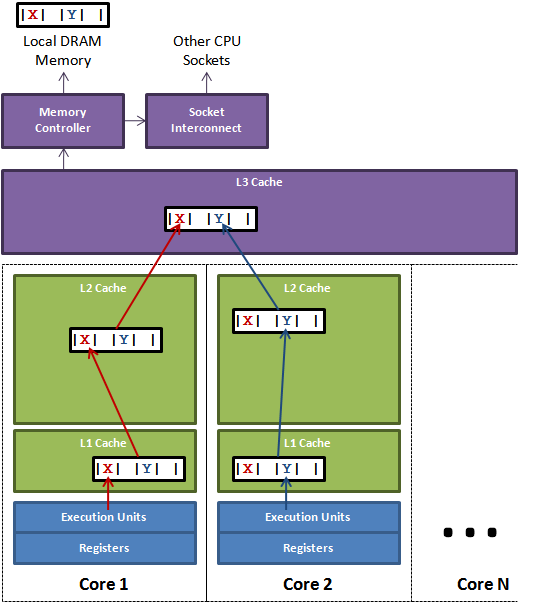
伪共享(False Sharing)

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**;**

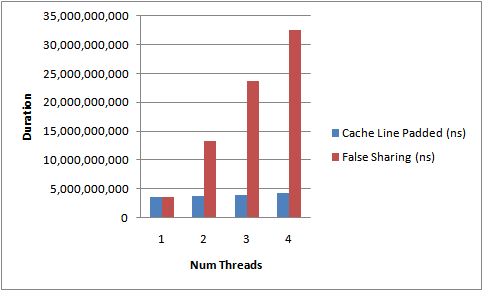
public long p1**,** p2**,** p3**,** p4**,** p5**,** p6**;** // comment out

**}**

**}**

**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。

Java 7与伪共享的新仇旧恨

原文：[False Shareing && Java 7 （依然是马丁的博客）](http://ifeve.com/false-sharing-java-7/)

**译者**：杨帆 **校对：**方腾飞

在我[前一篇有关伪共享的博文](http://ifeve.com/falsesharing/)中，我提到了可以加入闲置的long字段来填充缓存行来避免伪共享。但是看起来Java 7变得更加智慧了，它淘汰或者是重新排列了无用的字段，这样我们之前的办法在Java 7下就不奏效了，但是伪共享依然会发生。我在不同的平台上实验了一些列不同的方案，并且最终发现下面的代码是最可靠的。（译者注：下面的是最终版本，马丁在大家的帮助下修改了几次代码）

**import** java**.**util**.**concurrent**.**atomic**.**AtomicLong**;**

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 PaddedAtomicLong**[]** longs **=** **new** PaddedAtomicLong**[**NUM\_THREADS**];**

    static

**{**

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

**{**

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

**}**

**}**

    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**].**set**(**i**);**

**}**

**}**

  // 这段代码的来历可以看4楼的回复

    public static long sumPaddingToPreventOptimisation**(**final int index**)**

**{**

        PaddedAtomicLong v **=** longs**[**index**];**

        return v**.**p1 **+** v**.**p2 **+** v**.**p3 **+** v**.**p4 **+** v**.**p5 **+** v**.**p6**;**

**}**

    public static class PaddedAtomicLong **extends** AtomicLong

**{**

        public volatile long p1**,** p2**,** p3**,** p4**,** p5**,** p6 **=** 7L**;**

**}**

**}**

用以上这种办法我获得了和上一篇博客里提到的相近的性能，读者可以把PaddedAtomicLong里面那行填充物注释掉再跑测试看看效果。

我想我们大家都有权去跟Oracle投诉，让他们在JDK里默认加入缓存行对齐的函数或者是被填充好的原子类型，这和其他一些底层改变会让Java成为一门真真正正的并发编程语言。我们一直以来不断的在听到他们讲多核时代正在到来，但是我要说的是在这方面Java需要快点赶上来。

———————————————–

*（译者注：博文后面的评论和交流也很精彩，也讲述了这段示例代码的进化过程，一起翻译出来：）*

**1楼：Ashwin Jayaprakash**

在前一篇博文中你创建了一个数组来放VolatileLong，这次你又用一个数组放AtomicLongArray（译者注：此处我觉得他可能是写错了，应该是说AtomicLong吧）。  
但是如何能保证AtomicLongArray或VolatileLong会被紧挨着分配在内存中？  
那么，就算你在一个循环中创建他们，并且很幸运的，他们获得了连续的内存空间，但是依然无法保证这四个实例会在堆空间里紧挨着。如果他们被分布在JVM的旧生代堆里并且没有被压实的话，直到一次主要GC压实旧生代之前，重新分配填充是没必要的，因为他们在堆中是分散的。  
所以你最好对读者说明，我们无法控制JVM如何在堆中对这些实例分配内存。（译者注：没办法，要精确控制内存来保证性能的话就不要用Java了，要不直接用C好了）

