**如何形象地描述 RxJava 中的背压和流控机制？**

之前我在知乎上受邀回答过一个关于RxJava背压（Backpressure）机制的问题，今天我把它整理出来，希望对更多的人能有帮助。

RxJava的官方文档中对于背压（Backpressure）机制比较系统的描述是下面这个：

<https://github.com/ReactiveX/RxJava/wiki/Backpressure>

但本文的题目既然是要“形象地”描述各个机制，自然会力求表达简洁，让人一看就懂。所以，下面我会尽量抛开一些抽象的描述，主要采用打比方的方式来阐明我对于这些机制的理解。

首先，从大的方面说，上面这篇文档的题目，虽然叫“Backpressure”（背压），但却是在讲述一个更大的话题——“Flow Control”（流控）。Backpressure只是Flow Control的其中一个方案。

在RxJava中，可以通过对Observable连续调用多个Operator组成一个调用链，其中数据从上游向下游传递。当上游发送数据的速度大于下游处理数据的速度时，就需要进行Flow Control了。

这就像小学做的那道数学题：一个水池，有一个进水管和一个出水管。如果进水管水流更大，过一段时间水池就会满（溢出）。这就是没有Flow Control导致的结果。

Flow Control有哪些思路呢？大概是有四种:

* (1) 背压（Backpressure）。
* (2) 节流（Throttling）。
* (3) 打包处理。
* (4) 调用栈阻塞（Callstack blocking）。

下面分别详细介绍。

注意：目前RxJava的1.x和2.x两个版本序列同时并存，2.x相对于1.x在接口上有很大变动，其中也包括Backpressure的部分。但是，这里要讨论的Flow Control机制中的相关概念，却都是适用的。

**Flow Control的几种思路**

**背压（Backpressure）**

Backpressure，也称为Reactive Pull，就是下游需要多少（具体是通过下游的request请求指定需要多少），上游就发送多少。这有点类似于TCP里的流量控制，接收方根据自己的接收窗口的情况来控制接收速率，并通过反向的ACK包来控制发送方的发送速率。

这种方案只对于所谓的cold Observable有效。cold Observable指的是那些允许降低速率的发送源，比如两台机器传一个文件，速率可大可小，即使降低到每秒几个字节，只要时间足够长，还是能够完成的。相反的例子是音视频直播，数据速率低于某个值整个功能就没法用了（这种就属于hot Observable了）。

**节流（Throttling）**

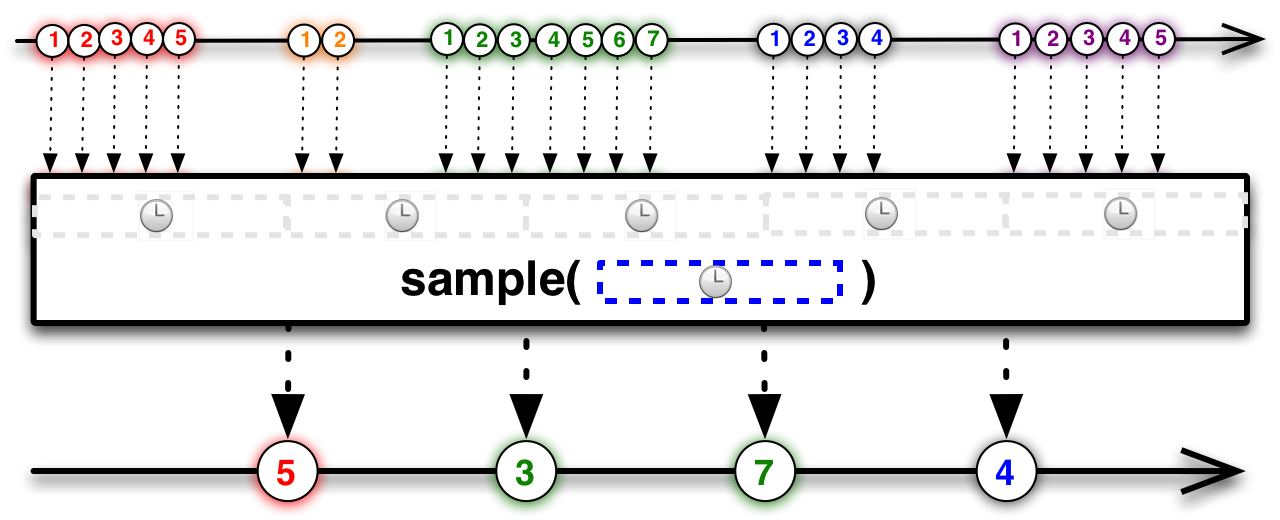
节流（Throttling），说白了就是丢弃。消费不过来，就处理其中一部分，剩下的丢弃。还是举音视频直播的例子，在下游处理不过来的时候，就需要丢弃数据包。

