

# 关于RxJava最友好的文章——背压（Backpressure）



作者 拉丁吴 (/u/1d8042233f67) [+ 关注](#)

2016.12.05 18:48 字数 2803 阅读 1001 评论 9 喜欢 28

(/u/1d8042233f67)

## 前言

背压（Backpressure）可能是所有想要深入运用RxJava的朋友必须理解的一个概念。

关于它的介绍，我本意是想写在RxJava2.0更新介绍的文章里的，可是写着写着发现，要完整介绍这个概念需要花费的篇幅太长，恰好目前对于背压的介绍文章比较少，所以决定单独拿出来，自成一篇。而关于RxJava2.0的文章修改之后就会发出来和大家探讨。

如果对于RxJava不是很熟悉，那么在这篇文章之前，我希望大家先看看那篇《关于Rxjava最友好的文章》，可以帮助大家很顺畅的了解RxJava。

## 从场景出发

让我们先忘掉背压（Backpressure）这个概念，从RxJava一个比较常见的工作场景说起。

RxJava是一个观察者模式的架构，当这个架构中被观察者(Observable)和观察者(Subscriber)处在不同的线程环境中时，由于者各自的工作量不一样，导致它们产生事件和处理事件的速度不一样，这就会出现两种情况：

- 被观察者产生事件慢一些，观察者处理事件很快。那么观察者就会等着被观察者发送事件，（好比观察者在等米下锅，程序等待，这没有问题）。
- 被观察者产生事件的速度很快，而观察者处理很慢。那就出问题了，如果不作处理的话，事件会堆积起来，最终挤爆你的内存，导致程序崩溃。（好比被观察者生产的大米没人吃，堆积最后就会烂掉）。

下面我们代码演示一下这种崩溃的场景：

```
//被观察者在主线程中，每1ms发送一个事件
Observable.interval(1, TimeUnit.MILLISECONDS)
    .subscribeOn(Schedulers.newThread())
    //将观察者的工作放在新线程环境中
    .observeOn(Schedulers.newThread())
    //观察者处理每1000ms才处理一个事件
    .subscribe(new Action1<Long>() {
        @Override
        public void call(Long aLong) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Log.w("TAG", "---->" + aLong);
        }
    });
```

在上面的代码中，被观察者发送事件的速度是观察者处理速度的1000倍

这段代码运行之后：



```
...
Caused by: rx.exceptions.MissingBackpressureException
...
...
```

抛出MissingBackpressureException往往就是因为，被观察者发送事件的速度太快，而观察者处理太慢，而且你还没有做相应措施，所以报异常。

而这个MissingBackpressureException异常里面就包含了Backpressure这个词，看来背压肯定和这种异常情况有关系。

那么背压（Backpressure）到底是什么呢？

---

## 关于背压（Backpressure）

我这两天翻阅了大量的中文和英文资料，我发现中文资料中，很多人对于背压（Backpressure）的理解是有很大的问题的，有的人把它看作一个需要避免的问题，或者程序的异常，有的人则干脆避而不谈，模棱两可，着实让人尴尬。

通过参考和对比大量的相关资料，我在这里先对背压（Backpressure）做一个明确的定义：**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。**

需要强调两点：

- 背压策略的一个前提是**异步环境**，也就是说，被观察者和观察者处在不同的线程环境中。
- 背压（Backpressure）并不是一个像flatMap一样可以在程序中直接使用的操作符，他只是一种控制事件流速的策略。

