剖析Disruptor:为什么会这么快？（一）锁的缺点

原文：<http://ifeve.com/disruptor-locks-are-bad/>

作者：Trisha’s  译者：张文灼，潘曦  整理和校对：方腾飞，丁一

Martin Fowler写了一篇[非常好的文章](http://ifeve.com/lmax/)，里面不仅提到了[Disruptor](http://code.google.com/p/disruptor/),而且还解释了Disruptor 如何应用在[LMAX](http://www.lmax.com/)的架构里。里面有提及了一些目前没有涉及的概念，但最经常问到的问题是 “Disruptor究竟是什么?＂。

目前我正准备在回答这个问题，但首先回答＂为什么它会这么快?＂

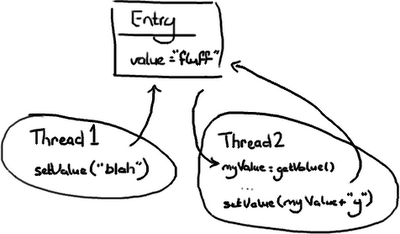
这些问题持续出现，但是我不能没有说它是干什么的就说它为什么会这么快，不能没有说它为什么这样做就说它是什么东西。

所以我陷入了一个僵局，一个如何写博客的僵局。

要打破这个僵局，我准备以最简单方式回答第一个问题，如果幸运的话，在以后博文里，如果需要解释的话我会重新提回：Disruptor提供了一种线程之间信息交换的方式。

作为一个开发者,因为＂线程＂一词的出现，我的警钟已经敲响，它意味着并发，而并发是困难的。

并发 01

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

想象有两个线程尝试修改同一个变量value：

**情况一：线程１先到达**

变量value的值变为”blah”。

然后当线程２到达时，变量value的值变为”blahy”。

**情况二：线程２先到达**

变量value的值变为”fluffy”。

然后当线程１到达时，值变为”blah”。

**情况三：线程１与线程２交互**

线程２得到值＂fluff＂然后赋给本地变量myValue。

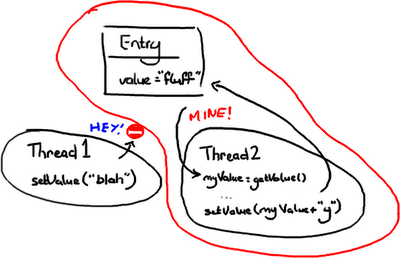
线程１改变value的值为”blah”。

然后线程２醒来并把变量value的值改为”fluffy”

情况三显然是唯一一个是错误的，除非除非你认为wiki编辑的幼稚做法是正确的（[Google Code](http://code.google.com/) Wiki，我一直在关注你）。其他两种情况主要是看你的意图和想要达到的效果。线程２可能不会关心变量value的值是什么，主要的意图就是在后面加上字符 ‘y＇而不管它原来的值是什么，在这种前提下，情况一和情况二都是正确的。

但是如果线程２只是想把＂fluff＂改为”fluffy”，那么情况二和三都不正确。假定线程２想把值设为”fluffy”,有几种办法可以解决这个问题：

**办法一：悲观锁**



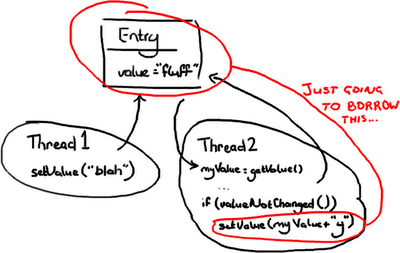
(“No Entry”的标志对于在没有在英国开车的人看得明白不？)

悲观锁和乐观锁这两个词通常在我们谈论数据库读写时经常会用到，但原理可以应用到在获得一个对象的锁的情况。

只要线程２一获得Entry 的互斥锁，它就会阻击其它线程去改变它，然后它就可以随意做它要做的事情，设置值，然后做其它事情。

你可以想象这里非常耗性能的，因为其它线程在系统各处徘徊着准备要获得锁然后又阻塞。线程越多，系统的响应性就会越慢．

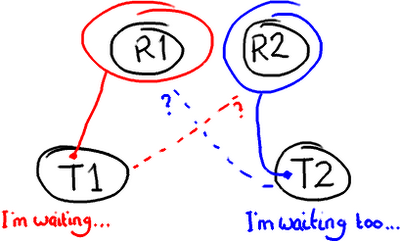
**办法二：乐观锁**

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

在这种情况，当线程２需要去写Entry时才会去锁定它．它需要检查Entry自从上次读过后是否已经被改过了。如果线程１在线程２读完后到达并把值改为”blah”,线程２读到了这个新值，线程２不会把＂fluffy＂写到Entry里并把线程１所写的数据覆盖．线程２会重试（重新读新的值，与旧值比较，如果相等则在变量的值后面附上’y’）,这里在线程２不会关心新的值是什么的情况．或者线程２会抛出一个异常，或者会返回一个某些字段已更新的标志，这是在期望把”fluff”改为”fluffy”的情况．举一个第二种情况的例子，如果你和另外一个用户同时更新一个Ｗiki的页面，你会告诉另外一个用户的线程 Thread 2，它们需要重新加载从Thread1来新的变化，然后再提交它们的内容。

