Disruptor是如何工作的？（一）Ringbuffer的特别之处

原文地址：<http://ifeve.com/ringbuffer/>

**作者**：Trisha    **译者**：[寒桐](http://weibo.com/u/1837899012)  **校对：**方腾飞

最近，我们开源了[LMAX Disruptor](http://lmax-exchange.github.com/disruptor/)，它是我们的交易系统吞吐量快（LMAX是一个新型的交易平台，号称能够单线程每秒处理数百万的订单）的关键原因。为什么我们要将其开源？我们意识到对高性能编程领域的一些传统观点，有点不对劲。我们找到了一种更好、更快地在线程间共享数据的方法，如果不公开于业界共享的话，那未免太自私了。同时开源也让我们觉得看起来更酷。

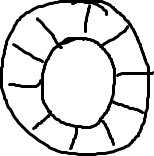
从这个[站点](http://lmax-exchange.github.com/disruptor/)，你可以下载到一篇解释什么是Disruptor及它为什么如此高性能的文档。这篇文档的编写过程，我并没有参与太多，只是简单地插入了一些标点符号和重组了一些我不懂的句子，但是非常高兴的是，我仍然从中提升了自己的写作水平。

我发现要把所有的事情一下子全部解释清楚还是有点困难的，所有我准备一部分一部分地解释它们，以适合我的[NADD](http://www.randsinrepose.com/archives/2003/07/10/nadd.html)听众。

首先介绍ringbuffer。我对Disruptor的最初印象就是ringbuffer。但是后来我意识到尽管ringbuffer是整个模式（Disruptor）的核心，但是Disruptor对ringbuffer的访问控制策略才是真正的关键点所在。

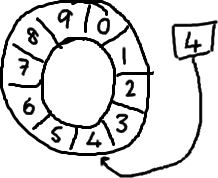
**ringbuffer到底是什么？**

嗯，正如名字所说的一样，它是一个环（首尾相接的环），你可以把它用做在不同上下文（线程）间传递数据的buffer。

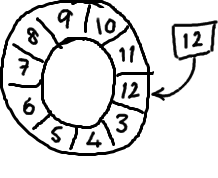
[](http://1.bp.blogspot.com/-3gtuTbWgI-Y/TgD92AhOVxI/AAAAAAAAIDE/cWsAT81B0AI/s1600/RingBuffer.png)

（好吧，这是我通过画图板手画的，我试着画草图，希望我的强迫症不会让我画完美的圆和直线）

基本来说，ringbuffer拥有一个序号，这个序号指向数组中下一个可用的元素。*（校对注：如下图右边的图片表示序号，这个序号指向数组的索引4的位置。）*

[](http://4.bp.blogspot.com/-WCm0iWYOrfc/TgD_cVBQExI/AAAAAAAAIDI/cEohveH8LP8/s1600/RingBufferInitial.png)

随着你不停地填充这个buffer（可能也会有相应的读取），这个序号会一直增长，直到绕过这个环。

[](http://ifeve.com/wp-content/uploads/2013/01/RingBufferWrapped.png)

要找到数组中当前序号指向的元素，可以通过mod操作：

               sequence mod array length = array index

以上面的ringbuffer为例（java的mod语法）：12 % 10 = 2。很简单吧。

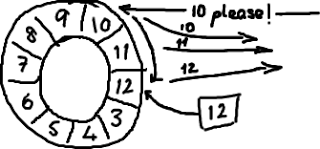
事实上，上图中的ringbuffer只有10个槽完全是个意外。如果槽的个数是2的N次方更有利于基于二进制的计算机进行计算。

*（校对注：2的N次方换成二进制就是1000，100，10，1这样的数字， sequence & （array length－1） = array index，比如一共有8槽，3&（8－1）=3，HashMap就是用这个方式来定位数组元素的，这种方式比取模的速度更快。）*

**那又怎么样？**

如果你看了维基百科里面的关于[环形buffer](http://en.wikipedia.org/wiki/Circular_buffer)的词条，你就会发现，我们的实现方式，与其最大的区别在于：没有尾指针。我们只维护了一个指向下一个可用位置的序号。这种实现是经过深思熟虑的—我们选择用环形buffer的最初原因就是想要提供可靠的消息传递。我们需要将已经被服务发送过的消息保存起来，这样当另外一个服务通过[nak](http://en.wikipedia.org/wiki/NaK) (校对注：拒绝应答信号)告诉我们没有成功收到消息时，我们能够重新发送给他们。

听起来，环形buffer非常适合这个场景。它维护了一个指向尾部的序号，当收到nak(校对注：拒绝应答信号)请求，可以重发从那一点到当前序号之间的所有消息：

[](http://ifeve.com/wp-content/uploads/2013/01/RingBufferReplay.png)