而至于处理哪些和丢弃哪些数据，就有不同的策略。主要有三种策略：

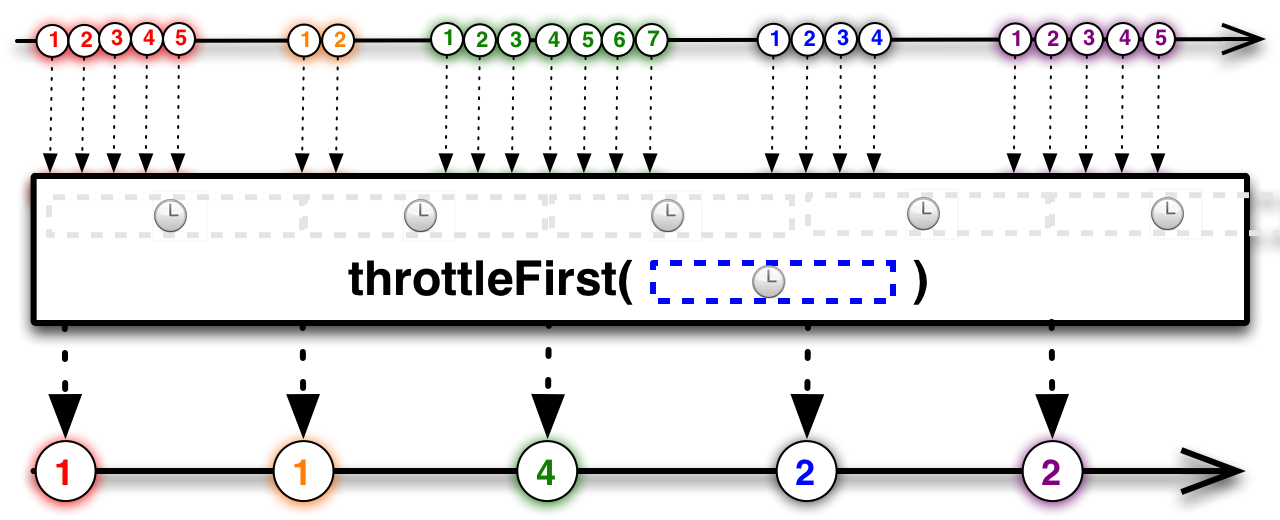
* sample (也叫throttleLast)
* throttleFirst
* debounce (也叫throttleWithTimeout)

从细的方面分别解释一下。

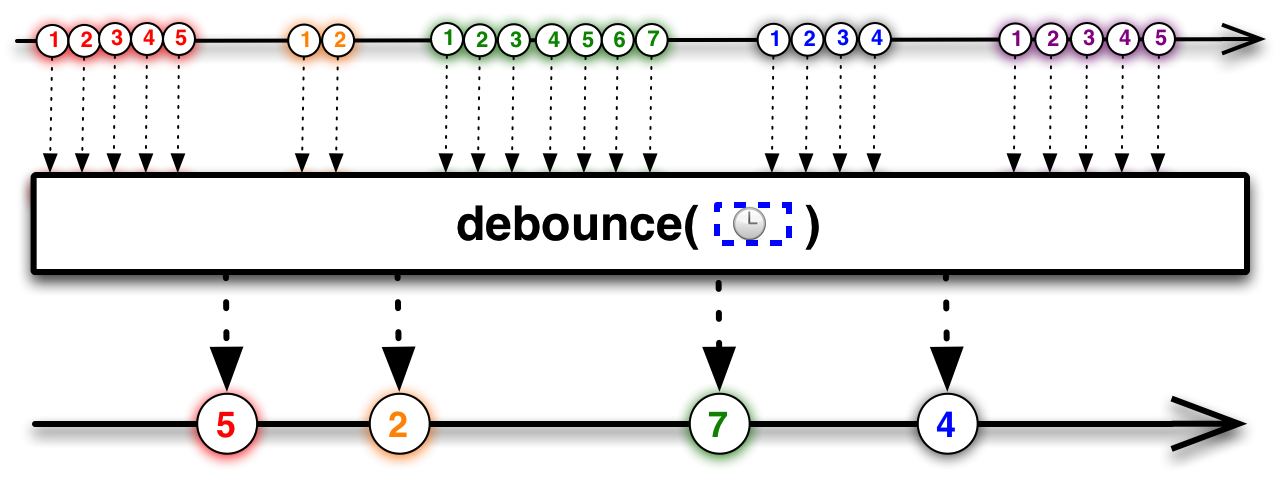
sample，采样。类比一下音频采样，8kHz的音频就是每125微秒采一个值。sample可以配置成，比如每100毫秒采样一个值，但100毫秒内上游可能过来很多值，选哪个值呢，就是选最后那个值。所以它也叫throttleLast。