那么我们再回看上面的程序异常就很好理解了，就是当被观察者发送事件速度过快的情况下，我们没有做流速控制，导致了异常。

那么背压（Backpressure）策略具体是哪如何实现流速控制的呢？

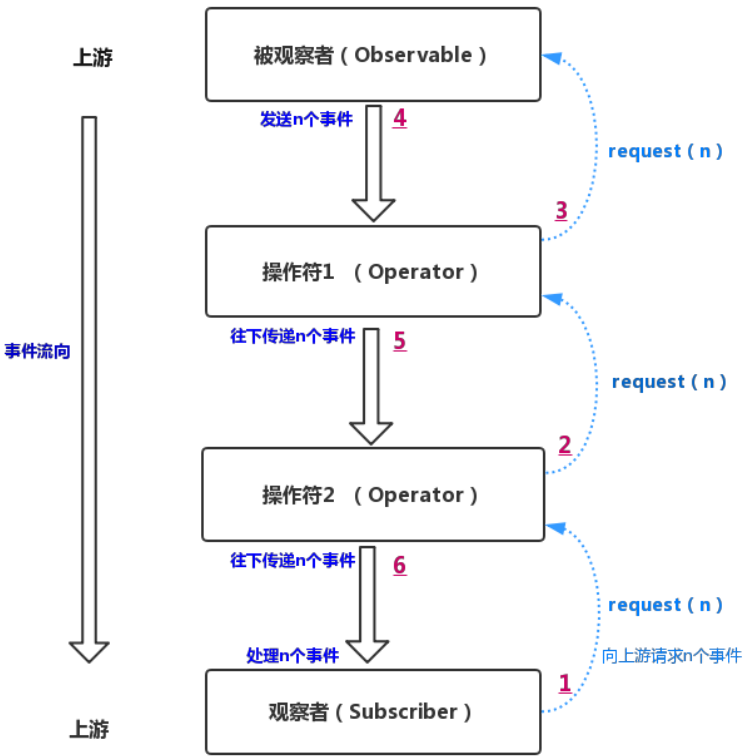
---

## 响应式拉取（reactive pull）

首先我们回忆之前那篇《关于Rxjava最友好的文章》，里面其实提到，在RxJava的观察者模型中，**被观察者是主动的推送数据给观察者，观察者是被动接收的。**而响应式拉取则反过来，**观察者主动从被观察者那里去拉取数据，而被观察者变成被动的等待通知再发送数据。**

结构示意图如下：





观察者可以根据自身实际情况按需拉取数据，而不是被动接收（也就相当于告诉上游观察者把速度慢下来），最终实现了上游被观察者发送事件的速度的控制，实现了背压的策略。

代码实例如下：

```
//被观察者将产生100000个事件
Observable observable=Observable.range(1,100000);
class MySubscriber extends Subscriber<T> {
    @Override
    public void onStart() {
        //一定要在onStart中通知被观察者先发送一个事件
        request(1);
    }

    @Override
    public void onCompleted() {
        ...
    }

    @Override
    public void onError(Throwable e) {
        ...
    }

    @Override
    public void onNext(T n) {
        ...
        ...
        //处理完毕之后，在通知被观察者发送下一个事件
        request(1);
    }
}

observable.observeOn(Schedulers.newThread())
    .subscribe(MySubscriber);
```

在代码中，传递事件开始前的onstart()中，调用了request(1)，通知被观察者先发送一个事件，然后在onNext()中处理完事件，再次调用request(1)，通知被观察者发送下一个事件....



注意在onNext()方法中，最好最后再调用request()方法。

如果你想取消这种backpressure 策略，调用quest(Long.MAX\_VALUE)即可。

实际上，在上面的代码中，你也可以不需要调用request(n)方法去拉取数据，程序依然能完美运行，这是因为range --> observeOn,这一段中间过程本身就是响应式拉取数据，observeOn这个操作符内部有一个缓冲区，Android环境下长度是16，它会告诉range最多发送16个事件，充满缓冲区即可。**不过话说回来，在观察者中使用request(n)这个方法可以使背压的策略表现得更加直观，更便于理解。**

如果你足够细心，会发现，在开头展示异常情况的代码中，使用的是interval这个操作符，但是在这里使用了range操作符，为什么呢？

这是因为interval操作符本身并不支持背压策略，它并不响应request(n)，也就是说，它发送事件的速度是不受控制的，而range这类操作符是支持背压的，它发送事件的速度可以被控制。

那么到底什么样的Observable是支持背压的呢？

## Hot and Cold Observables

需要说明的时，Hot Observables 和cold Observables并不是严格的概念区分，它只是对于两类Observable形象的描述

- Cold Observables：指的是那些在订阅之后才开始发送事件的Observable（每个Subscriber都能接收到完整的事件）。
- Hot Observables:指的是那些在创建了Observable之后，（不管是否订阅）就开始发送事件的Observable

其实也有创建了Observable之后调用诸如publish()方法就可以开始发送事件的,这里咱们暂且忽略。

我们一般使用的都是Cold Observable，除非特殊需求，才会使用Hot Observable,在这里，Hot Observable这一类是不支持背压的，而是Cold Observable这一类中也有一部分并不支持背压（比如interval，timer等操作符创建的Observable）。

懵逼了吧？

Tips: 都是Observable，结果有的支持背压，有的不支持，这就是RxJava1.X的一个问题。在2.0中，这种问题已经解决了，以后谈到2.0时再细说。

在那些不支持背压策略的操作符中使用响应式拉取数据的话，还是会抛出MissingBackpressureException。

那么，不支持背压的Observable如何做流速控制呢？

## 流速控制相关的操作符

### 过滤（抛弃）

就是虽然生产者产生事件的速度很快，但是把大部分的事件都直接过滤（浪费）掉，从而间接的降低事件发送的速度。



相关类似的操作符：Sample, ThrottleFirst....