我们实现的ring buffer和大家常用的队列之间的区别是，我们不删除buffer中的数据，也就是说这些数据一直存放在buffer中，直到新的数据覆盖他们。这就是和维基百科版本相比，我们不需要尾指针的原因。ringbuffer本身并不控制是否需要重叠（决定是否重叠是生产者-消费者行为模式的一部分–如果你等不急我写blog来说明它们，那么可以自行检出[Disruptor项目](https://code.google.com/p/disruptor/)）。

**它为什么如此优秀？**

之所以ringbuffer采用这种数据结构，是因为它在可靠消息传递方面有很好的性能。这就够了，不过它还有一些其他的优点。

首先，因为它是数组，所以要比链表快，而且有一个容易预测的访问模式。（*译者注：数组内元素的内存地址的连续性存储的*）。这是对CPU缓存友好的—也就是说，在硬件级别，数组中的元素是会被预加载的，因此在ringbuffer当中，cpu无需时不时去主存加载数组中的下一个元素。*（校对注：因为只要一个元素被加载到缓存行，其他相邻的几个元素也会被加载进同一个缓存行）*

其次，你可以为数组预先分配内存，使得数组对象一直存在（除非程序终止）。这就意味着不需要花大量的时间用于垃圾回收。此外，不像链表那样，需要为每一个添加到其上面的对象创造节点对象—对应的，当删除节点时，需要执行相应的内存清理操作。

**缺少的部分**

我并没有在本文中介绍如何避免ringbuffer产生重叠，以及如何对ringbuffer进行读写操作。你可能注意到了我将ringbuffer和链表那样的数据结构进行比较，因为我并认为链表是实际问题的标准答案。

当你将Disruptor和基于 队列之类的实现进行比较时，事情将变得很有趣。队列通常注重维护队列的头尾元素，添加和删除元素等。所有的这些我都没有在ringbuffer里提到，这是因为ringbuffer不负责这些事情，我们把这些操作都移到了数据结构（ringbuffer）的外部

到这个[站点](https://code.google.com/p/disruptor/)阅读文章或者检出代码可以了解更多细节。或者观看Mike 和Martin在去年[San Francisco QCon大会上的视频](http://www.infoq.com/presentations/LMAX)，或者再等我一些时间来思考剩下的东西，然后在接下来的blog中逐一介绍。

Disruptor是如何工作的？（二）如何从Ringbuffer读取

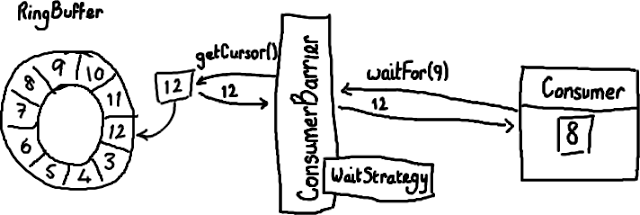
英文原文：<http://ifeve.com/dissecting-the-disruptor-how-do-i-read-from-the-ring-buffer/>

**作者**：Trisha **译者**：古圣昌  **校对**：方腾飞

从[上一篇文章](http://ifeve.com/dissecting-disruptor-whats-so-special/)中我们都了解了什么是***Ring Buffer***以及它是如何的特别。但遗憾的是，我还没有讲述如何使用***Disruptor***向***Ring Buffer***写数据和从***Ring Buffer***中读取数据。

**ConsumerBarrier与消费者**

这里我要稍微反过来介绍，因为总的来说读取数据这一过程比写数据要容易理解。假设通过一些“魔法”已经把数据写入到***Ring Buffer***了，怎样从***Ring Buffer***读出这些数据呢？

[](http://ifeve.com/wp-content/uploads/2013/02/Image1.png)

(好，我开始后悔使用***Paint/Gimp*** 了。尽管这是个购买绘图板的好借口，如果我继续写下去的话… ***UML***界的权威们大概也在诅咒我的名字了。)

消费者(***Consumer***)是一个想从***Ring Buffer***里读取数据的线程，它可以访问***ConsumerBarrier***对象——这个对象由***RingBuffer***创建并且代表消费者与***RingBuffer***进行交互。就像***Ring Buffer***显然需要一个序号才能找到下一个可用节点一样，消费者也需要知道它将要处理的序号——每个消费者都需要找到下一个它要访问的序号。在上面的例子中，消费者处理完了***Ring Buffer***里序号***8***之前（包括***8***）的所有数据，那么它期待访问的下一个序号是***9***。

