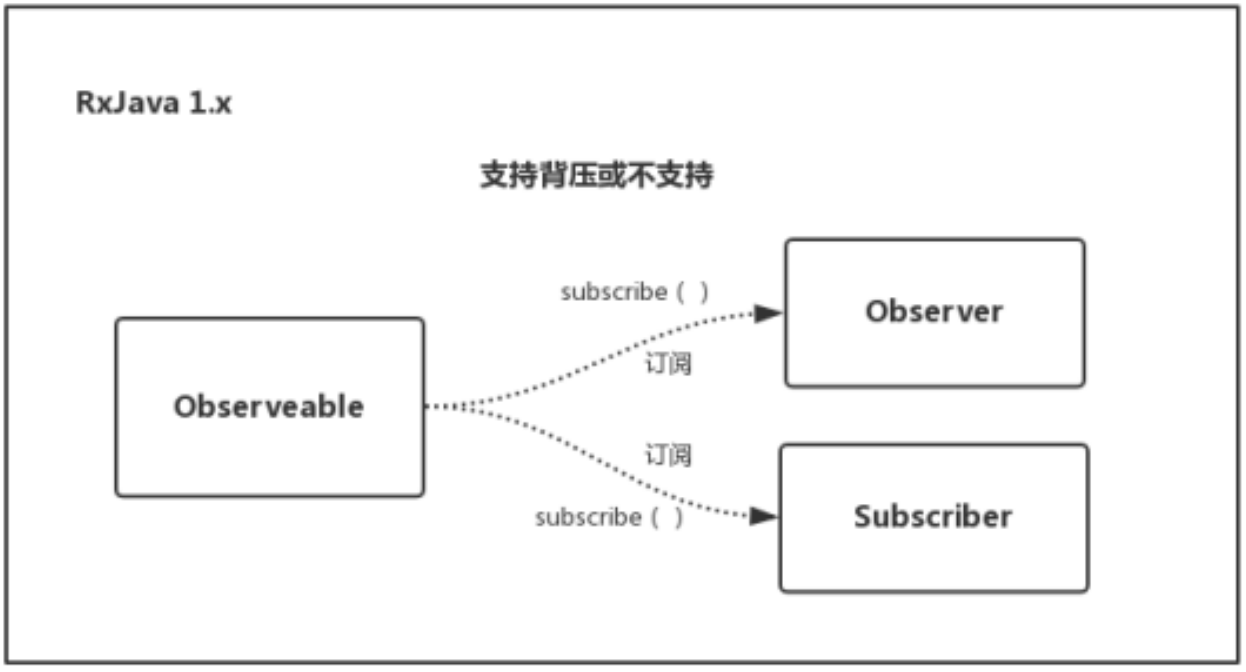
```
//RxJava的依赖包（我使用的最新版本）
compile 'io.reactivex.rxjava2:rxjava:2.0.1'
//RxAndroid的依赖包
compile 'io.reactivex.rxjava2:rxandroid:2.0.1'
```