以sample为例，

```
Observable.interval(1, TimeUnit.MILLISECONDS)

    .observeOn(Schedulers.newThread())
    //这个操作符简单理解就是每隔200ms发送里时间点最近那个事件，
    //其他的事件浪费掉
    .sample(200, TimeUnit.MILLISECONDS)
    .subscribe(new Action1<Long>() {
        @Override
        public void call(Long aLong) {
            try {
                Thread.sleep(200);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Log.w("TAG", "---->" + aLong);
        }
    });
```

这是以杀敌一千，自损八百的方式解决这个问题，因为抛弃了绝大部分的事件，而在我们使用RxJava 时候，我们自己定义的Observable产生的事件可能都是我们需要的，一般来说不会抛弃，所以这种方案有它的缺陷。

## 缓存

就是虽然被观察者发送事件速度很快，观察者处理不过来，但是可以选择先缓存一部分，然后慢慢读。

相关类似的操作符：buffer, window...

以buffer为例，

```
Observable.interval(1, TimeUnit.MILLISECONDS)

    .observeOn(Schedulers.newThread())
    //这个操作符简单理解就是把100毫秒内的事件打包成list发送
    .buffer(100, TimeUnit.MILLISECONDS)
    .subscribe(new Action1<List<Long>>() {
        @Override
        public void call(List<Long> aLong) {
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            Log.w("TAG", "---->" + aLong.size());
        }
    });
```

## 两个特殊操作符

对于不支持背压的Observable除了使用上述两类生硬的操作符之外，还有更好的选择：**onBackpressurebuffer**，**onBackpressureDrop**。

- **onBackpressurebuffer**：把observable发送出来的事件做缓存，当request方法被调用的时候，给下层流发送一个item(如果给这个缓存区设置了大小，那么超过了这个大小就会抛出异常)。
- **onBackpressureDrop**：将observable发送的事件抛弃掉，直到subscriber再次调用request (n) 方法的时候，就发送给它这之后的n个事件。

下面，我们以onBackpressureDrop为例说说用法：



```
Observable.interval(1, TimeUnit.MILLISECONDS)
    .onBackpressureDrop()
    .observeOn(Schedulers.newThread())
    .subscribe(new Subscriber<Long>() {

        @Override
        public void onStart() {
            Log.w("TAG", "start");
            request(1);
        }

        @Override
        public void onCompleted() {

        }

        @Override
        public void onError(Throwable e) {
            Log.e("ERROR", e.toString());
        }

        @Override
        public void onNext(Long aLong) {
            Log.w("TAG", "---->" + aLong);
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    });
```

这段代码的输出：

```
W/TAG: start
W/TAG: ---->0
W/TAG: ---->1
W/TAG: ---->2
W/TAG: ---->3
W/TAG: ---->4
W/TAG: ---->5
W/TAG: ---->6
W/TAG: ---->7
W/TAG: ---->8
W/TAG: ---->9
W/TAG: ---->10
W/TAG: ---->11
W/TAG: ---->12
W/TAG: ---->13
W/TAG: ---->14
W/TAG: ---->15
W/TAG: ---->1218
W/TAG: ---->1219
W/TAG: ---->1220
...
```

之所以出现0-15这样连贯的数据，就是因为observeOn操作符内部有一个长度为16的缓存区，它会首先请求16个事件缓存起来....