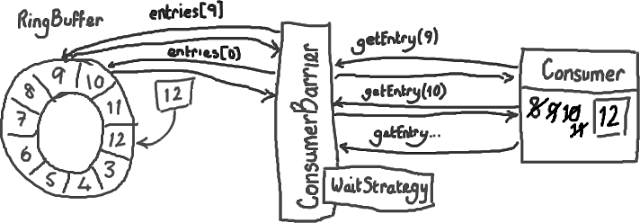
消费者可以调用***ConsumerBarrier***对象的***waitFor()***方法，传递它所需要的下一个序号.

final long availableSeq **=** consumerBarrier**.**waitFor**(**nextSequence**);**

***ConsumerBarrier***返回***RingBuffer***的最大可访问序号——在上面的例子中是***12***。***ConsumerBarrier***有一个***WaitStrategy***方法来决定它如何等待这个序号，我现在不会去描述它的细节，代码的注释里已经概括了每一种***WaitStrategy***的优点和缺点 。

**接下来怎么做？**

接下来，消费者会一直原地停留，等待更多数据被写入***Ring Buffer***。并且，一旦数据写入后消费者会收到通知——节点***9***，***10***，***11***和***12*** 已写入。现在序号***12***到了，消费者可以让***ConsumerBarrier***去拿这些序号节点里的数据了。

[](http://ifeve.com/wp-content/uploads/2013/02/Image21.png)

拿到了数据后，消费者(***Consumer***)会更新自己的标识(***cursor***)。

你应该已经感觉得到，这样做是怎样有助于平缓延迟的峰值了——以前需要逐个节点地询问“我可以拿下一个数据吗？现在可以了么？现在呢？”，消费者(***Consumer***)现在只需要简单的说“当你拿到的数字比我这个要大的时候请告诉我”，函数返回值会告诉它有多少个新的节点可以读取数据了。因为这些新的节点的确已经写入了数据（***Ring Buffer***本身的序号已经更新），而且消费者对这些节点的唯一操作是读而不是写，因此访问不用加锁。这太好了，不仅代码实现起来可以更加安全和简单，而且不用加锁使得速度更快。

另一个好处是——你可以用多个消费者(***Consumer)***去读同一个***RingBuffer*** ，不需要加锁，也不需要用另外的队列来协调不同的线程(消费者)。这样你可以在***Disruptor***的协调下实现真正的并发数据处理。

***BatchConsumer***代码是一个消费者的例子。如果你实现了***BatchHandler,***你可以用***BatchConsumer***来完成上面我提到的复杂工作。它很容易对付那些需要成批处理的节点（例如上文中要处理的***9-12***节点）而不用单独地去读取每一个节点。

更新：注意***Disruptor 2.0***版本使用了与本文不一样的命名。如果你对类名感到困惑，请阅读我的[**变更总结**](http://ifeve.com/disruptor-2-change/)。

Disruptor是如何工作的？（三）写入 Ringbuffer

原文地址：<http://ifeve.com/dissecting-the-disruptor-writing-to-the-ring-buffer/>

**作者**：Trisha   **译者**：廖涵  **校对**：方腾飞

这是 Disruptor 全方位解析（end-to-end view）中缺少的一章。当心，本文非常长。但是为了让你能联系上下文阅读，我还是决定把它们写进一篇博客里。

本文的 **重点**是：不要让 Ring 重叠；如何通知消费者；生产者一端的批处理；以及多个生产者如何协同工作。

**ProducerBarriers**

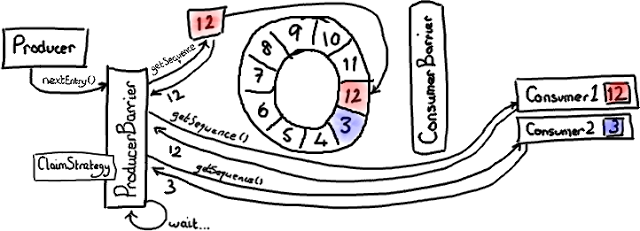
[Disruptor 代码](http://code.google.com/p/disruptor/)​ 给 **消费者** 提供了一些接口和辅助类，但是没有给写入 Ring Buffer 的 **生产者**提供接口。这是因为除了你需要知道生产者之外，没有别人需要访问它。尽管如此，Ring Buffer 还是与消费端一样提供了一个 ProducerBarrier 对象，让生产者通过它来写入 Ring Buffer。

写入 Ring Buffer 的过程涉及到两阶段提交 (two-phase commit)。首先，你的生产者需要申请 buffer 里的下一个节点。然后，当生产者向节点写完数据，它将会调用 ProducerBarrier 的 commit 方法。