**潜在的问题：死锁**

锁定会带来各种各样的问题，比如死锁，想象有２个线程需要访问两个资源

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

如果你滥用锁技术，两个锁都在获得锁的情况下尝试去获得另外一个锁，那就是你应该重启你的电脑的时候了。（校注：作者还挺幽默）

**很明确的一个问题：锁技术是慢的．．**

关于锁就是它们需要操作系统去做裁定。线程就像两姐妹在为一个玩具在争吵，然后操作系统就是能决定他们谁能拿到玩具的父母，就像当你跑向你父亲告诉他你的姐姐在你玩着的时候抢走了你的变形金刚－他还有比你们争吵更大的事情去担心，他或许在解决你们争吵之前要启动洗碗机并把它摆在洗衣房里。如果你把你的注意力放在锁上，不仅要花时间来让操作系统来裁定。Disruptor论文中讲述了我们所做的一个实验。这个测试程序调用了一个函数，该函数会对一个64位的计数器循环自增5亿次。当单线程无锁时，程序耗时300ms。如果增加一个锁（仍是单线程、没有竞争、仅仅增加锁），程序需要耗时10000ms，慢了两个数量级。更令人吃惊的是，如果增加一个线程（简单从逻辑上想，应该比单线程加锁快一倍），耗时224000ms。使用两个线程对计数器自增5亿次比使用无锁单线程慢1000倍。**并发很难而锁的性能糟糕。**我仅仅是揭示了问题的表面，而且，这个例子很简单。但重点是，如果代码在多线程环境中执行，作为开发者将会遇到更多的困难：

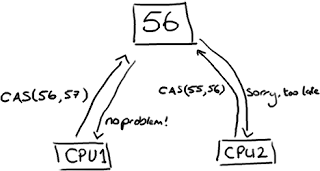
* 代码没有按设想的顺序执行。上面的场景3表明，如果没有注意到多线程访问和写入相同的数据，事情可能会很糟糕。
* 减慢系统的速度。场景3中，使用锁保护代码可能导致诸如死锁或者效率问题。

这就是为什么许多公司在面试时会多少问些并发问题（当然针对Java面试）。不幸的是，即使未能理解问题的本质或没有问题的解决方案，也很容易学会如何回答这些问题。

**Disruptor如何解决这些问题。**

首先，Disruptor根本就不用锁。

取而代之的是，在需要确保操作是线程安全的（特别是，在[多生产者](http://mechanitis.blogspot.com/2011/07/dissecting-disruptor-writing-to-ring.html)的环境下，更新下一个可用的序列号）地方，我们使用[CAS](http://en.wikipedia.org/wiki/Compare-and-swap)（Compare And Swap/Set）操作。这是一个CPU级别的指令，在我的意识中，它的工作方式有点像乐观锁——CPU去更新一个值，但如果想改的值不再是原来的值，操作就失败，因为很明显，有其它操作先改变了这个值。

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

注意，这可以是CPU的两个不同的核心，但不会是两个独立的CPU。

CAS操作比锁消耗资源少的多，因为它们不牵涉操作系统，它们直接在CPU上操作。但它们并非没有代价——在上面的试验中，单线程无锁耗时300ms，单线程有锁耗时10000ms，单线程使用CAS耗时5700ms。所以它比使用锁耗时少，但比不需要考虑竞争的单线程耗时多。

回到Disruptor，在我[讲生产者](http://ifeve.com/disruptor-writing-ringbuffer/)时讲过[ClaimStrategy](https://github.com/LMAX-Exchange/disruptor/blob/version-2.x/code/src/main/com/lmax/disruptor/ClaimStrategy.java)。在这些代码中，你可以看见两个策略，一个是SingleThreadedStrategy（单线程策略）另一个是MultiThreadedStrategy（多线程策略）。你可能会有疑问，为什么在只有单个生产者时不用多线程的那个策略？它是否能够处理这种场景？当然可以。但多线程的那个使用了[AtomicLong](http://download.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/AtomicLong.html)（Java提供的CAS操作），而单线程的使用long，没有锁也没有CAS。这意味着单线程版本会非常快，因为它只有一个生产者，不会产生序号上的冲突。

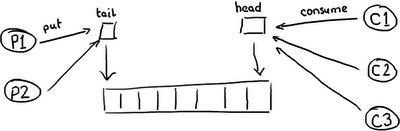
我知道，你可能在想：把一个数字转成AtomicLong不可能是Disruptor速度快的唯一秘密。当然，它不是，否则它不可能称为“为什么这么快（第一部分）”。

但这是非常重要的一点——在整个复杂的框架中，只有这一个地方出现多线程竞争修改同一个变量值。这就是秘密。还记得所有的访问对象都拥有序号吗？如果只有一个生产者，那么系统中的每一个序列号只会由一个线程写入。这意味着没有竞争、不需要锁、甚至不需要CAS。在ClaimStrategy中，如果存在多个生产者，唯一会被多线程竞争写入的序号就是 ClaimStrategy 对象里的那个。