throttleFirst跟sample类似，比如还是每100毫秒采样一个值，但选这100毫秒内的第一个值。在Android开发中有时候可以把throttleFirst用作点击事件的防抖动处理，就是因为它可以在指定的一段时间内处理第一个点击事件（即采样第一个值），但丢弃后面的点击事件。

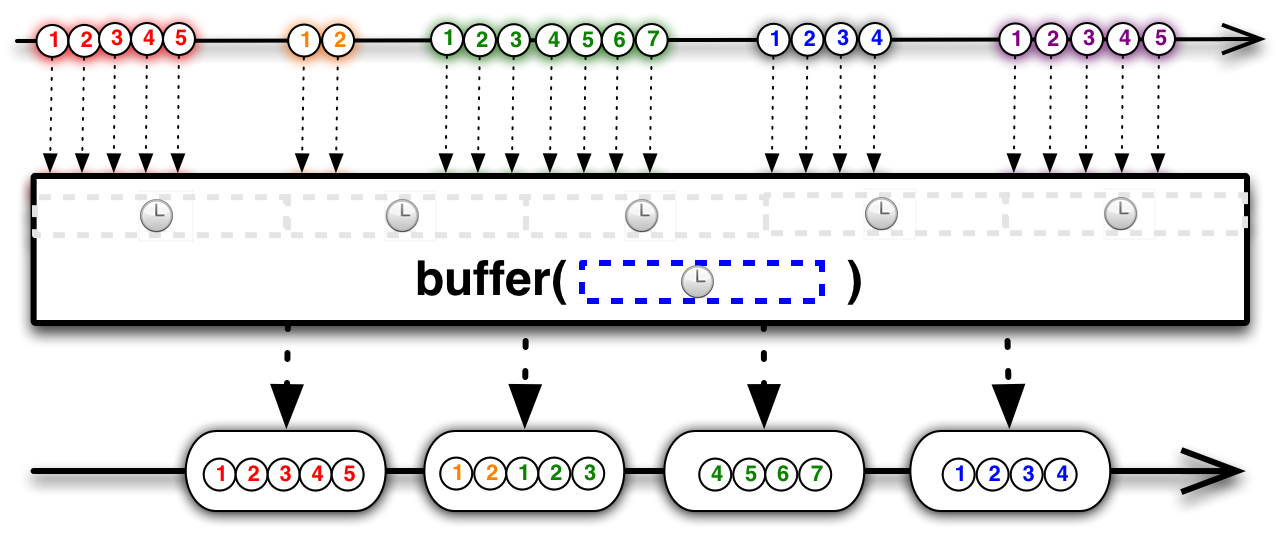


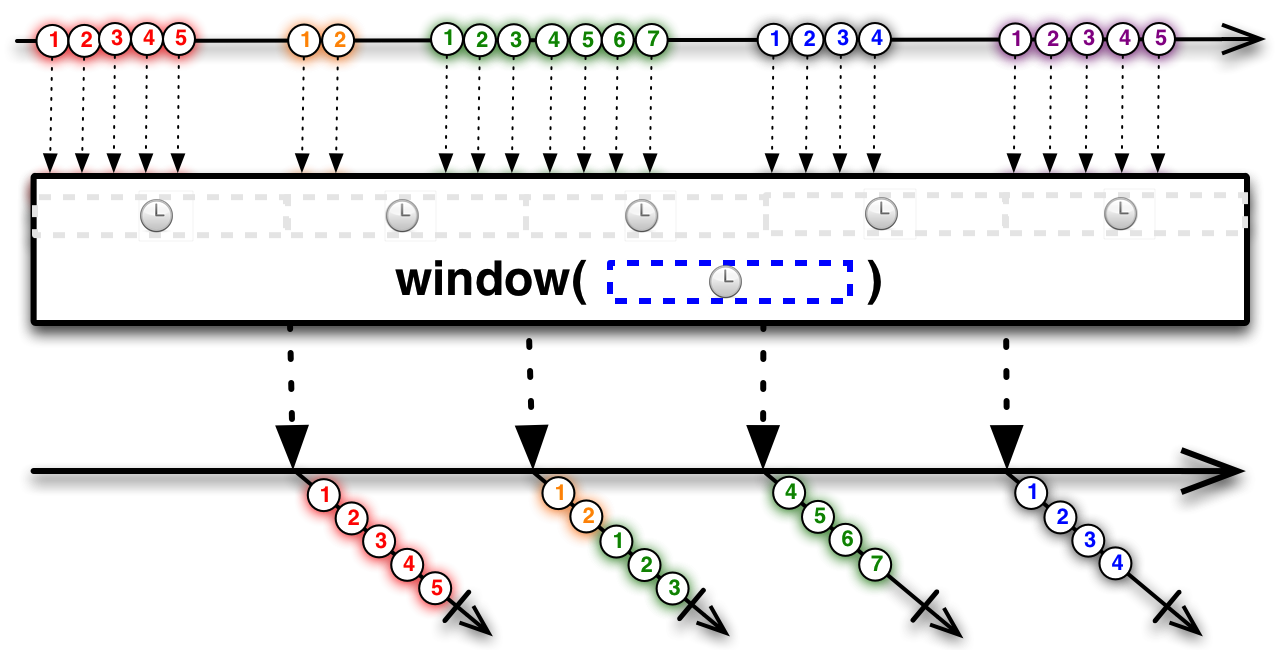
debounce，也叫throttleWithTimeout，名字里就包含一个例子。比如，一个网络程序维护一个TCP连接，不停地收发数据，但中间没数据可以收发的时候，就有间歇。这段间歇的时间，可以称为idle time。当idle time超过一个预设值的时候，就算超时了（time out），这个时候可能就需要把连接断开了。实际上一些做server端的网络程序就是这么工作的。每收发一个数据包之后，启动一个计时器，等待一个idle time。如果计时器到时之前，又有收发数据包的行为，那么计时器重置，等待一个新的idle time；而如果计时器时间到了，就超时了（time out），这个连接就可以关闭了。debounce的行为，跟这个非常类似，可以用它来找到那些连续的收发事件之后的idle time超时事件。换句话说，debounce可以把连续发生的事件之间的较大的间歇找出来。



**打包处理**

打包就是把上游来的小包裹打成大包裹，分发到下游。这样下游需要处理的包裹的个数就减少了。RxJava中提供了两类这样的机制：buffer和window。