那么让我们首先来看看第一步。 “给我 Ring Buffer 里的下一个节点”，这句话听起来很简单。的确，从生产者角度来看它很简单：简单地调用 ProducerBarrier 的 nextEntry() 方法，这样会返回给你一个 Entry 对象，这个对象就是 Ring Buffer 的下一个节点。

**ProducerBarrier 如何防止 Ring Buffer 重叠**

在后台，由 ProducerBarrier 负责所有的交互细节来从 Ring Buffer 中找到下一个节点，然后才允许生产者向它写入数据。



（我不确定 [闪闪发亮的新手写板](http://www.amazon.com/Wacom-CTL460-Bamboo-Pen-Tablet/dp/B002OOWC3I?ie=UTF8&tag=trissramb-20&link_code=btl&camp=213689&creative=392969)​ 能否有助于提高我画图片的清晰度，但是它用起来很有意思）。

在这幅图中，我们假设只有一个生产者写入 Ring Buffer。过一会儿我们再处理多个生产者的复杂问题。

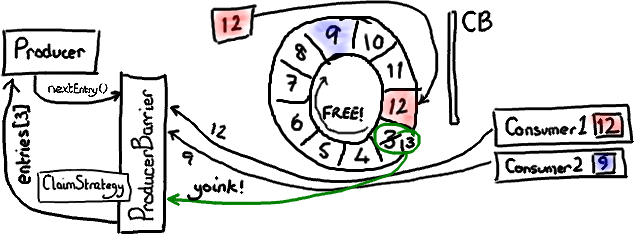
**ConsumerTrackingProducerBarrier** 对象拥有所有正在访问 Ring Buffer 的 **消费者**列表。这看起来有点儿奇怪－我从没有期望 ProducerBarrier 了解任何有关消费端那边的事情。但是等等，这是有原因的。因为我们不想与队列“混为一谈”（队列需要追踪队列的头和尾，它们有时候会指向相同的位置），Disruptor 由消费者负责通知它们处理到了哪个序列号，而不是 Ring Buffer。所以，如果我们想确定我们没有让 Ring Buffer 重叠，需要检查所有的消费者们都读到了哪里。

在上图中，有一个 **消费者** 顺利的读到了最大序号 12（用红色/粉色高亮）。第二个**消费者** 有点儿落后——可能它在做 I/O 操作之类的——它停在序号 3。因此消费者 2 在赶上消费者 1 之前要跑完整个 Ring Buffer 一圈的距离。

现在生产者想要写入 Ring Buffer 中序号 3 占据的节点，因为它是 Ring Buffer 当前游标的下一个节点。但是 ProducerBarrier 明白现在不能写入，因为有一个消费者正在占用它。所以，ProducerBarrier 停下来自旋 (spins)，等待，直到那个消费者离开。

**申请下一个节点**

现在可以想像消费者 2 已经处理完了一批节点，并且向前移动了它的序号。可能它挪到了序号 9（因为消费端的批处理方式，现实中我会预计它到达 12，但那样的话这个例子就不够有趣了）。

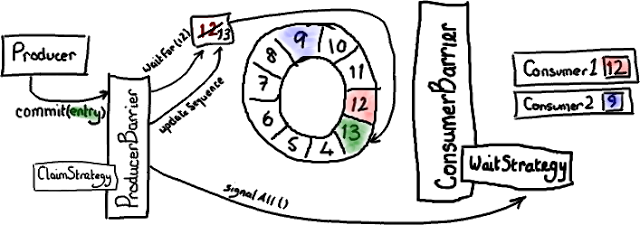


上图显示了当消费者 2 挪动到序号 9 时发生的情况。在这张图中我已经忽略了ConsumerBarrier，因为它没有参与这个场景。

ProducerBarier 会看到下一个节点——序号 3 那个已经可以用了。它会抢占这个节点上的 Entry（我还没有特别介绍 Entry 对象，基本上它是一个放写入到某个序号的 Ring Buffer 数据的桶），把下一个序号（13）更新成 Entry 的序号，然后把 Entry 返回给生产者。生产者可以接着往 Entry 里写入数据。

**提交新的数据**

两阶段提交的第二步是——对，提交。



绿色表示最近写入的 Entry，序号是 13 ——厄，抱歉，我也是红绿色盲。但是其他颜色甚至更糟糕。

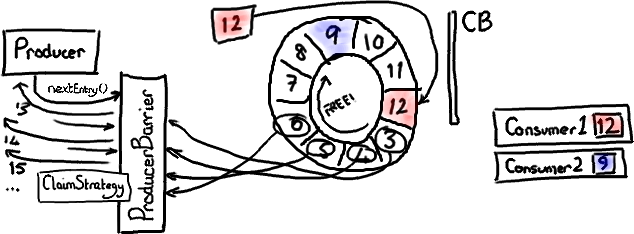
当生产者结束向 Entry 写入数据后，它会要求 ProducerBarrier 提交。

