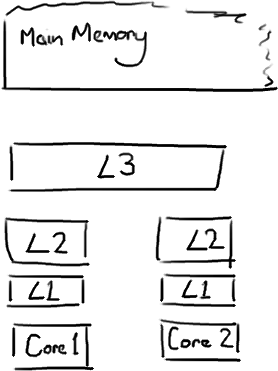
# **缓存行**

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/" \t "/home/ckg/Documents\\x/_blank)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。

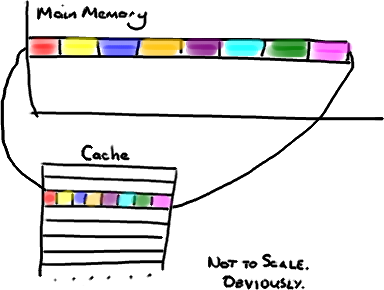
Martin和Mike的 [QCon presentation](http://www.infoq.com/presentations/LMAX" \t "/home/ckg/Documents\\x/_blank)演讲中给出了一些缓存未命中的消耗数据：

|  |  |  |
| --- | --- | --- |
| ****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。

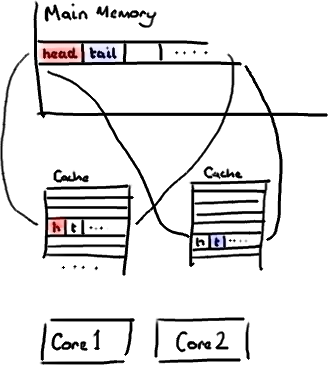
[](http://ifeve.com/wp-content/uploads/2013/01/CacheLines.png)

(为了简化，我将忽略多级缓存)

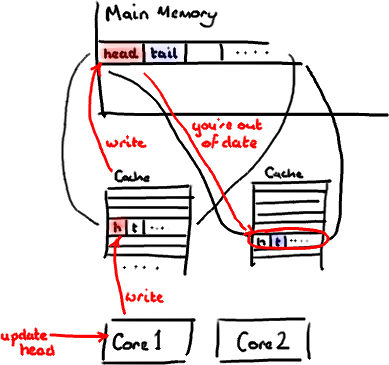
非常奇妙的是如果你访问一个long数组，当数组中的一个值被加载到缓存中，它会额外加载另外7个。因此你能非常快地遍历这个数组。事实上，你可以非常快速的遍历在连续的内存块中分配的任意数据结构。我在第一篇[关于ring buffer的文章](http://mechanitis.blogspot.com/2011/06/dissecting-disruptor-whats-so-special.html" \t "/home/ckg/Documents\\x/_blank)中顺便提到过这个，它解释了我们的ring buffer使用数组的原因。

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。

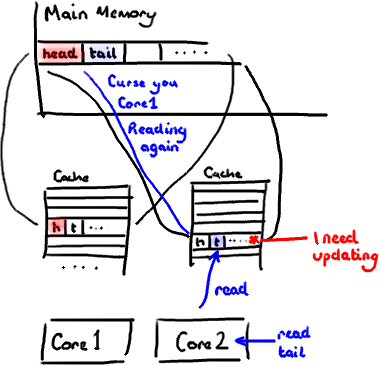
不过，所有这种免费加载有一个弊端。设想你的long类型的数据不是数组的一部分。设想它只是一个单独的变量。让我们称它为head，这么称呼它其实没有什么原因。然后再设想在你的类中有另一个变量紧挨着它。让我们直接称它为tail。现在，当你加载head到缓存的时候，你也免费加载了tail。

[](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" \t "/home/ckg/Documents\\x/_blank)），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" \t "/home/ckg/Documents\\x/_blank)”（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。

|  |  |
| --- | --- |
| 1 | public longp1, p2, p3, p4, p5, p6, p7; // cache line padding |

|  |  |
| --- | --- |
| 2 | privatevolatile long cursor = INITIAL\_CURSOR\_VALUE; |

|  |  |
| --- | --- |
| 3 | publiclong 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。

### **False Sharing**

一个cache lien可以被多个不同的线程所使用。如果有其他线程修改了v2的值，线程1和线程2将会强制重新加载cache line。你可以会疑惑我们只是修改了v2的值不应该会影响其他变量，为啥线程1和线程2需要重新加载cache line呢。然后，即使对于多个线程来说这些更新操作是逻辑独立的，但是一致性的保持是以cache line为基础的，而不是以单个独立的元素。这种明显没有必要的共享数据的方式被称作“False sharing”.