这也是为什么Entry中的每一个变量都[只能被一个消费者写](http://ifeve.com/dissecting-disruptor-wiring-up/)。它确保了没有写竞争，因此不需要锁或者CAS。

**回到为什么队列不能胜任这个工作**

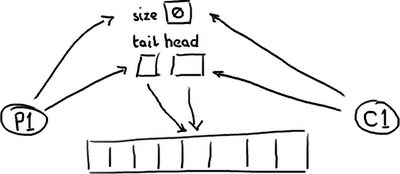
因此你可能会有疑问，为什么队列底层用RingBuffer来实现，仍然在性能上无法与 Disruptor 相比。队列和[最简单的ring buffer](http://en.wikipedia.org/wiki/Circular_buffer)只有两个指针——一个指向队列的头，一个指向队尾：

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

如果有超过一个生产者想要往队列里放东西，尾指针就将成为一个冲突点，因为有多个线程要更新它。如果有多个消费者，那么头指针就会产生竞争，因为元素被消费之后，需要更新指针，所以不仅有读操作还有写操作了。

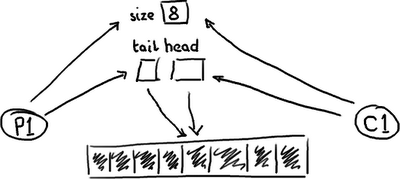
等等，我听到你喊冤了！因为我们已经知道这些了，所以队列常常是单生产者和单消费者（或者至少在我们的测试里是）。  
队列的目的就是为生产者和消费者提供一个地方存放要交互的数据，帮助缓冲它们之间传递的消息。这意味着缓冲常常是满的（生产者比消费者快）或者空的（消费者比生产者快）。生产者和消费者能够步调一致的情况非常少见。

所以，这就是事情的真面目。一个空的队列：

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

…

一个满的队列：

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

*(校对注：这应该是一个双向队列)*

队列需要保存一个关于大小的变量，以便区分队列是空还是满。否则，它需要根据队列中的元素的内容来判断，这样的话，消费一个节点（Entry）后需要做一次写入来清除标记，或者标记节点已经被消费过了。无论采用何种方式实现，在头、尾和大小变量上总是会有很多竞争，或者如果消费操作移除元素时需要使用一个写操作，那元素本身也包含竞争。

基于以上，这三个变量常常在一个[cache line](http://en.wikipedia.org/wiki/CPU_cache)里面，有可能导致[false sharing](http://en.wikipedia.org/wiki/False_sharing)。因此，不仅要担心生产者和消费者同时写size变量（或者元素），还要注意由于头指针尾指针在同一位置，当头指针更新时，更新尾指针会导致缓存不命中。这篇文章已经很长了，所以我就不再详述细节了。

这就是我们所说的“分离竞争点问题”或者队列的“合并竞争点问题”。通过将所有的东西都赋予私有的序列号，并且只允许一个消费者写Entry对象中的变量来消除竞争，Disruptor 唯一需要处理访问冲突的地方，是多个生产者写入 Ring Buffer 的场景。

**总结**

Disruptor相对于传统方式的优点：

1. 没有竞争=没有锁=非常快。
2. 所有访问者都记录自己的序号的实现方式，允许多个生产者与多个消费者共享相同的数据结构。
3. 在每个对象中都能跟踪序列号（ring buffer，claim Strategy，生产者和消费者），加上神奇的[cache line padding](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java)，就意味着没有为伪共享和非预期的竞争。

校订：需要注意Disruptor2.0使用了与本文中不一样的名字。如果对类名感到困惑，请参考[我的变更汇总](http://ifeve.com/disruptor-2-change/)。

剖析Disruptor:为什么会这么快？（二）神奇的缓存行填充

原文地址：<http://ifeve.com/disruptor-padding/>

作者：Trisha  译者：方腾飞  校对：丁一

我们经常提到一个短语Mechanical Sympathy，这个短语也是[Martin博客的标题](http://mechanical-sympathy.blogspot.com/)（译注：Martin Thompson），Mechanical Sympathy讲的是底层硬件是如何运作的，以及与其协作而非相悖的编程方式。

我在[上一篇文章](http://ifeve.com/locks-are-bad/)中提到[RingBuffer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java)后，我们收到一些关于[RingBuffer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java)中填充高速缓存行的评论和疑问。由于这个适合用漂亮的图片来说明，所以我想这是下一个我该解决的问题了。  
*（译注：Martin Thompson很喜欢用Mechanical Sympathy这个短语，这个短语源于赛车驾驶,它反映了驾驶员对于汽车有一种天生的感觉，所以他们对于如何最佳的驾御它非常有感觉。）*