ProducerBarrier 先等待 Ring Buffer 的游标追上当前的位置（对于单生产者这毫无意义－比如，我们已经知道游标到了 12 ，而且没有其他人正在写入 Ring Buffer）。然后 ProducerBarrier 更新 Ring Buffer 的游标到刚才写入的 Entry 序号－在我们这儿是 13。接下来，ProducerBarrier 会让消费者知道 buffer 中有新东西了。它戳一下 ConsumerBarrier 上的 WaitStrategy 对象说－“喂，醒醒！有事情发生了！”（注意－不同的 WaitStrategy 实现以不同的方式来实现提醒，取决于它是否采用阻塞模式。）

现在消费者 1 可以读 Entry 13 的数据，消费者 2 可以读 Entry 13 以及前面的所有数据，然后它们都过得很 happy。

**ProducerBarrier 上的批处理**

有趣的是 Disruptor 可以同时在生产者和 [消费者](http://mechanitis.blogspot.com/2011/06/dissecting-disruptor-how-do-i-read-from.html)​ 两端实现批处理。还记得伴随着程序运行，消费者 2 最后达到了序号 9 吗？ProducerBarrier 可以在这里做一件很狡猾的事－它知道 Ring Buffer 的大小，也知道最慢的消费者位置。因此它能够发现当前有哪些节点是可用的。



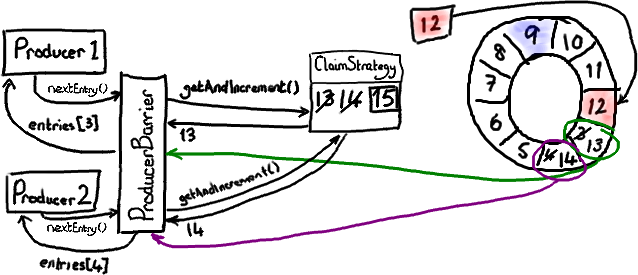
如果 ProducerBarrier 知道 Ring Buffer 的游标指向 12，而最慢的消费者在 9 的位置，它就可以让生产者写入节点 3，4，5，6，7 和 8，中间不需要再次检查消费者的位置。

**多个生产者的场景**

到这里你也许会以为我讲完了，但其实还有一些细节。

在上面的图中我稍微撒了个谎。我暗示了 ProducerBarrier 拿到的序号直接来自 Ring Buffer 的游标。然而，如果你看过代码的话，你会发现它是通过 ClaimStrategy 获取的。我省略这个对象是为了简化示意图，在单个生产者的情况下它不是很重要。

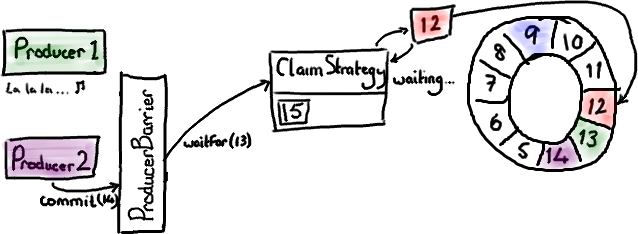
在多个生产者的场景下，你还需要其他东西来追踪序号。这个序号是指当前可写入的序号。注意这和“向 Ring Buffer 的游标加 1”不一样－如果你有一个以上的生产者同时在向 Ring Buffer 写入，就有可能出现某些 Entry 正在被生产者写入但还没有提交的情况。

[](http://ifeve.com/wp-content/uploads/2013/01/ProducersNextEntry.png)

让我们复习一下如何申请写入节点。每个生产者都向 ClaimStrategy 申请下一个可用的节点。生产者 1 拿到序号 13，这和上面单个生产者的情况一样。生产者 2 拿到序号 14，尽管 Ring Buffer的当前游标仅仅指向 12。这是因为 ClaimSequence 不但负责分发序号，而且负责跟踪哪些序号已经被分配。

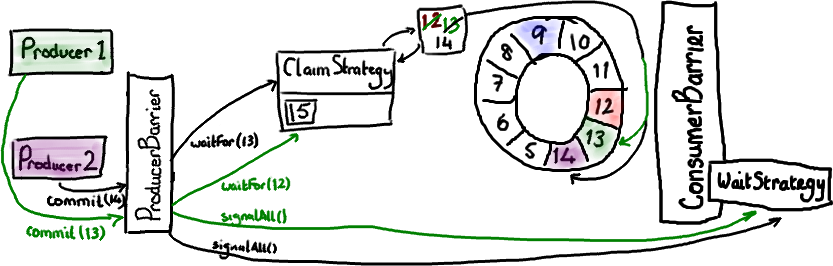
现在每个生产者都拥有自己的写入节点和一个崭新的序号。

我把生产者 1 和它的写入节点涂上绿色，把生产者 2 和它的写入节点涂上可疑的粉色－看起来像紫色。

[](http://ifeve.com/wp-content/uploads/2013/01/ProducersCommit.png)