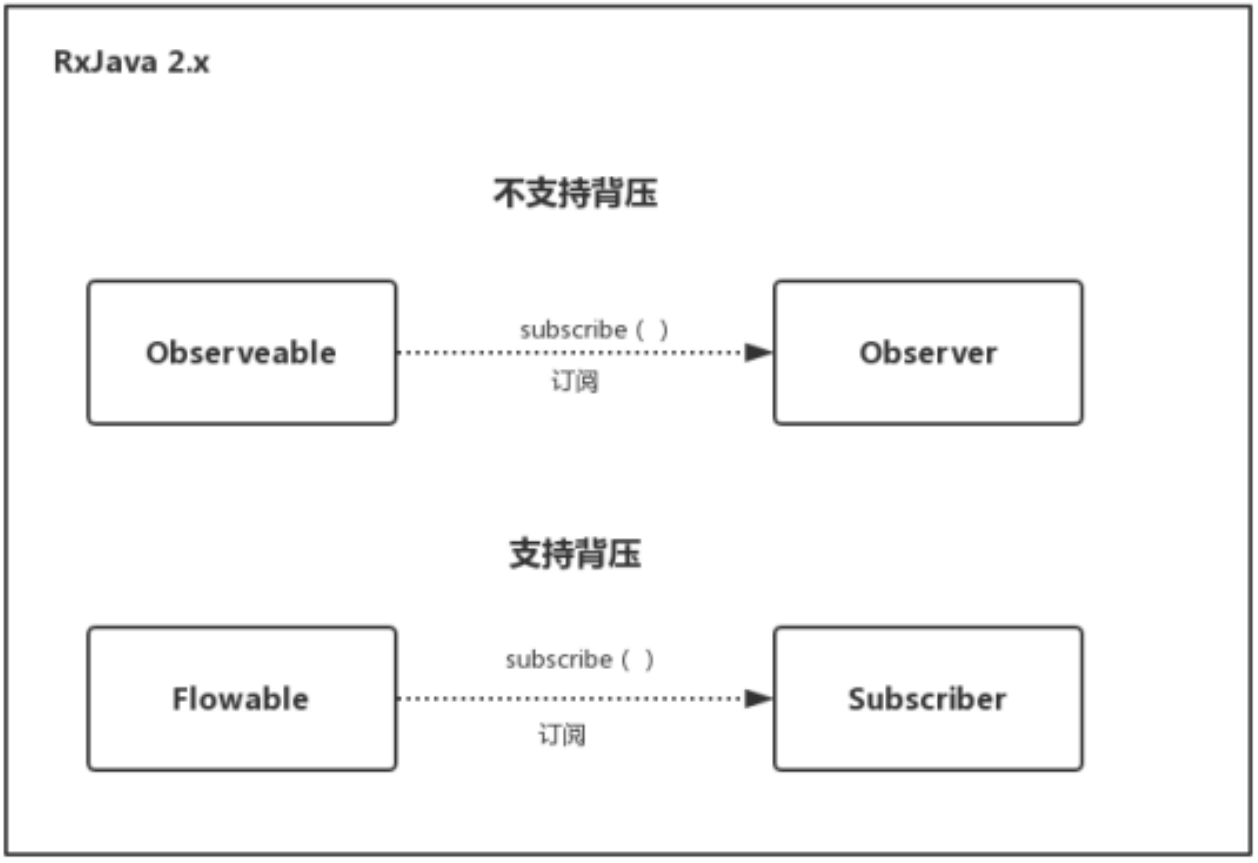
观察者模式

首先声明，**RxJava以观察者模式为骨架，在2.0中依然如此。**

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)





RxJava2.X中，**Observeable用于订阅Observer**，是不支持背压的，而**Flowable用于订阅Subscriber**，是支持背压(Backpressure)的。

关于背压这个概念以及它在1.0版本中的缺憾在上一篇文章中我已经介绍到了，如果你不是很清楚，我在这里在做一个介绍：**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，在1.0中，关于背压最大的遗憾，就是集中在Observable这个类中，导致有的Observable支持背压，有的不支持。为了解决这种缺憾，新版本把支持背压和不支持背压的Observable区分开来。

Observable/Observer

Observable正常用法：

```
   Observable mObservable=Observable.create(new ObservableOnSubscribe<Integer>(
            @Override
            public void subscribe(ObservableEmitter<Integer> e) throws Excepti
                e.onNext(1);
                e.onNext(2);
                e.onComplete();
            }
        });

Observer mObserver=new Observer<Integer>() {
            //这是新加入的方法，在订阅后发送数据之前，
            //回首先调用这个方法，而Disposable可用于取消订阅
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onNext(Integer value) {

            }

            @Override
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {

            }
        };

mObservable.subscribe(mObserver);
```

这种观察者模型是不支持背压的。

啥叫不支持背压呢？

当被观察者快速发送大量数据时，下游不会做其他处理，即使数据大量堆积，调用链也不会报MissingBackpressureException,消耗内存过大只会OOM

