# RxJava中backpressure这个概念的理解

## RxJava

> 英文原文：https://github.com/ReactiveX/RxJava/wiki/Backpressure

## backpressure

在rxjava中会经常遇到一种情况就是*被观察者*发送消息十分迅速以至于它的操作符或者订阅者不能及时的响应这些消息。那么问题来了，要怎么处理这些慢慢堆积起来的消息呢?

举个栗子，使用 `zip` 操作符把两个无限大（假设）的*被观察者*压缩在一起，其中一个被观察者发送消息的速度是另外一个的两倍。一个比较天真（不科学）的做法就是把发送比较快的消息缓存起来，当比较慢的管擦者发送消息的时候取出来将他们结合在一起。但是这样会使得rxJava变得笨重且十分占用系统资源

在rxJava中有多重控制流以及反压力（backpressure）策略以应对当一个快速发送消息的被观察者遇到一个处理消息缓慢的观察者。本页面将会解释说明这些坑以及像你展示你应当怎么设计属于你自己的被观察者和操作符去应对流量控制（flow control）

## Hot and cold Observables, and multicasted（多路广播） Observables

一个cold Observable在它的订阅者订阅它的时候发送完整的数据序列，不管它的观察者们什么时候订阅它，或者观察者们什么以什么速率去消耗这个消息，都不会扰乱observable发送的完整性。例如把一个静态的迭代器（iterable）对象转换成了一个observable，这个observable将会对后来每个与它发生订阅关系的观察者发送通用的序列。cold observable的例子可能包括数据库查询的结果，文件序列，或者网络请求

Hot observable 不管有没有订阅者订阅，他们创建后就开发发射数据流。 一个比较好的示例就是 鼠标事件。 不管系统有没有订阅者监听鼠标事件，鼠标事件一直在发生，当有订阅者订阅后，从订阅后的事件开始发送给这个订阅者，之前的事件这个订阅者是接受不到的；如果订阅者取消订阅了，鼠标事件依然继续发射

当一个cold observable是multicast的时候，为了应对反压力，应该吧一个cold observable转换成一个hot observable

cold observable 相当于响应式拉（就是observer处理完了一个事件就从observable拉取下一个时间），hotobservable通常不能很好的处理响应式拉模型，但是它却是更好的处理本页讨论的流量控制问题的候选人，例如使用onBackpressureBuffer or onBackpressureDrop operators, throttling, buffers, or windows.

## 使用操作符避免发生Backpressure

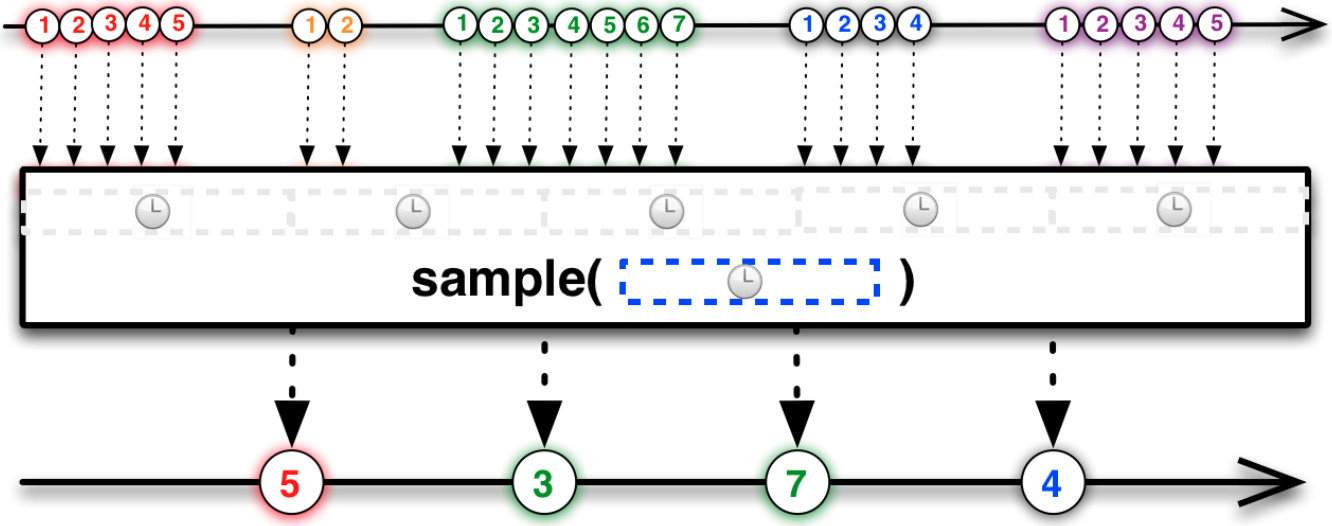
防止产生过度产生observable的第一道防线就是使用普通数组去减少observable发送消息的数量，在这一节会使用一些操作符去应对突然observable发送爆发性数据（一会没有，一会很多）就像下面的这张图片