现在假设生产者 1 还生活在童话里，因为某些原因没有来得及提交数据。生产者 2 已经准备好提交了，并且向 ProducerBarrier 发出了请求。

就像我们先前在 commit 示意图中看到的一样，ProducerBarrier 只有在 Ring Buffer 游标到达准备提交的节点的前一个节点时它才会提交。在当前情况下，游标必须先到达序号 13 我们才能提交节点 14 的数据。但是我们不能这样做，因为生产者 1 正盯着一些闪闪发光的东西，还没来得及提交。因此 ClaimStrategy 就停在那儿自旋 (spins)， 直到 Ring Buffer 游标到达它应该在的位置。

[](http://ifeve.com/wp-content/uploads/2013/02/ProducersCommit2.png)

现在生产者 1 从迷糊中清醒过来并且申请提交节点 13 的数据（生产者 1 发出的绿色箭头代表这个请求）。ProducerBarrier 让 ClaimStrategy 先等待 Ring Buffer 的游标到达序号 12，当然现在已经到了。因此 Ring Buffer 移动游标到 13，让 ProducerBarrier 戳一下 WaitStrategy 告诉所有人都知道 Ring Buffer 有更新了。现在 ProducerBarrier 可以完成生产者 2 的请求，让 Ring Buffer 移动游标到 14，并且通知所有人都知道。

你会看到，尽管生产者在不同的时间完成数据写入，但是 Ring Buffer 的内容顺序总是会遵循 nextEntry() 的初始调用顺序。也就是说，如果一个生产者在写入 Ring Buffer 的时候暂停了，只有当它解除暂停后，其他等待中的提交才会立即执行。

呼——。我终于设法讲完了这一切的内容并且一次也没有提到内存屏障（Memory Barrier）。

**更新**：最近的 [RingBuffer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java?r=239)​ 版本去掉了 Producer Barrier。如果在你看的代码里找不到 ProducerBarrier，那就假设当我讲“Producer Barrier”时，我的意思是“Ring Buffer”。

**更新2**：注意 Disruptor 2.0 版使用了与本文不一样的命名。如果你对类名感到困惑，请阅读我写的[Disruptor 2.0更新摘要](http://ifeve.com/disruptor-2-change/)​。

Disruptor是如何工作的？（四）解析Disruptor的依赖关系

原文地址：<http://ifeve.com/dissecting-disruptor-wiring-up/>

**作者**：Trisha   **译者**：廖涵  **校对**：方腾飞

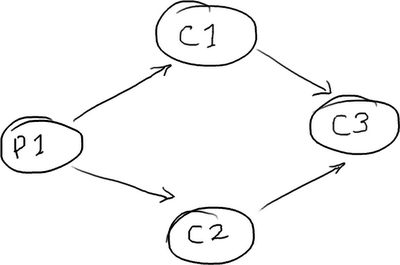
现在我已经讲了 [RingBuffer](http://ifeve.com/dissecting-disruptor-whats-so-special/)​ 本身，如何从它 [读取](http://ifeve.com/dissecting_the_disruptor_how_doi_read_from_the_ring_buffer/)​ 以及如何向它 [写入](http://ifeve.com/disruptor-writing-ringbuffer/)​。从逻辑上来说，下一件要做的事情就是把所有的东西拼装到在一起。

我前面提到过多生产者的情况——他们通过 ProducerBarrier 保证写入操作顺序与可控。我也提到过简单场景下的多消费者数据访问。更多的消费者的场景会变得更加复杂，[我们](http://www.lmaxtrader.co.uk/)​ 实现了一些聪明的机制允许多个消费者在访问 Ring Buffer 的时候互相等待（依赖）。像很多应用里，有一连串的工作需要在实际执行业务逻辑之前完成 (happen before) —— 例如，在做任何操作之前，我们都必须先保证消息写入磁盘。

[Disruptor 论文](http://disruptor.googlecode.com/files/Disruptor-1.0.pdf)​ 和性能测试里包含了你可能想到的一些基本结构。我准备讲一下其中最有趣的那个，这多半是因为我需要练习如何使用画图板。