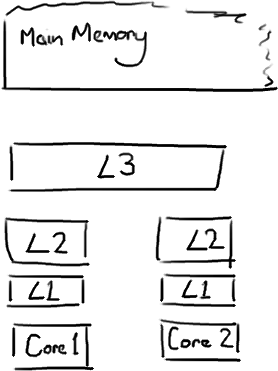
**计算机入门**

我喜欢在[LMAX](http://www.lmaxtrader.co.uk/)工作的原因之一是，在这里工作让我明白从大学和A Level Computing所学的东西实际上还是有意义的。做为一个开发者你可以逃避不去了解CPU，数据结构或者[大O符号](http://en.wikipedia.org/wiki/Big_O_notation)—— 而我用了10年的职业生涯来忘记这些东西。但是现在看来，如果你知道这些知识并应用它，你能写出一些非常巧妙和非常快速的代码。

因此，对在学校学过的人是种复习，对未学过的人是个简单介绍。但是请注意，这篇文章包含了大量的过度简化。

CPU是你机器的心脏，最终由它来执行所有运算和程序。主内存（RAM）是你的数据（包括代码行）存放的地方。本文将忽略硬件驱动和网络之类的东西，因为[Disruptor](https://code.google.com/p/disruptor/)的目标是尽可能多的在内存中运行。

CPU和主内存之间有好几层缓存，因为即使直接访问主内存也是非常慢的。如果你正在多次对一块数据做相同的运算，那么在执行运算的时候把它加载到离CPU很近的地方就有意义了（比如一个循环计数－你不想每次循环都跑到主内存去取这个数据来增长它吧）。

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

越靠近CPU的缓存越快也越小。所以L1缓存很小但很快(译注：L1表示一级缓存)，并且紧靠着在使用它的CPU内核。L2大一些，也慢一些，并且仍然只能被一个单独的 CPU 核使用。L3在现代多核机器中更普遍，仍然更大，更慢，并且被单个插槽上的所有 CPU 核共享。最后，你拥有一块主存，由全部插槽上的所有 CPU 核共享。

当CPU执行运算的时候，它先去L1查找所需的数据，再去L2，然后是L3，最后如果这些缓存中都没有，所需的数据就要去主内存拿。走得越远，运算耗费的时间就越长。所以如果你在做一些很频繁的事，你要确保数据在L1缓存中。

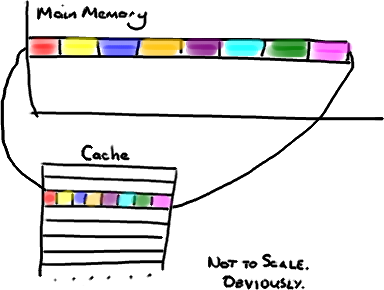
Martin和Mike的 [QCon presentation](http://www.infoq.com/presentations/LMAX)演讲中给出了一些缓存未命中的消耗数据：

|  |  |  |
| --- | --- | --- |
| **从CPU到** | **大约需要的 CPU 周期** | **大约需要的时间** |
| 主存 |  | 约60-80纳秒 |
| QPI 总线传输 (between sockets, not drawn) |  | 约20ns |
| L3 cache | 约40-45 cycles, | 约15ns |
| L2 cache | 约10 cycles, | 约3ns |
| L1 cache | 约3-4 cycles, | 约1ns |
| 寄存器 | 1 cycle |  |

如果你的目标是让端到端的延迟只有 10毫秒，而其中花80纳秒去主存拿一些未命中数据的过程将占很重的一块。

**缓存行**

现在需要注意一件有趣的事情，数据在缓存中不是以独立的项来存储的，如不是一个单独的变量，也不是一个单独的指针。缓存是由缓存行组成的，通常是64字节（译注：这篇文章发表时常用处理器的缓存行是64字节的，比较旧的处理器缓存行是32字节），并且它有效地引用主内存中的一块地址。一个Java的long类型是8字节，因此在一个缓存行中可以存8个long类型的变量。

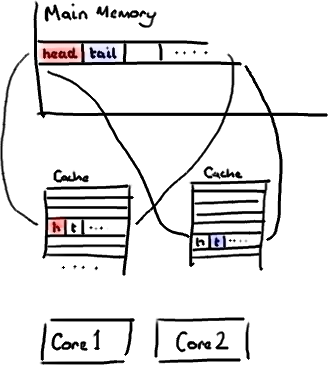
[](http://ifeve.com/wp-content/uploads/2013/01/CacheLines.png)

(为了简化，我将忽略多级缓存)

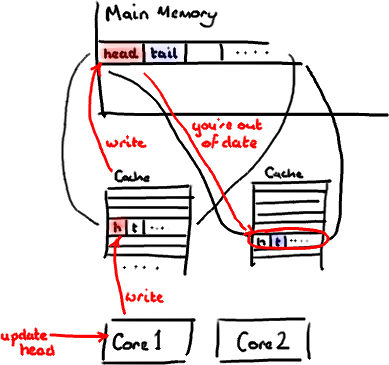
非常奇妙的是如果你访问一个long数组，当数组中的一个值被加载到缓存中，它会额外加载另外7个。因此你能非常快地遍历这个数组。事实上，你可以非常快速的遍历在连续的内存块中分配的任意数据结构。我在第一篇[关于ring buffer的文章](http://mechanitis.blogspot.com/2011/06/dissecting-disruptor-whats-so-special.html)中顺便提到过这个，它解释了我们的ring buffer使用数组的原因。