我在测试的时候，快速发送了100000个整形数据，下游延迟接收，结果被观察者的数据全部发送出去了，内存确实明显增加了，遗憾的是没有OOM。

所以，当我们使用Observable/Observer的时候，我们需要考虑的是，数据量是不是很大(官方给出以1000个事件为分界线，仅供各位参考)

Flowable/Subscriber

```
Flowable.range(0,10)
.subscribe(new Subscriber<Integer>() {
    Subscription sub;
    //当订阅后，会首先调用这个方法，其实就相当于onStart(),
    //传入的Subscription s参数可以用于请求数据或者取消订阅
    @Override
    public void onSubscribe(Subscription s) {
        Log.w("TAG","onsubscribe start");
        sub=s;
        sub.request(1);
        Log.w("TAG","onsubscribe end");
    }

    @Override
    public void onNext(Integer o) {
        Log.w("TAG","onNext--->"+o);
        sub.request(1);
    }
    @Override
    public void onError(Throwable t) {
        t.printStackTrace();
    }
    @Override
    public void onComplete() {
        Log.w("TAG","onComplete");
    }
```

```
        });
```

输出如下：

```
onsubscribe start
onNext--->0
onNext--->1
onNext--->2
...
onNext--->10
onComplete
onsubscribe end
```

Flowable是支持背压的，也就是说，一般而言，上游的被观察者会响应下游观察者的数据请求，下游调用request(n)来告诉上游发送多少个数据。这样避免了大量数据堆积在调用链上，使内存一直处于较低水平。

当然，Flowable也可以通过creat()来创建：

```
        Flowable.create(new FlowableOnSubscribe<Integer>() {
            @Override
            public void subscribe(FlowableEmitter<Integer> e) throws Exception
                e.onNext(1);
                e.onNext(2);
                e.onNext(3);
                e.onNext(4);
                e.onComplete();
            }
        }
        //需要指定背压策略
        , BackpressureStrategy.BUFFER);
```

Flowable虽然可以通过create()来创建，但是你必须指定背压的策略，以保证你创建的Flowable是支持背压的（这个在1.0的时候就很难保证，可以说RxJava2.0收紧了create()的权限）。

根据上面的代码的结果输出中可以看到，当我们调用subscription.request(n)方法的时候，不等onSubscribe()中后面的代码执行，就会立刻执行到onNext方法，因此，如果你在onNext方法中使用到需要初始化的类时，应当尽量在subscription.request(n)这个方法调用之前做好初始化的工作;

当然，这也不是绝对的，我在测试的时候发现，通过create（ ）自定义Flowable的时候，即使调用了subscription.request(n)方法，也会等onSubscribe（ ）方法中后面的代码都执行完之后，才开始调用onNext。

> **TIPS: 尽可能确保在request（ ）之前已经完成了所有的初始化工作，否则就有空指针的风险。**

其他观察者模式

当然，除了上面这两种观察者，还有一类观察者

- Single/SingleObserver
- Completable/CompletableObserver
- Maybe/MaybeObserver

其实这三者都差不多，Maybe/MaybeObserver可以说是前两者的复合体，因此以Maybe/MaybeObserver为例简单介绍一下这种观察者模式的用法

```
//判断是否登陆
Maybe.just(isLogin())
    //可能涉及到IO操作，放在子线程
    .subscribeOn(Schedulers.newThread())
    //取回结果传到主线程
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe(new MaybeObserver<Boolean>() {
            @Override
            public void onSubscribe(Disposable d) {

            }

            @Override
            public void onSuccess(Boolean value) {
                if(value){
                    ...
                }else{
                    ...
                }
            }

            @Override
```

```
            public void onError(Throwable e) {

            }

            @Override
            public void onComplete() {

            }
        });
```

上面就是Maybe/MaybeObserver的普通用法，你可以看到，实际上，这种观察者模式并不用于发送大量数据，而是发送单个数据，也就是说，当你只想要某个事件的结果（true or false）的时候，你可以用这种观察者模式