**菱形结构**

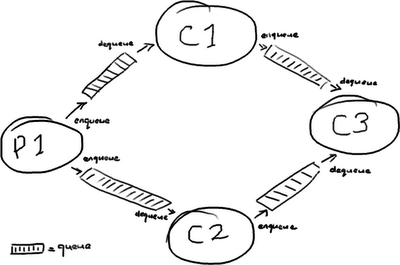
[DiamondPath1P3CPerfTest](http://code.google.com/p/disruptor/source/browse/trunk/code/src/perf/com/lmax/disruptor/DiamondPath1P3CPerfTest.java)​ 展示了一个并不罕见的结构——独立的一个生产者和三个消费者。最棘手的一点是：第三个消费者必须等待前两个消费者处理完成后，才能开始工作。



消费者 C3 也许是你的业务逻辑。消费者 C1 可能在备份接收到的数据，而消费者 C2 可能在准备数据或者别的东西。

**用队列实现菱形结构**

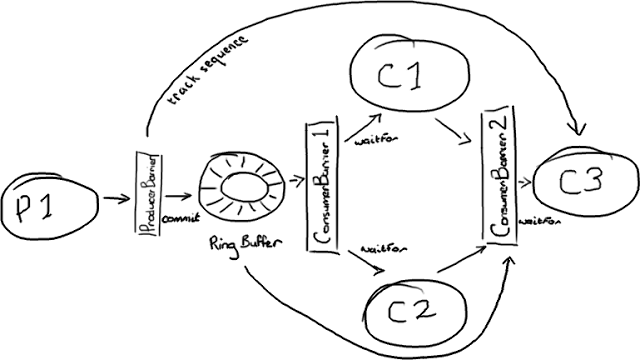
在一个 [SEDA-风格的架构](http://www.theserverside.com/news/1363672/Building-a-Scalable-Enterprise-Applications-Using-Asynchronous-IO-and-SEDA-Model)​ 中，每个处理阶段都会用队列分开：

  
（为什么单词 Queue 里必须有这么多 “e” 呢？这是我在画这些图时遇到的最麻烦的词）。

你也许从这里看到了问题的端倪：一条消息从 P1 传输到 C3 要完整的穿过四个队列，每个队列在消息进入队列和取出队列时都会产生消耗成本。

**用 Disruptor 实现菱形结构**

在 [Disruptor](http://code.google.com/p/disruptor/)​ 的世界里，一切都由一个单独的 Ring Buffer 管理：



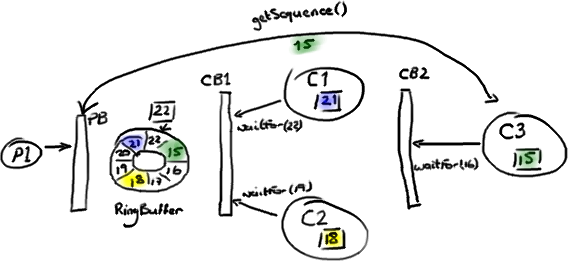
这张图看起来更复杂。不过所有的参与者都只依赖 Ring Buffer 作为一个单独的联结点，而且所有的交互都是基于 Barrier 对象与检查依赖的目标序号来实现的。

生产者这边比较简单，它是我在 [上文](http://ifeve.com/dissecting-the-disruptor-writing-to-the-ring-buffer/)​ 中描述过的单生产者模型。有趣的是，生产者并不需要关心所有的消费者。它只关心消费者 C3，如果消费者 C3 处理完了 Ring Buffer 的某一个节点，那么另外两个消费者肯定也处理完了。因此，只要 C3 的位置向前移动，Ring Buffer 的后续节点就会空闲出来。

管理消费者的依赖关系需要两个 ConsumerBarrier 对象。第一个仅仅与 Ring Buffer 交互，C1 和 C2 消费者向它申请下一个可访问节点。第二个 ConsumerBarrier 只知道消费者 C1 和 C2，它返回两个消费者访问过的消息序号中较小的那个。

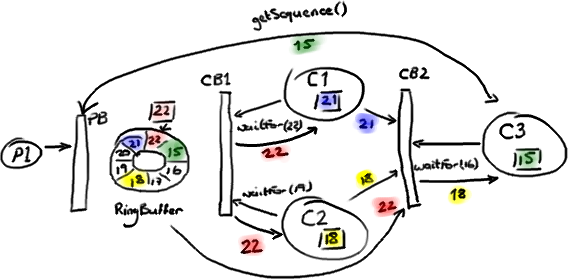
**Disruptor 怎样实现消费者等待（依赖）**

Hmmm。我想需要一个例子。



我们从这个故事发生到一半的时候来看：生产者 P1 已经在 Ring Buffer 里写到序号 22 了，消费者 C1 已经访问和处理完了序号 21 之前的所有数据。消费者 C2 处理到了序号 18。消费者 C3，就是依赖其他消费者的那个，才处理到序号 15。