你可能会觉得这两个操作符和上面讲的过滤和缓存很类似，确实，功能上是有些类似，但是这两个操作符提供了更多的特性，那就是**可以响应下游观察者的request(n)方法了**，也就是说，**使用了这两种操作符，可以让原本不支持背压的Observable“支持”背压了。**

---

## 勘误

暂无

---

## 后记

讲了这么多终于要到尾声了。



下面我们总结一下：

- 背压是一种策略，具体措施是下游观察者通知上游的被观察者发送事件
- 背压策略很好的解决了异步环境下被观察者和观察者速度不一致的问题
- 在RxJava1.X中，同样是Observable，有的不支持背压策略，导致某些情况下，显得特别麻烦，出了问题也很难排查，使得RxJava的学习曲线变得十分陡峭。

这篇文章并不是为了让你学习在RxJava1.0中使用背压（如果你之前不了解背压的话），因为在1.0中，背压的设计并不十分完美。而是希望你**对背压有一个全面清晰的认识，对于它在RxJava1.0中的设计缺陷有所了解**即可。因为这篇文章本身是为了2.0做一个铺垫，后续的文章中我会继续谈到背压和使用背压的正确姿势。

随笔 (/nb/1401156)

举报文章 © 著作权归作者所有



拉丁吴 (/u/1d8042233f67)

写了 42193 字，被 310 人关注，获得了 516 个喜欢 (/u/1d8042233f67)




+ 关注

java python 算法，文艺青年，Android工程师

如果觉得我的文章对您有用，请随意打赏。您的支持将鼓励我继续创作！

赞赏支持

喜欢 (/sign\_in) | 28



更多分享


(http://cwb.assets.jianshu.io/notes/images/7438116



登录 (/sign\_in) 后发表评论

9条评论 只看作者

按喜欢排序 按时间正序 按时间倒序



原来是控股 (/u/24706b53c453)

2楼 · 2016.12.12 10:09 (/u/24706b53c453)

写得不错，请问如果项目刚准备接入RxJava，是直接上2.0比较好吗

赞

回复


陈日明 (/u/8d90a8c35cc6)： @原来是控股 (/users/24706b53c453) 可以使用

2016.12.12 10:11 回复

陈日明 (/u/8d90a8c35cc6)： 不错


2016.12.12 10:12 回复

添加新评论



寻蜜人 (/u/d9a6b94c2bd2)

3楼 · 2016.12.22 15:04 (/u/d9a6b94c2bd2)



http://www.jianshu.com/p/2c4799fa91a4

7/9

看完了《关于RxJava最友好的文章》过来的，第一次学习rxJava，1.x的写法基本了解了，但确实关于flatMap等几个操作符不太了解。背压虽然懂了，但感觉如果不是处理大数据的话，这个问题应该是不会出现的吧 🤔

👍 赞    💬 回复

拉丁吴 (/u/1d8042233f67): @寻蜜人 (/users/d9a6b94c2bd2) 是的，但是万一出现了这样的问题，也一定要知道是怎么会事哦，同时，推荐使用rxjava2.0

2016.12.22 16:00    💬 回复

寻蜜人 (/u/d9a6b94c2bd2): @拉丁吴 (/users/1d8042233f67) 嗯，有新的东西肯定用新的好，而且我还是刚入门，学的话最好学新的

2016.12.23 19:41    💬 回复

✍️ 添加新评论



旋哥 (/u/ec9e8bda5fbb)

4楼 · 2016.12.22 15:49

(/u/ec9e8bda5fbb)  
不错

👍 赞    💬 回复



boboyuwu (/u/e42b3f4e23f3)

5楼 · 2017.02.08 13:56

(/u/e42b3f4e23f3)

下面我们代码演示一下这种崩溃的场景：

```
//被观察者在主线程中，每1ms发送一个事件
Observable.interval(1, TimeUnit.MILLISECONDS)
//.subscribeOn(Schedulers.newThread())
//将观察者的工作放在新线程环境中
.observeOn(Schedulers.newThread())
//观察者处理每1000ms才处理一个事件
.subscribe(new Action1<Long>() {
    @Override (/users/d55323762b08)
    public void call(Long aLong) {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        Log.w("TAG", "---->" + aLong);
    }
});
```

这段为啥我写没有提示异常呢,程序正常运行为啥啊我的代码

```
Observable.interval(1, TimeUnit.MILLISECONDS)
.subscribeOn(AndroidSchedulers.mainThread())
.observeOn(Schedulers.newThread())
.subscribe(new Consumer<Long>() {
    @Override (/users/d55323762b08)
    public void accept(Long aLong) throws Exception {
        Thread.sleep(1000);
        Log.e("wubo", "value:" + aLong);
    }
});
```


👍 赞    💬 回复

拉丁吴 (/u/1d8042233f67): @boboyuwu (/users/e42b3f4e23f3) 你用的RxJava2.0版本吧？2.0对于针对背压做了新的设计，具体情况参考我关于rxjava2.0的文章

2017.02.08 14:00    💬 回复





 添加新评论

被以下专题收入，发现更多相似内容