这是上面那些被观察者的上层接口：

```
//Observable接口
interface ObservableSource<T> {
    void subscribe(Observer<? super T> observer);
}
//Single接口
interface SingleSource<T> {
    void subscribe(SingleObserver<? super T> observer);
}
//Completable接口
interface CompletableSource {
    void subscribe(CompletableObserver observer);
}
//Maybe接口
interface MaybeSource<T> {
    void subscribe(MaybeObserver<? super T> observer);
}
//Flowable接口
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

其实我们可以看到，每一种观察者都继承自各自的接口，这也就把他们能完全的区分开，各自独立（特别是Observable和Flowable）,保证了他们各自的拓展或者配套的操作符不会相互影响。

例如flatMap操作符实现：

```
//Flowable中flatMap的定义
Flowable<R> flatMap(Function<? super T, ? extends Publisher<? extends R>> mapp

//Observable中flatMap的定义
Observable<R> flatMap(Function<? super T, ? extends ObservableSource<? extends
```

假如你想为Flowable写一个自定义的操作符，那么只要保证Function< Publisher >中的类型实现了Publisher接口即可。这么说可能很抽象，大家不理解其实也没关系，因为并不推荐大家自定义操作符，RxJava中的操纵符的组合已经可以满足大家的需求了。

当然，你也会注意到上面那些接口中的subscribe（ ）方法的返回类型为void了，在1.X中，这个方法一般会返回一个Subscription对象，用于取消订阅。现在，这个功能的对象已经被放到观察者Observer或者subscriber的内部实现方法中了，

Flowable/Subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}

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

上面的实例中，onSubscribe(Subscription s)传入的参数s就肩负着取消订阅的功能，当然，他也可以用于请求上游的数据。

在Observable/observer中，传入的参数是另一个对象

Observable/Observer

```
public interface Observer<T> {
   void onSubscribe(Disposable d);
    void onNext(T value);
    void onError(Throwable e);
    void onComplete();
```

```
}

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();
}
```

在Observer接口中，onSubscribe(Disposable d)方法传入的Disposable也是用于取消订阅，基本功能是差不多的，只不过命名不一致，大家知道就好。

其实这种设计可以说还是符合逻辑的，因为取消订阅这个动作就只有观察者（Observer等）才能做的，现在把它并入到观察者内部，也算顺理成章吧。

最后再提一点更新，就是被观察者不再接收null作为数据源了。

操作符相关

这一块其实可以说没什么改动，大部分之前你用过的操作符都没变，即使有所变动，也只是包名或类名的改动。大家可能经常用到的就是Action和Function。

- **Action相关**

之前我在文章里介绍过关于Action这类接口，在1.0中，这类接口是从Action0，Action1...往后排（数字代表可接受的参数），现在做出了改动

Rx1.0----------Rx2.0

Action1--------Action
Action1--------Consumer
Action2--------BiConsumer
后面的Action都去掉了，只保留了ActionN

- **Function相关**

同上，也是命名方式的改变

上面那两个类，**和RxJava1.0相比，他们都增加了throws Exception，也就是说，在这些方法做某些操作就不需要try-catch。**

例如：

```
Flowable.just("file.txt")
.map(name -> Files.readLines(name))
.subscribe(lines -> System.out.println(lines.size()), Throwable::printStackTra
```

Files.readLines(name)这类io方法本来是需要try-catch的，现在直接抛出异常，就可以放心的使用lambda ，保证代码的简洁优美。

- **doOnCancel/doOnDispose/unsubscribeOn**

以doOnCancel为例，大概就是当取消订阅时，会调用这个方法，例如：