buffer和window的功能基本一样，只是输出格式不太一样：buffer打包后的包裹用一个List表示，而window打包后的包裹又是一个Observable。

**调用栈阻塞（Callstack blocking）**

这是一种特殊情况，阻塞住整个调用栈（Callstack blocking）。之所以说这是一种特殊情况，是因为这种方式只适用于整个调用链都在一个线程上同步执行的情况，这要求中间的各个operator都不能启动新的线程。在平常使用中这种应该是比较少见的，因为我们经常使用subscribeOn或observeOn来切换执行线程，而且有些复杂的operator本身也会在内部启动新的线程来处理。另外，如果真的出现了完全同步的调用链，前面的另外三种Flow Control思路仍然可能是适用的，只不过这种阻塞的方式更简单，不需要额外的支持。

这里举个例子把调用栈阻塞和前面的Backpressure比较一下。“调用栈阻塞”相当于很多车行驶在盘山公路上，而公路只有一条车道。那么排在最前面的第一辆车就挡住了整条路，后面的车也只能排在后面。而“Backpressure”相当于银行办业务时的窗口叫号，窗口主动叫某个号过去（相当于请求），那个人才过去办理。

**如何让Observable支持Backpressure？**

在RxJava 1.x中，有些Observable是支持Backpressure的，而有些不支持。但不支持Backpressure的Observable可以通过一些operator来转化成支持Backpressure的Observable。这些operator包括：