## Throttling

sample( )或throttleLast( ) throttleFirst( )和throttleWithTimeout( )或debounce( ) 等操作符允许调节observable在其中官方侧发射项目的速度

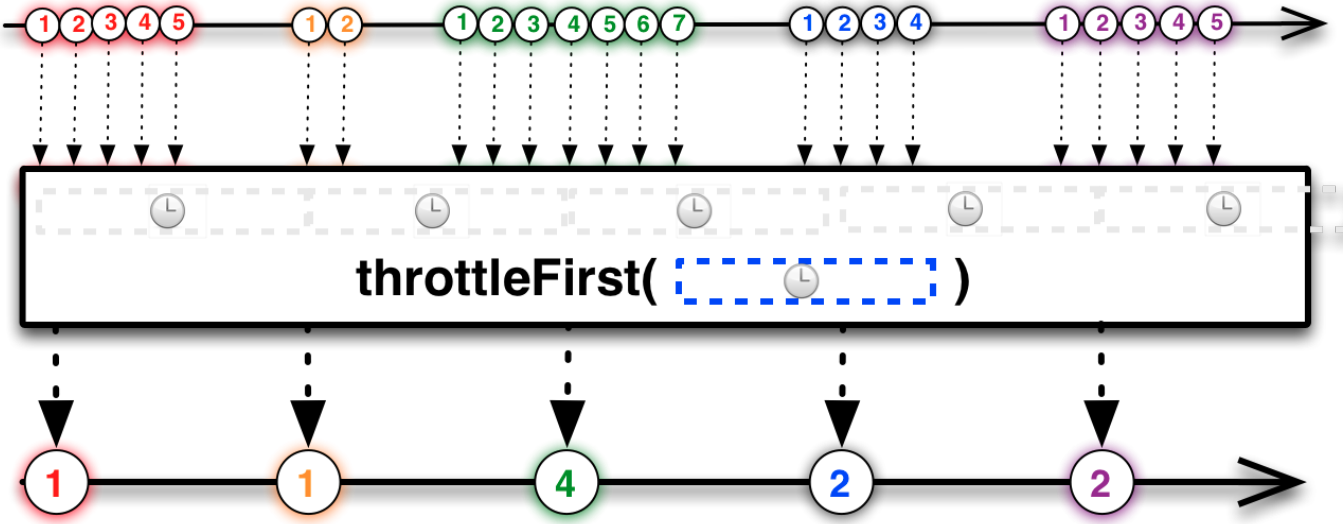
### sample (or throttleLast)