```
Flowable.just(1, 2, 3)
.doOnCancel(() -> System.out.println("Cancelled!"))
.take(2)
.subscribe(System.out::println);
```

take新操符会取消后面那些还未被发送的事件，因而会触发doOnCancel

其他的一些操作符基本没变，或者只是改变了名字，在这里就不一一介绍了，需要提一下的是，很多操作符都有两套，一套用于Observable，一套用于Flowable。

线程调度

可以说这一块儿基本也没有改动，如果一定要说的话。

- 那就是去掉了Schedulers.immediate()这个线程环境
- 移除的还有Schedulers.test()（我好像从来没用过这个方法）
- io.reactivex.Scheduler这个抽象类支持直接调度自定义线程任务（这个我也没怎么用）

补充

如果你想把自己的RxJava1.0的迁移到2.0的版本，可以使用这个库RxJava2Interop ,它可以在Rxjava1.0和2.0之间相互转换，也就是说，不仅可以把1.0的代码迁移到2.0，你还可以把2.0的代码迁移到1.0,哈哈。

补充2

在RxJava1.0中，有的人会使用CompositeSubscription来收集Subscription，来统一取消订阅，现在在RxJava2.0中，由于subscribe()方法现在返回void，那怎么办呢？

其实在RxJava2.0中，**Flowable提供了subscribeWith这个方法可以返回当前订阅的观察者，并且通过ResourceSubscriber DisposableSubscriber等观察者来提供 Disposable的接口。**

所以，如果想要达成RxJava1.0的效果，现在应该是这样做：

CompositeDisposable composite = new CompositeDisposable();

composite.add(Flowable.range(1, 8).subscribeWith(subscriber));

这个subscriber 应该是 ResourceSubscriber 或者 DisposableSubscriber 的实例。

结尾

其实从整篇文章的分析来看，改动最大的还是观察者模式的实现，被拆分和细化了，主要分成了Observable和Flowable两大类，当然还有与之相关联的其他变动，**总体来看这一版本可以说是对于观察者和被观察者的重构。**

RxJava2.0的范例代码我没精力去写了，也正巧有位外国朋友已经写了RxJava2.0的demo,下面是他的项目地址：

RxJava2-Android-Samples

当然，学习2.0 的过程中有什么问题也可以在这里留言讨论。

后记

这篇文章半个月前就开始写了，但是一直不太满意，所以在草稿箱里躺了很久，但是本着不放弃任何一篇落后文章的信念，还是振作起来，完成关于RxJava2.0的介绍。你可能不信，写完顿时有了一种解(xu)脱的感觉。

身体被掏空...

附录