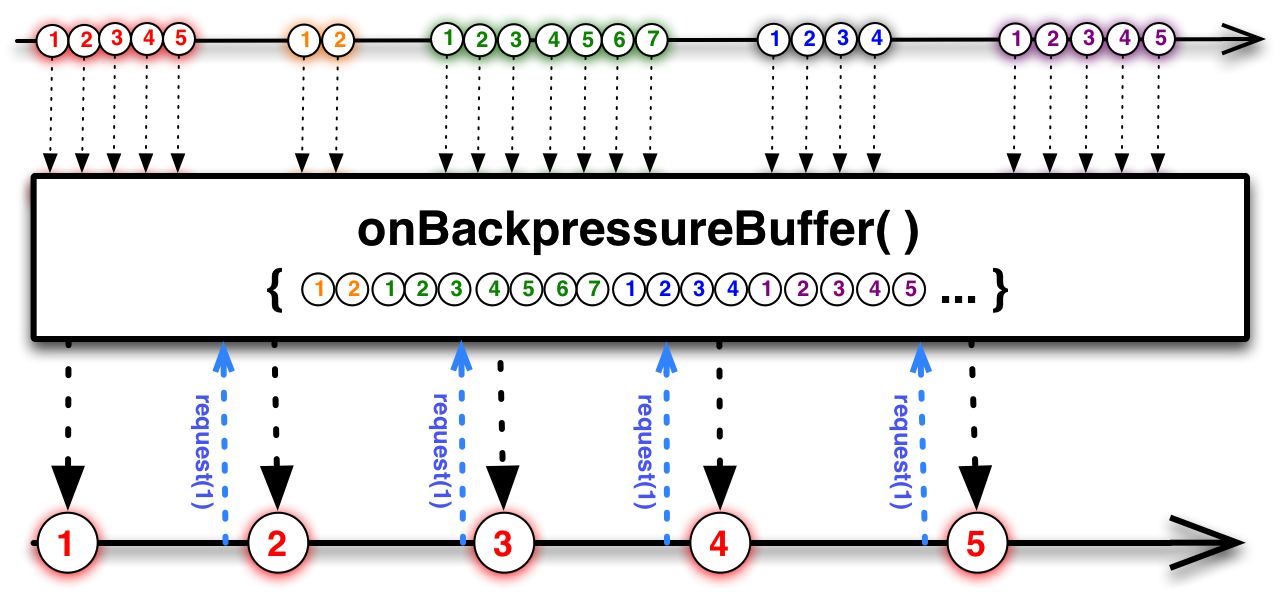
* onBackpressureBuffer
* onBackpressureDrop
* onBackpressureLatest
* onBackpressureBlock（已过期）

它们转化成的Observable分别具有不同的Backpressure策略。

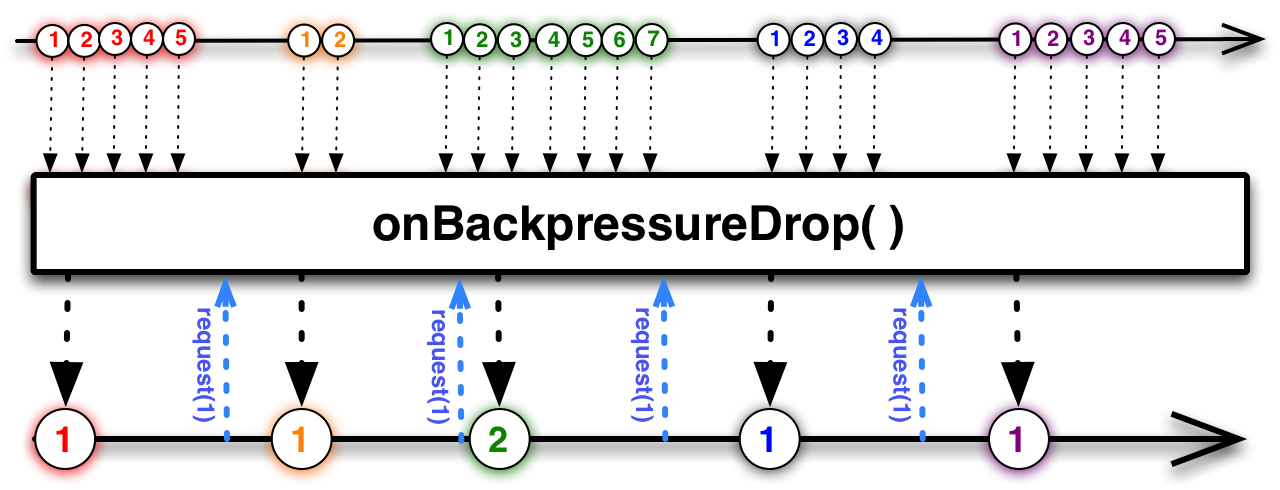
而在RxJava 2.x中，Observable不再支持Backpressure，而是改用Flowable来专门支持Backpressure。上面提到的四种operator的前三种分别对应Flowable的三种Backpressure策略：