因此如果你数据结构中的项在内存中不是彼此相邻的（链表，我正在关注你呢），你将得不到免费缓存加载所带来的优势。并且在这些数据结构中的每一个项都可能会出现缓存未命中。

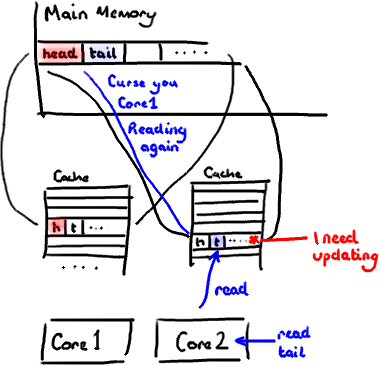
不过，所有这种免费加载有一个弊端。设想你的long类型的数据不是数组的一部分。设想它只是一个单独的变量。让我们称它为head，这么称呼它其实没有什么原因。然后再设想在你的类中有另一个变量紧挨着它。让我们直接称它为tail。现在，当你加载head到缓存的时候，你也免费加载了tail。

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

听想来不错。直到你意识到tail正在被你的生产者写入，而head正在被你的消费者写入。这两个变量实际上并不是密切相关的，而事实上却要被两个不同内核中运行的线程所使用。

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

设想你的消费者更新了head的值。缓存中的值和内存中的值都被更新了，而其他所有存储head的缓存行都会都会失效，因为其它缓存中head不是最新值了。请记住我们必须以整个缓存行作为单位来处理（译注：这是CPU的实现所规定的，详细可参见[深入分析Volatile的实现原理](http://ifeve.com/volatile)），不能只把head标记为无效。

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

现在如果一些正在其他内核中运行的进程只是想读tail的值，整个缓存行需要从主内存重新读取。那么一个和你的消费者无关的线程读一个和head无关的值，它被缓存未命中给拖慢了。

当然如果两个独立的线程同时写两个不同的值会更糟。因为每次线程对缓存行进行写操作时，每个内核都要把另一个内核上的缓存块无效掉并重新读取里面的数据。你基本上是遇到两个线程之间的写冲突了，尽管它们写入的是不同的变量。

这叫作“[伪共享](http://en.wikipedia.org/wiki/False_sharing)”（译注：可以理解为错误的共享），因为每次你访问head你也会得到tail，而且每次你访问tail，你也会得到head。这一切都在后台发生，并且没有任何编译警告会告诉你，你正在写一个并发访问效率很低的代码。

**解决方案－神奇的缓存行填充**

你会看到Disruptor消除这个问题，至少对于缓存行大小是64字节或更少的处理器架构来说是这样的（译注：有可能处理器的缓存行是128字节，那么使用64字节填充还是会存在伪共享问题），通过增加补全来确保ring buffer的序列号不会和其他东西同时存在于一个缓存行中。

public long p1**,** p2**,** p3**,** p4**,** p5**,** p6**,** p7**;** // cache line padding

private volatile long cursor **=** INITIAL\_CURSOR\_VALUE**;**

public long p8**,** p9**,** p10**,** p11**,** p12**,** p13**,** p14**;** // cache line padding

因此没有伪共享，就没有和其它任何变量的意外冲突，没有不必要的缓存未命中。

在你的Entry类中也值得这样做，如果你有不同的消费者往不同的字段写入，你需要确保各个字段间不会出现伪共享。

修改：Martin写了一个从技术上来说更准确更详细的关于[伪共享的文章](http://ifeve.com/false-sharing/)，并且发布了性能测试结果。

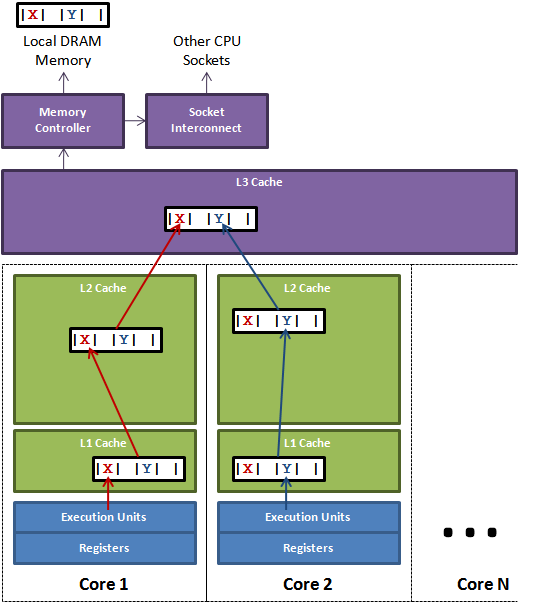
剖析Disruptor:为什么会这么快？（三）伪共享(False Sharing)