### **Padding**

为了获取一个cache line，核心需要执行几百个指令。

如果核心需要等待一个cache line重新加载，核心将会停止做其他事情，这种现象被称为"Stall".Stalls可以通过减少“False Sharing”,一个减少"false sharing"的技巧是填充数据结构，使得线程操作的变量落入到不同的cache line中。

下面是一个填充了的数据结构的例子，尝试着把x和v1放入到不同的cache line中

public class FalseSharingWithPadding {

public volatile long x;

public volatile long p2; *// padding*

public volatile long p3; *// padding*

public volatile long p4; *// padding*

public volatile long p5; *// padding*

public volatile long p6; *// padding*

public volatile long p7; *// padding*

public volatile long p8; *// padding*

public volatile long v1;

}

在你准备填充你的所有数据结构之前，你必须了解jvm会减少或者重排序没有使用的字段，因此可能会重新引入“false sharing”。因此对象会在堆中的位置是没有办法保证的。

为了减少未使用的填充字段被优化掉的机会，将这些字段设置成为volatile会很有帮助。对于填充的建议是你只需要在高度竞争的并发类上使用填充，并且在你的目标架构上测试使用有很大提升之后采用填充。最好的方式是做10000玄幻迭代，消除JVM的实时优化的影响。

### **java8 和 @Contended**

比起引入填充字段，一个更加简单有效的方式是在你需要避免“false sharing”的字段上标记注解，这可以暗示虚拟机“这个字段可以分离到不同的cache line中”，这是JEP 142的目标。

JEP引入了 @Contended 注解。

public class Point {

int x;

@Contended

int y;

}

以上代码使得x和y都在不同的cache line中。@Contended 使得y字段远离了对象头部分。

ｖｏｌａｔｉｌｅ

## ****引言****

在多线程并发编程中synchronized和Volatile都扮演着重要的角色，Volatile是**轻量级的synchronized**，它在多处理器开发中保证了共享变量的“可见性”。可见性的意思是当一个线程修改一个共享变量时，另外一个线程能读到这个修改的值。它在某些情况下比synchronized的开销更小，本文将深入分析在硬件层面上Inter处理器是如何实现Volatile的，通过深入分析能帮助我们正确的使用Volatile变量。

## ****术语定义****

|  |  |  |
| --- | --- | --- |
| 术语 | 英文单词 | 描述 |
| 共享变量 |  | 在多个线程之间能够被共享的变量被称为共享变量。共享变量包括所有的实例变量，静态变量和数组元素。他们都被存放在堆内存中，Volatile只作用于共享变量。 |
| 内存屏障 | Memory Barriers | 是一组处理器指令，用于实现对内存操作的顺序限制。 |
| 缓冲行 | Cache line | 缓存中可以分配的最小存储单位。处理器填写缓存线时会加载整个缓存线，需要使用多个主内存读周期。 |
| 原子操作 | Atomic operations | 不可中断的一个或一系列操作。 |
| 缓存行填充 | cache line fill | 当处理器识别到从内存中读取操作数是可缓存的，处理器读取整个缓存行到适当的缓存（L1，L2，L3的或所有） |
| 缓存命中 | cache hit | 如果进行高速缓存行填充操作的内存位置仍然是下次处理器访问的地址时，处理器从缓存中读取操作数，而不是从内存。 |
| 写命中 | write hit | 当处理器将操作数写回到一个内存缓存的区域时，它首先会检查这个缓存的内存地址是否在缓存行中，如果存在一个有效的缓存行，则处理器将这个操作数写回到缓存，而不是写回到内存，这个操作被称为写命中。 |
| 写缺失 | write misses the cache | 一个有效的缓存行被写入到不存在的内存区域。 |