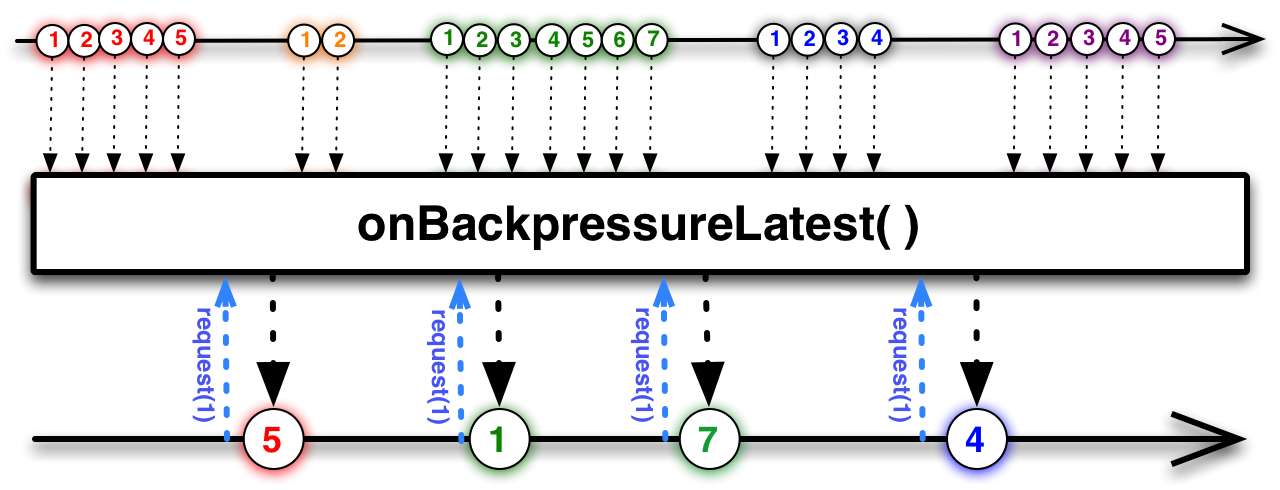
* BackpressureStrategy.BUFFER
* BackpressureStrategy.DROP
* BackpressureStrategy.LATEST

onBackpressureBuffer是不丢弃数据的处理方式。把上游收到的全部缓存下来，等下游来请求再发给下游。相当于一个水库。但上游太快，水库（buffer）就会溢出。



onBackpressureDrop和onBackpressureLatest比较类似，都会丢弃数据。这两种策略相当于一种令牌机制（或者配额机制），下游通过request请求产生令牌（配额）给上游，上游接到多少令牌，就给下游发送多少数据。当令牌数消耗到0的时候，上游开始丢弃数据。但这两种策略在令牌数为0的时候有一点微妙的区别：onBackpressureDrop直接丢弃数据，不缓存任何数据；而onBackpressureLatest则缓存最新的一条数据，这样当上游接到新令牌的时候，它就先把缓存的上一条“最新”数据发送给下游。可以结合下面两幅图来理解。





onBackpressureBlock是看下游有没有需求，有需求就发给下游，下游没有需求，不丢弃，但试图堵住上游的入口（能不能真堵得住还得看上游的情况了），自己并不缓存。这种策略已经废弃不用。

本文重点在于以宏观的角度来描述和对比RxJava中的Flow Control机制和Backpressure的各种机制，很多细节没有涉及。比如，buffer和window除了能把一段时间内收到的数据打包，还能把固定数量的数据进行打包。再比如，onBackpressureDrop和onBackpressureLatest在一次收到下游多条数据的请求时分别会如何表现，本文没有详细说明。大家可以查阅相应的API Reference来获得答案，也欢迎留言与我一起讨论。

（完）