生产者 P1 不能继续向 RingBuffer 写入数据了，因为序号 15 占据了我们想要写入序号 23 的数据节点 (Slot)。

  
（抱歉，我真的试过用其他颜色来代替红色和绿色，但是别的都更容易混淆。）

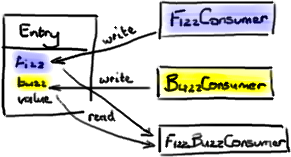
第一个 ConsumerBarrier（CB1）告诉 C1 和 C2 消费者可以去访问序号 22 前面的所有数据，这是 Ring Buffer 中的最大序号。第二个 ConsumerBarrier (CB2) 不但会检查 RingBuffer 的序号，也会检查另外两个消费者的序号并且返回它们之间的最小值。因此，三号消费者被告知可以访问 Ring Buffer 里序号 18 前面的数据。

注意这些消费者还是直接从 Ring Buffer 拿数据节点——并不是由 C1 和 C2 消费者把数据节点从 Ring Buffer 里取出再传递给 C3 消费者的。作为替代的是，由第二个 ConsumerBarrier 告诉 C3 消费者，在 RingBuffer 里的哪些节点可以安全的处理。

这产生了一个问题——如果任何数据都来自于 Ring Buffer，那么 C3 消费者如何读到前面两个消费者处理完成的数据呢？如果 C3 消费者关心的只是先前的消费者是否已经完成它们的工作（例如，把数据复制到别的地方），那么这一切都没有问题—— C3 消费者知道工作已完成就放心了。但是，如果 C3 消费者需要访问先前的消费者的处理结果，它又从哪里去获取呢？

**更新数据节点**

秘密在于把处理结果写入 Ring Buffer 数据节点 (Entry) 本身。这样，当 C3 消费者从 Ring Buffer 取出节点时，它已经填充好了 C3 消费者工作需要的所有信息。这里 **真正** 重要的地方是节点 (Entry) 对象的每一个字段应该只允许一个消费者写入。这可以避免产生并发写入冲突 (write-contention) 减慢了整个处理过程。



你可以在 [DiamondPath1P3CPerfTest](http://code.google.com/p/disruptor/source/browse/trunk/code/src/perf/com/lmax/disruptor/DiamondPath1P3CPerfTest.java)​ 里看到这个例子—— [FizzBuzzEntry](http://code.google.com/p/disruptor/source/browse/trunk/code/src/perf/com/lmax/disruptor/support/FizzBuzzEntry.java)​ 有两个字段：fizz 和 buzz。如果消费者是 Fizz Consumer, 它只写入字段 fizz。如果是 Buzz Consumer, 它只写入字段 buzz。第三个消费者 FizzBuzz，它只去读这两个字段但是不会做写入，因为读没问题，不会引起争用。

**一些实际的 Java 代码**

这一切看起来都要比队列实现更复杂。是的，它涉及到更多的内部协调。但是这些细节对于消费者和生产者是隐藏的，它们只和 Barrier 对象交互。诀窍在消费者结构里。上文例子中提到的菱形结构可以用下面的方法创建：

ConsumerBarrier consumerBarrier1 **=**

ringBuffer**.**createConsumerBarrier**();**

BatchConsumer consumer1 **=**

**new** BatchConsumer**(**consumerBarrier1**,** handler1**);**

BatchConsumer consumer2 **=**

**new** BatchConsumer**(**consumerBarrier1**,** handler2**);**

ConsumerBarrier consumerBarrier2 **=**

ringBuffer**.**createConsumerBarrier**(**consumer1**,** consumer2**);**

BatchConsumer consumer3 **=**

**new** BatchConsumer**(**consumerBarrier2**,** handler3**);**

ProducerBarrier producerBarrier **=**

ringBuffer**.**createProducerBarrier**(**consumer3**);**

**总结**

现在你知道了——如何关联 Disruptor 与相互依赖（等待）的多个消费者。关键点是：

* 使用多个 ConsumerBarrier 来管理消费者之间的依赖（等待）关系。
* 使用 ProducerBarrier 监视结构图中最后一个消费者。
* 只允许一个消费者更新数据节点 (Entry) 的每一个独立字段。

更新：Adrian 写了一个非常好的 [DSL](http://www.symphonious.net/2011/07/11/lmax-disruptor-high-performance-low-latency-and-simple-too/) 工具让拼接 Disruptor 更加简单了。

更新 2：注意 Disruptor 2.0 版使用了与本文不一样的命名。如果你对类名感到困惑，请阅读我的 [变更总结](http://ifeve.com/disruptor-2-change/)​​。另外，Adrian 的 DSL 工具现在是 Disruptor 主干代码的一部分了。