sample这个玩意就是把一段时间内采集到的observable发送的itme的**最后一个**item发送出去（意思是会丢失部分？）



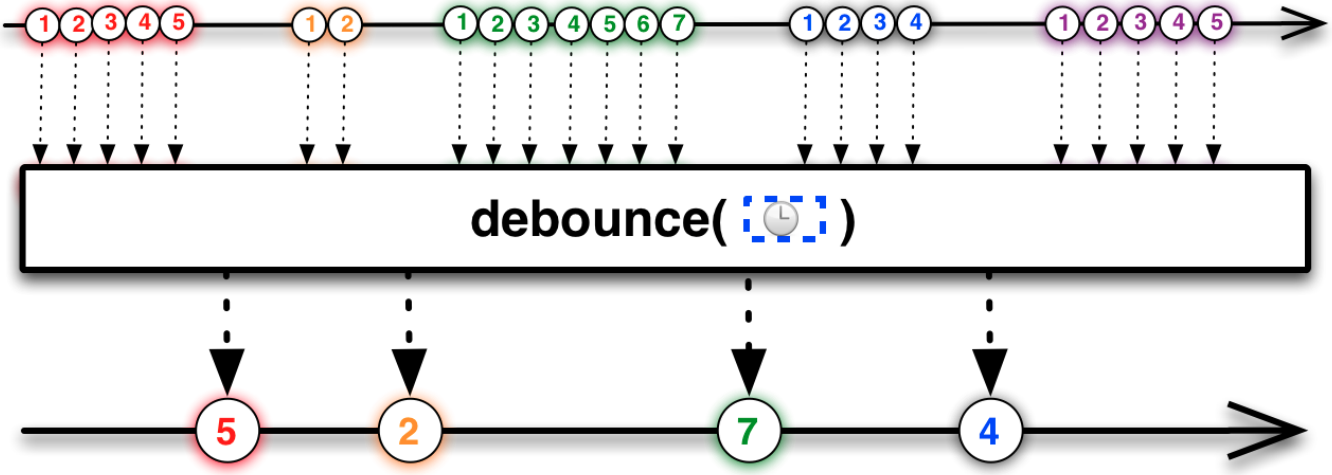

## throttleFirst

跟sample有点类似，但是并不是把观测到的最后一个item发送出去，而是把该时间段后下一个item发送出去




## debounce (or throttleWithTimeout)

debounce操作符会只发送两个在规定间隔内的时间发送的序列的最后一个




## Buffers and windows

buffer允许在不同规则下缓冲从observable发送出来的项目，具体查看buffer部分的java doc Window和Buffer类似，但不是发射来自原始Observable的数据包，它发射的是Observables，这些Observables中的每一个都发射原始Observable数据的一个子集，最后发射一个onCompleted通知。

## 使用线程阻塞

处理过快生产item的其他策略就是使用线程阻塞，但是这么做违背了响应式设计和非阻塞模型设计，但是它的确是一个可行的选择。在rxJava中并没有操作符可以做到这一点。

如果observable发送消息，subscriber消耗消息都是在同一个线程这将很好的处理这个问题，但是你要知道，在rxJava中，很多时候生产者和消费者都不在同一个线程

## 如何建立“响应式拉动（reactive pull）”backpressure

当subscribe订阅observable的时候可以通过调用subscribe.request（n），n是你想要的observable发送出来的量

## Reactive pull backpressure isn't magic

backpressure 不会使得过度生产的observable的问题消失，这只是提供了一种更好的解决问题的方法

让我们更仔细的研究刚刚说到的zip操作符的问题

这里有两个observable，a和b，b发射item比a更加的频繁，当你想zip这两个observable的时候，你需要把a发送出来的第n个和b发送出来的第n个对象处理，然而由于b发送出来的速率更快，这时候b已经发送出了n+1~n+m个消息了，这时候你要想要把a的n+1~n+m个消息结合的话，就必须持有b已经发送出来的n+1~n+m消息，同时，这意味着缓存的数量在不断的增长。

当然你可以给b添加操作符throttling，但是这意味着你将丢失某些从b发送出来的项，你真正想要做的其实就是告诉b：“b你需要慢下来，但是你要保持你给我的数据是完整的”。

响应式拉（reective pull）模型可以当你做到这一点，subscriber从observable那里拉取数据，这比较通常在observable那里推送数据这种模式形成鲜明的对比。

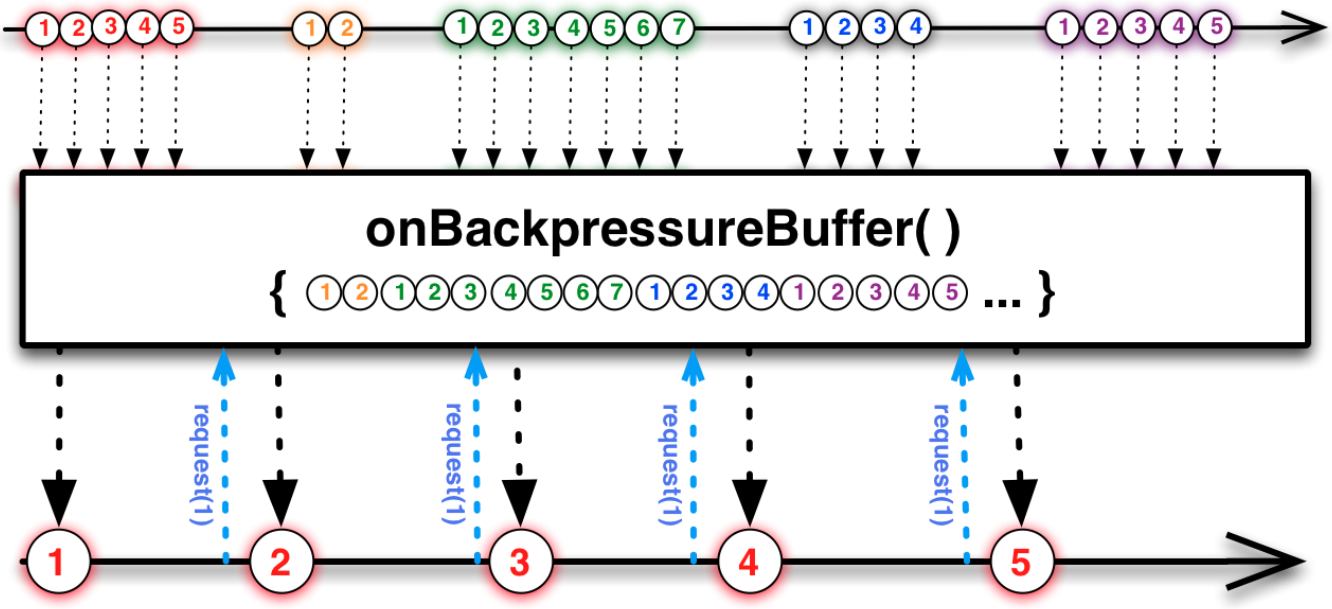
在rxJava中，zip操作符正是使用了这种技巧。它给每个源observable维护了一个小的缓存池，当它的缓存池满了以后，它将不会从源observable那里拉取item。每当zip发送一个item的时候，他从它的缓存池里面移除相应的项，并从源observable那里拉取下一个项