原文地址：<http://ifeve.com/false-sharing/>

作者：Martin Thompson  译者：丁一

缓存系统中是以缓存行（cache line）为单位存储的。缓存行是2的整数幂个连续字节，一般为32-256个字节。最常见的缓存行大小是64个字节。当多线程修改互相独立的变量时，如果这些变量共享同一个缓存行，就会无意中影响彼此的性能，这就是伪共享。缓存行上的写竞争是运行在SMP系统中并行线程实现可伸缩性最重要的限制因素。有人将伪共享描述成无声的性能杀手，因为从代码中很难看清楚是否会出现伪共享。

为了让可伸缩性与线程数呈线性关系，就必须确保不会有两个线程往同一个变量或缓存行中写。两个线程写同一个变量可以在代码中发现。为了确定互相独立的变量是否共享了同一个缓存行，就需要了解内存布局，或找个工具告诉我们。Intel VTune就是这样一个分析工具。本文中我将解释Java对象的内存布局以及我们该如何填充缓存行以避免伪共享。



上图说明了伪共享的问题。在核心1上运行的线程想更新变量X，同时核心2上的线程想要更新变量Y。不幸的是，这两个变量在同一个缓存行中。每个线程都要去竞争缓存行的所有权来更新变量。如果核心1获得了所有权，缓存子系统将会使核心2中对应的缓存行失效。当核心2获得了所有权然后执行更新操作，核心1就要使自己对应的缓存行失效。这会来来回回的经过L3缓存，大大影响了性能。如果互相竞争的核心位于不同的插槽，就要额外横跨插槽连接，问题可能更加严重。

**Java内存布局(Java Memory Layout)**

对于HotSpot JVM，所有对象都有两个字长的对象头。第一个字是由24位哈希码和8位标志位（如锁的状态或作为锁对象）组成的Mark Word。第二个字是对象所属类的引用。如果是数组对象还需要一个额外的字来存储数组的长度。每个对象的起始地址都对齐于8字节以提高性能。因此当封装对象的时候为了高效率，对象字段声明的顺序会被重排序成下列基于字节大小的顺序：

1. doubles (8) 和 longs (8)
2. ints (4) 和 floats (4)
3. shorts (2) 和 chars (2)
4. booleans (1) 和 bytes (1)
5. references (4/8)
6. <子类字段重复上述顺序>

*（译注：更多HotSpot虚拟机对象结构相关内容:*[*http://www.infoq.com/cn/articles/jvm-hotspot*](http://www.infoq.com/cn/articles/jvm-hotspot)*）*

了解这些之后就可以在任意字段间用7个long来填充缓存行。在[Disruptor](http://code.google.com/p/disruptor/)里我们对[RingBuffer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java)的cursor和[BatchEventProcessor](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/BatchEventProcessor.java)的序列进行了缓存行填充。

为了展示其性能影响，我们启动几个线程，每个都更新它自己独立的计数器。计数器是volatile long类型的，所以其它线程能看到它们的进展。

public final class FalseSharing **implements** Runnable

**{**

public final static int NUM\_THREADS **=** 4**;** // change

public final static long ITERATIONS **=** 500L **\*** 1000L **\*** 1000L**;**

private final int arrayIndex**;**

private static VolatileLong**[]** longs **=** **new** VolatileLong**[**NUM\_THREADS**];**

static

**{**

**for** **(**int i **=** 0**;** i **<** longs**.**length**;** i**++)**

**{**

longs**[**i**]** **=** **new** VolatileLong**();**

**}**

**}**

public FalseSharing**(**final int arrayIndex**)**

**{**

**this.**arrayIndex **=** arrayIndex**;**

**}**

public static void main**(**final String**[]** args**)** **throws** Exception

**{**

final long start **=** System**.**nanoTime**();**

runTest**();**

System**.**out**.**println**(**"duration = " **+** **(**System**.**nanoTime**()** **-** start**));**

**}**

private static void runTest**()** **throws** InterruptedException

**{**

Thread**[]** threads **=** **new** Thread**[**NUM\_THREADS**];**

**for** **(**int i **=** 0**;** i **<** threads**.**length**;** i**++)**

**{**

threads**[**i**]** **=** **new** Thread**(new** FalseSharing**(**i**));**

**}**

**for** **(**Thread t **:** threads**)**

**{**

t**.**start**();**

**}**

**for** **(**Thread t **:** threads**)**

**{**

t**.**join**();**

**}**

**}**

public void run**()**

**{**

long i **=** ITERATIONS **+** 1**;**

**while** **(**0 **!=** **--**i**)**

**{**

longs**[**arrayIndex**].**value **=** i**;**

**}**

**}**

public final static class VolatileLong

**{**

public volatile long value **=** 0L**;**

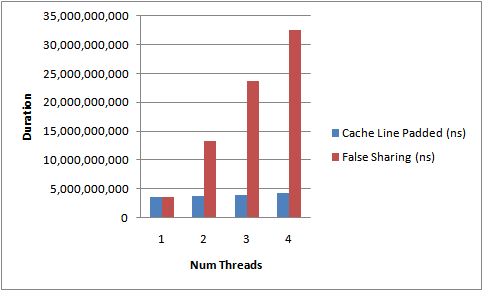
public long p1**,** p2**,** p3**,** p4**,** p5**,** p6**;** // comment out