## ****Volatile的官方定义****

Java语言规范第三版中对volatile的定义如下： java编程语言允许线程访问共享变量，为了确保共享变量能被准确和一致的更新，线程应该确保通过排他锁单独获得这个变量。Java语言提供了volatile，在某些情况下比锁更加方便。如果一个字段被声明成volatile，java线程内存模型确保所有线程看到这个变量的值是一致的。

## ****为什么要使用Volatile****

Volatile变量修饰符如果使用**恰当**的话，它比synchronized的**使用和执行成本会更低**，因为它不会引起线程上下文的切换和调度。

## ****Volatile的实现原理****

那么Volatile是如何来保证可见性的呢？在x86处理器下通过工具获取JIT编译器生成的汇编指令来看看对Volatile进行写操作CPU会做什么事情。

|  |  |
| --- | --- |
| Java代码： | instance = new Singleton();//instance是volatile变量 |
| 汇编代码： | 0x01a3de1d: movb $0x0,0x1104800(%esi);0x01a3de24: **lock** addl $0x0,(%esp); |

有volatile变量修饰的共享变量进行写操作的时候会多第二行汇编代码，通过查IA-32架构软件开发者手册可知，lock前缀的指令在多核处理器下会引发了两件事情。

* 将当前处理器缓存行的数据会写回到系统内存。
* 这个写回内存的操作会引起在其他CPU里缓存了该内存地址的数据无效。

处理器为了提高处理速度，不直接和内存进行通讯，而是先将系统内存的数据读到内部缓存（L1,L2或其他）后再进行操作，但操作完之后不知道何时会写到内存，如果对声明了Volatile变量进行写操作，JVM就会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写回到系统内存。但是就算写回到内存，如果其他处理器缓存的值还是旧的，再执行计算操作就会有问题，所以在多处理器下，为了保证各个处理器的缓存是一致的，就会实现缓存一致性协议，每个处理器通过嗅探在总线上传播的数据来检查自己缓存的值是不是过期了，当处理器发现自己缓存行对应的内存地址被修改，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作的时候，会强制重新从系统内存里把数据读到处理器缓存里。

这两件事情在IA-32软件开发者架构手册的第三册的多处理器管理章节（第八章）中有详细阐述。

**Lock前缀指令会引起处理器缓存回写到内存**。Lock前缀指令导致在执行指令期间，声言处理器的 LOCK# 信号。在多处理器环境中，LOCK# 信号确保在声言该信号期间，处理器可以独占使用任何共享内存。（因为它会锁住总线，导致其他CPU不能访问总线，不能访问总线就意味着不能访问系统内存），但是在最近的处理器里，LOCK＃信号一般不锁总线，而是锁缓存，毕竟锁总线开销比较大。在8.1.4章节有详细说明锁定操作对处理器缓存的影响，对于Intel486和Pentium处理器，在锁操作时，总是在总线上声言LOCK#信号。但在P6和最近的处理器中，如果访问的内存区域已经缓存在处理器内部，则不会声言LOCK#信号。相反地，它会锁定这块内存区域的缓存并回写到内存，并使用缓存一致性机制来确保修改的原子性，此操作被称为“缓存锁定”，**缓存一致性机制会阻止同时修改被两个以上处理器缓存的内存区域数据**。

**一个处理器的缓存回写到内存会导致其他处理器的缓存无效**。IA-32处理器和Intel 64处理器使用MESI（修改，独占，共享，无效）控制协议去维护内部缓存和其他处理器缓存的一致性。在多核处理器系统中进行操作的时候，IA-32 和Intel 64处理器能嗅探其他处理器访问系统内存和它们的内部缓存。它们使用嗅探技术保证它的内部缓存，系统内存和其他处理器的缓存的数据在总线上保持一致。例如在Pentium和P6 family处理器中，如果通过嗅探一个处理器来检测其他处理器打算写内存地址，而这个地址当前处理共享状态，那么正在嗅探的处理器将无效它的缓存行，在下次访问相同内存地址时，强制执行缓存行填充。