在rxJava中，很多操作符都使用了这种模式（响应式拉），但是有的操作符并没有使用这种模式，因为他们也许执行的操作跟源observable处于相同的进程。在这种情况下，由于消耗事件会阻塞本进程，所以这一项的工作完成后，才有机会收到下一项。还有另外一种情况，backpressure也是不适合的，因为他们有指定的其他方式去处理流量控制，这些特殊的情况在rxJava的java文档里面都会有详细说明为毛。

但是，observable a和b必须正确的响应request()方法，如果一个observable还没有被支持响应式拉（并不是每个observable都会支持），你可以采取以下其中一种操作都可以达到backpressure的行为：

**onBackpressurebuffer**

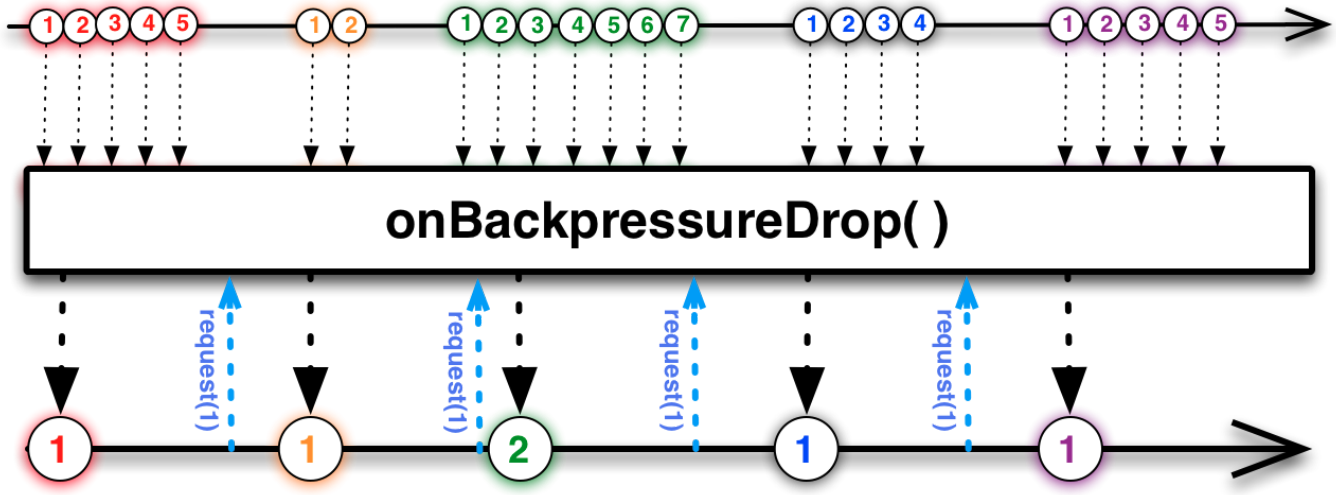
给observable发送出来的数据持有一个缓存，当request方法被调用的时候，给下层流发送一个item



这个操作符还有一个实验性的版本允许去设置这个缓存池的大小，但当缓存池满了以后将会终止执行并抛出异常

**onBackpressureDrop**

命令observable丢弃后来的事件，直到subscriber再次调用request（n）方法的时候，就发送给它的subscriber调用时间以后的n个事件。




2017-2-6 10:25 阅读(5988) 评论(0)

♥ 30

已关闭评论