**}**

**}**

**结果(Results)**

运行上面的代码，增加线程数以及添加/移除缓存行的填充，下面的图2描述了我得到的结果。这是在我4核Nehalem上测得的运行时间。



从不断上升的测试所需时间中能够明显看出伪共享的影响。没有缓存行竞争时，我们几近达到了随着线程数的线性扩展。

这并不是个完美的测试，因为我们不能确定这些VolatileLong会布局在内存的什么位置。它们是独立的对象。但是经验告诉我们同一时间分配的对象趋向集中于一块。

所以你也看到了，伪共享可能是无声的性能杀手。

剖析Disruptor:为什么会这么快？（四）揭秘内存屏障

原文地址：<http://ifeve.com/disruptor-memory-barriers/>

作者：Trisha  译者：杜建雄  校对：欧振聪

最近我博客文章更新有点慢，因为我在忙着写一篇介绍内存屏障（[Memory Barries](http://en.wikipedia.org/wiki/Memory_barrier)）以及如何将其应用于[Disruptor](http://code.google.com/p/disruptor)的文章。问题是，无论我翻阅了多少资料，向耐心的[Martin](http://mechanical-sympathy.blogspot.com/)和[Mike](http://mikes-tech.blogspot.com/)请教了多少遍，以试图理清一些知识点，可我总是不能直观地抓到重点。大概是因为我不具备深厚的背景知识来帮助我透彻理解。

所以，与其像个傻瓜一样试图去解释一些自己都没完全弄懂的东西，还不如在抽象和大量简化的层次上，把我在该领域所掌握的知识分享给大家 。Martin已经写了一篇文章《[going into memory barriers](http://mechanical-sympathy.blogspot.com/2011/07/memory-barriersfences.html)》介绍内存屏障的一些具体细节，所以我就略过不说了。

免责声明：文章中如有错误全由本人负责，与Disruptor的实现和[LMAX](http://www.lmaxtrader.co.uk/)里真正懂这些知识的大牛们无关。

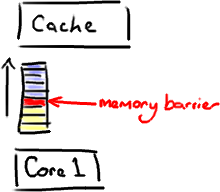
**主题是什么？**

我写这个系列的博客主要目的是解析Disruptor是如何工作的，并深入了解下为什么这样工作。理论上，我应该从可能准备使用disruptor的开发人员的角度来写，以便在代码和技术论文[[Disruptor-1.0.pdf](http://disruptor.googlecode.com/files/Disruptor-1.0.pdf)]之间搭建一座桥梁。这篇文章提及到了内存屏障，我想弄清楚它们到底是什么，以及它们是如何应用于实践中的。

**什么是内存屏障？**

它是一个CPU指令。没错，又一次，我们在讨论CPU级别的东西，以便获得我们想要的性能（Martin著名的Mechanical Sympathy理论）。基本上，它是这样一条指令： a)确保一些特定操作执行的顺序； b)影响一些数据的可见性(可能是某些指令执行后的结果)。

编译器和CPU可以在保证输出结果一样的情况下对指令重排序，使性能得到优化。插入一个内存屏障，相当于告诉CPU和编译器先于这个命令的必须先执行，后于这个命令的必须后执行。正如去拉斯维加斯旅途中各个站点的先后顺序在你心中都一清二楚。

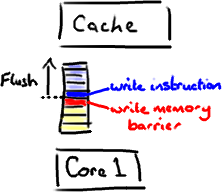


内存屏障另一个作用是强制更新一次不同CPU的缓存。例如，一个写屏障会把这个屏障前写入的数据刷新到缓存，这样任何试图读取该数据的线程将得到最新值，而不用考虑到底是被哪个cpu核心或者哪颗CPU执行的。

**和Java有什么关系？**

现在我知道你在想什么——这不是汇编程序。它是Java。

这里有个神奇咒语叫volatile(我觉得这个词在Java规范中从未被解释清楚)。如果你的字段是volatile，Java内存模型将在写操作后插入一个写屏障指令，在读操作前插入一个读屏障指令。



这意味着如果你对一个volatile字段进行写操作，你必须知道：

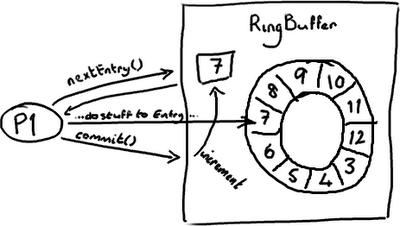
1、一旦你完成写入，任何访问这个字段的线程将会得到最新的值。

2、在你写入前，会保证所有之前发生的事已经发生，并且任何更新过的数据值也是可见的，因为内存屏障会把之前的写入值都刷新到缓存。

**举个例子呗！**

很高兴你这样说了。又是时候让我来画几个甜甜圈了。

[RingBuffer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java)的指针(cursor)（译注：指向队尾元素）属于一个神奇的volatile变量，同时也是我们能够不用锁操作就能实现Disruptor的原因之一。