## ****Volatile的使用优化****

著名的Java并发编程大师Doug lea在JDK7的并发包里新增一个队列集合类LinkedTransferQueue，他在使用Volatile变量时，用一种追加字节的方式来优化队列出队和入队的性能。

**追加字节能优化性能**？这种方式看起来很神奇，但如果深入理解处理器架构就能理解其中的奥秘。让我们先来看看LinkedTransferQueue这个类，它使用一个内部类类型来定义队列的头队列（Head）和尾节点（tail），而这个内部类PaddedAtomicReference相对于父类AtomicReference只做了一件事情，就将共享变量追加到64字节。我们可以来计算下，一个对象的引用占4个字节，它追加了15个变量共占60个字节，再加上父类的Value变量，一共64个字节。

|  |  |
| --- | --- |
| 01 | /\*\* head of the queue \*/ |

|  |  |
| --- | --- |
| 02 | **private** **transient** **final** PaddedAtomicReference<QNode> head; |

|  |  |
| --- | --- |
| 03 |  |

|  |  |
| --- | --- |
| 04 | /\*\* tail of the queue \*/ |

|  |  |
| --- | --- |
| 05 | **private** **transient** **final** PaddedAtomicReference<QNode> tail; |

|  |  |
| --- | --- |
| 06 |  |

|  |  |
| --- | --- |
| 07 | **static** **final** **class** PaddedAtomicReference <T> **extends** AtomicReference <T> { |

|  |  |
| --- | --- |
| 08 |  |

|  |  |
| --- | --- |
| 09 | // enough padding for 64bytes with 4byte refs |

|  |  |
| --- | --- |
| 10 | Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe; |

|  |  |
| --- | --- |
| 11 |  |

|  |  |
| --- | --- |
| 12 | PaddedAtomicReference(T r) { |

|  |  |
| --- | --- |
| 13 |  |

|  |  |
| --- | --- |
| 14 | **super**(r); |

|  |  |
| --- | --- |
| 15 |  |

|  |  |
| --- | --- |
| 16 | } |

|  |  |
| --- | --- |
| 17 |  |

|  |  |
| --- | --- |
| 18 | } |

|  |  |
| --- | --- |
| 19 |  |

|  |  |
| --- | --- |
| 20 | **public** **class** AtomicReference <V> **implements** java.io.Serializable { |

|  |  |
| --- | --- |
| 21 |  |

|  |  |
| --- | --- |
| 22 | **private** **volatile** V value; |

|  |  |
| --- | --- |
| 23 |  |

|  |  |
| --- | --- |
| 24 | //省略其他代码 |

|  |  |
| --- | --- |
| 25 |  |

|  |  |
| --- | --- |
| 26 | ｝ |

**为什么追加64字节能够提高并发编程的效率呢**？ 因为对于英特尔酷睿i7，酷睿， Atom和NetBurst， Core Solo和Pentium M处理器的L1，L2或L3缓存的高速缓存行是64个字节宽，不支持部分填充缓存行，这意味着如果队列的头节点和尾节点都不足64字节的话，处理器会将它们都读到同一个高速缓存行中，在多处理器下每个处理器都会缓存同样的头尾节点，当一个处理器试图修改头接点时会将整个缓存行锁定，那么在缓存一致性机制的作用下，会导致其他处理器不能访问自己高速缓存中的尾节点，而队列的入队和出队操作是需要不停修改头接点和尾节点，所以在多处理器的情况下将会严重影响到队列的入队和出队效率。Doug lea使用追加到64字节的方式来填满高速缓冲区的缓存行，避免头接点和尾节点加载到同一个缓存行，使得头尾节点在修改时不会互相锁定。

那么是不是在使用Volatile变量时都应该追加到64字节呢？不是的。在两种场景下不应该使用这种方式。第一：**缓存行非64字节宽的处理器**，如P6系列和奔腾处理器，它们的L1和L2高速缓存行是32个字节宽。第二：**共享变量不会被频繁的写**。因为使用追加字节的方式需要处理器读取更多的字节到高速缓冲区，这本身就会带来一定的性能消耗，共享变量如果不被频繁写的话，锁的几率也非常小，就没必要通过追加字节的方式来避免相互锁定。