下面我截图展示一下2.0相对于1.0的一些改动的细节，仅做参考。

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

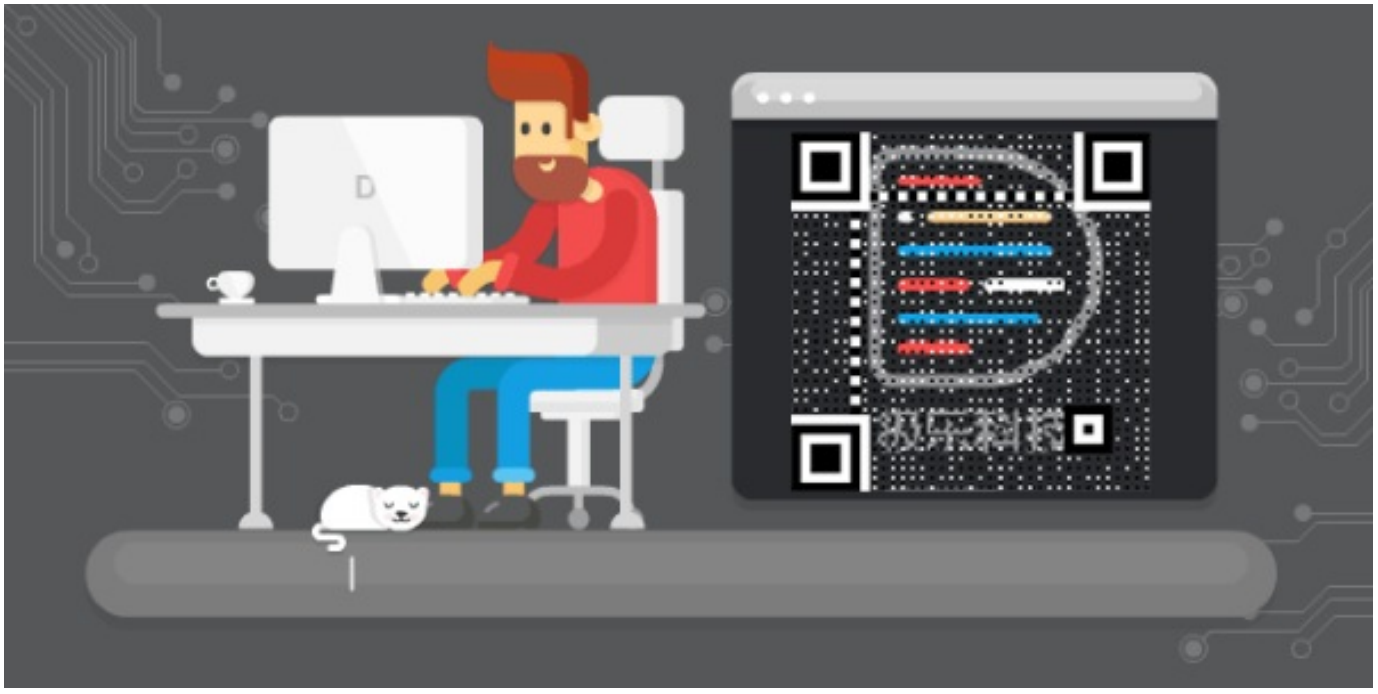
`takeLastBuffer`	dropped
N/A	added `test()` (returns TestSubscriber subscribed to this) with overloads to fluently test
`timeout(Func0<Observable>, ...)`	signature changed to `timeout(Publisher, ...)` and dropped the function, use `defer(Callable<Publisher>>)` if necessary
`toBlocking().y`	inlined as `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

其实这些都是官方给出的列表，截图在这里只是方便大家观摩。

在学习过程如果有任何疑问，请来极乐网(dreawer.com)提问，或者扫描下方二维码，关注极乐官方微信，在平台下方留言~



「所有赞赏将转给原作者 极乐网鼓励优质原创」

赞赏

还没有人赞赏，快来当第一个赞赏的人吧！

RxJava

17

分享 举报

文章被以下专栏收录

极乐科技

高效的中文IT技术平台

进入专栏

2 条评论

写下你的评论

Jarvis

赞赞赞，现在用的还是rxjava1.x，看来是时候切到rxjava2.x了，背压这个很关键

1 个月前

韩建飞

可以呀 好文。 如果业务跟背压没有啥关系 是不是还是继续用1.0算了

15 天前

推荐阅读

网页中实现正六边形的N种姿势

经常在别人家的网页上看到各中好看图形，其中就有正六边形和组合的蜂窝状图形。今天我们来盘点一下，网页上有哪些姿势实现这个效果姿势1css的… 查看全文 >

yuyuyu · 2 个月前　　发表于 极乐科技

va最友好的文章——背压（Backpressure）

xJava最友好的文章——背压（Backpressure）作者：拉丁吴（转载已获得作者
xJava最友好的文章（初级篇）关于RxJava最友好的文章（进阶篇）前言背压（Backpressure）可… 查看全文 >

Dreawer · 2 个月前　　发表于 极乐科技

卡普空最该重启的IP——《混沌军团》

本文系游民星空读者投稿：作者—HUSH13 又过了一年，工作与生活依旧在固执的日常中重复，仅存的心高气傲只是庆幸自己能够将“业余游… 查看全文 >

战术大米 · 15 天前 · 编辑精选　　发表于 游民星空独家专栏

如何优雅地在伯克利的公园里举行一场示威活动

发这篇文章，其实是因为一个误会：今天和同学聊起伯克利抗议事件，想要研究一下此次活动的合法性问题。我一开始误以为活动是在一处公园中进行… 查看全文 >

王瑞恩 · 3 天前 · 编辑精选 发表于 正义女神瞎了眼