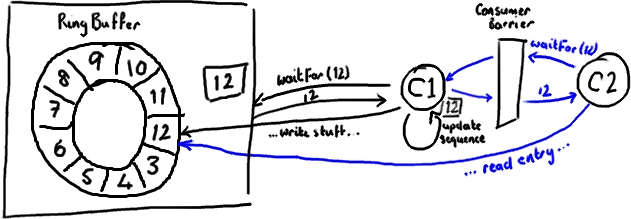


生产者将会取得下一个[Entry](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/AbstractEntry.java?spec=svn109&r=201)（或者是一批）,并可对它（们）作任意改动， 把它（们）更新为任何想要的值。[如你所知](http://ifeve.com/dissecting-the-disruptor-writing-to-the-ring-buffer/)，在所有改动都完成后，生产者对ring buffer调用commit方法来更新序列号（译注:把cursor更新为该Entry的序列号）。对volatile字段(cursor)的写操作创建了一个内存屏障，这个屏障将刷新所有缓存里的值（或者至少相应地使得缓存失效）。

这时候，消费者们能获得最新的序列号码(8)，并且因为内存屏障保证了它之前执行的指令的顺序，消费者们可以确信生产者对7号Entry所作的改动已经可用。

**…那么消费者那边会发生什么？**

消费者中的序列号是volatile类型的，会被若干个外部对象读取——其他的[下游消费者](http://mechanitis.blogspot.com/2011/07/dissecting-disruptor-wiring-up.html)可能在跟踪这个消费者。[ProducerBarrier](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/ProducerBarrier.java?spec=svn109&r=201)/[RingBuffer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/RingBuffer.java?spec=svn109&r=242)(取决于你看的是旧的还是新的代码)跟踪它以确保环没有出现重叠（wrap）的情况（译注：为了防止下游的消费者和上游的消费者对同一个Entry竞争消费，导致在环形队列中互相覆盖数据，下游消费者要对上游消费者的消费情况进行跟踪）。



所以，如果你的下游消费者(C2)看见前一个消费者(C1)在消费号码为12的Entry，当C2的读取也到了12，它在更新序列号前将可以获得C1对该Entry的所作的更新。

基本来说就是，C1更新序列号前对ring buffer的所有操作（如上图黑色所示），必须先发生，待C2拿到C1更新过的序列号之后，C2才可以为所欲为（如上图蓝色所示）。

**对性能的影响**

内存屏障作为另一个CPU级的指令，没有[锁那样大的开销](http://ifeve.com/locks-are-bad/)。内核并没有在多个线程间干涉和调度。但凡事都是有代价的。内存屏障的确是有开销的——编译器/cpu不能重排序指令，导致不可以尽可能地高效利用CPU，另外刷新缓存亦会有开销。所以不要以为用volatile代替锁操作就一点事都没。

你会注意到Disruptor的实现对序列号的读写频率尽量降到最低。对volatile字段的每次读或写都是相对高成本的操作。但是，也应该认识到在批量的情况下可以获得很好的表现。如果你知道不应对序列号频繁读写，那么很合理的想到，先获得一整批Entries,并在更新序列号前处理它们。这个技巧对生产者和消费者都适用。以下的例子来自[BatchConsumer](http://code.google.com/p/disruptor/source/browse/trunk/code/src/main/com/lmax/disruptor/BatchConsumer.java?r=239):

long nextSequence **=** sequence **+** 1**;**

**while** **(**running**)**

**{**

**try**

**{**

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

**while** **(**nextSequence **<=** availableSequence**)**

**{**

entry **=** consumerBarrier**.**getEntry**(**nextSequence**);**

handler**.**onAvailable**(**entry**);**

nextSequence**++;**

**}**

handler**.**onEndOfBatch**();**

sequence **=** entry**.**getSequence**();**

**}**

…

**catch** **(**final Exception ex**)**

**{**

exceptionHandler**.**handle**(**ex**,** entry**);**

sequence **=** entry**.**getSequence**();**

nextSequence **=** entry**.**getSequence**()** **+** 1**;**

**}**

**}**

（你会注意到，这是个旧式的代码和命名习惯，因为这是摘自我以前的博客文章，我认为如果直接转换为新式的代码和命名习惯会让人有点混乱）

在上面的代码中，我们在消费者处理entries的循环中用一个局部变量（nextSequence）来递增。这表明我们想尽可能地减少对volatile类型的序列号的进行读写。

**总结**

内存屏障是CPU指令，它允许你对数据什么时候对其他进程可见作出假设。在Java里，你使用volatile关键字来实现内存屏障。使用volatile意味着你不用被迫选择加锁，并且还能让你获得性能的提升。

但是，你需要对你的设计进行一些更细致的思考，特别是你对volatile字段的使用有多频繁，以及对它们的读写有多频繁。

PS：上文中讲到的Disruptor中使用的[New World Order](http://mechanitis.blogspot.com/2011/08/disruptor-20-all-change-please.html)是一种完全不同于我目前为止所发表的博文中的命名习惯。我想下一篇文章会对旧式的和新式的命名习惯做一个对照。