———————————————–  
**2楼：马丁**

你大体上说的是对的，Ashwin，我们无法保证如何在堆空间中放置Java对象，这是伪共享问题发生的根源。如果你有一些跨线程的指针或者计数器，那么确保他们在不同的缓存行中是非常重要的，否则的话程序就无法按CPU的核数扩展。填充的根本意义在于保护这些跨线程的指针和计数器，以确保他们在不同的缓存行中。  
这个是有意义的吧？

———————————————–  
**3楼：Ashwin Jayaprakash**

嗯，有道理。那你可不可以创建一个大的AtomicLongArray，然后让不同的线程去更新第8,16,32个元素呢？（译者注：也算是消除竞争的一个办法，但是既然完全没有竞争还要多线程做什么？）而不是搞四个AtomicLongArray，而每个线程都去竞争访问同一个数组元素。  
谢谢马丁花时间写了这么多。  
———————————————–  
**4楼：马丁**

如果我可以提前知道更多的业务逻辑那么你说的方式是可行的。但通常情况下在设计一个大型系统的时候，我们无法提前知道很多事情，或者我们要为其他的应用创造一个通用的类库。  
我很难为很多不同的上下文场景写一个足够小巧简单的示例，而我上面写的示例是为了说明当伪共享发生的时候有多糟糕。如果你在你的数据结构中做了填充，那么你就不必担心他们在内存中如何分配。我们用一个更好的方案来替代AtomicLong，并且你可以使用AtomicLong的所有常规方法：

static class PaddedAtomicLong extends AtomicLong  
{  
public volatile long p1, p2, p3, p4, p5, p6, p7 = 7L;  
}

我是多希望Java委员会可以认识到这个问题的严重性，并且在JDK里加入对缓存行对齐和填充的基础方法。这是在Disruptor中有关性能BUG的最大根源。  
我也根据以上的反馈更新了文章。

———————————————–

**5楼：Gil Tene**  
马丁，我很同意你的观点，如果我们有一种办法可以指定某个字段占有独自的缓存行，并且让JVM自动处理如何在对象布局上的正确填充，那这个世界会和谐的多。你搞的这个人造填充将会是很美好的一个事情，但是你也知道，实际上的对象布局情况要取决于JVM的特定实现。  
我是一个偏执狂，我给你的填充方案里加了一些东西，使那些个用于填充的字段很难被JVM优化掉。一个耍小聪明的JVM还是会把你用于填充的P1-P7的字段优化掉，原理是这样滴：PaddedAtomicLong类如果只对final的FalseSharing类可见（就是说PaddedAtomicLong不能再被继承了）。这样一来编译器就会“知道”它正在审视的是所有可以看到这个填充字段的代码，这样就可以证明没有行为依赖于p1到p7这些字段。那么“聪明”的JVM会把上面这些丝毫不占地方的字段统统优化掉。  
那么针对这样的情况，你可以巧妙的让PaddedAtomicLong类在FalseSharing类之外可见，比如直接加一个依赖于p1到p7的公开的访问函数，并且这个函数在理论上可以被外界访问到。

———————————————–  
**6楼：马丁**  
我根据Gil的反馈做了修改。

———————————————–  
**7楼：Stanimir Simeonoff**  
直接用一个数组并且把元素放在中间的位置上（或者直接用bytebuffer，而你却为了这个写了这么一大篇），Java是不会重排他们的，我就是这样来消除伪共享的。

———————————————–  
8楼：马丁  
我以前经常像你这么干，比如搞一个长度是15的数组，把元素放在正中间，但是，如果我们需要volatile这个语意就行不通了。对于你的情况来说，你只需要用AtomicLongArray或者类似的。根据我的测量，在一个算法中，这种间接引用（译者注:原词是indirection，我理解也许是指间接引用，即不是直接使用数组，而是使用AtomicLongArray这种包装过的数组）和边界检查的消耗是显著的。  
据我所知，一些人建议加入@Contened注解来标记一个字段，让这个被标记的字段拥有独立的缓存行，我希望这个快点到来。

8.1楼：John  
你好，马丁，我看到在Disruptor当前的版本中Sequence类用的是unsafe.compareAndSwapLong(..)来更新第七个下标的long。  
为什么不数组的长度不是15或者是其他的数值？如果长度是15的话会把2级缓存的缓存行也填充掉么？  
谢谢。

8.2楼：马丁

因为用7个下标保证了会有56个字节填充在数值的任何一边，56字节的填充+8字节的long数值正好装进一行64字节的缓存行。  
———————————————–  
**9楼：Stanimir Simeonoff**

是的，马丁，我指的就是AtomicLongArray，如果你不想为间接引用和边界检查买单，Unsafe 是一个选项(甚至总是这样)。  
———————————————–  
**10楼：Mohan Radhakrishnan**

哪里有一些简单硬件说明书是讲述缓存行的关键概念么？我想找一些插图什么的来理解核心和缓存直接如何交互造成了伪共享。  
———————————————–  
**11楼：马丁**  
你可以参照下面这个PDF的第3和第4章：  
<http://img.delivery.net/cm50content/intel/ProductLibrary/100412_Parallel_Programming_02.pdf>  
———————————————–  
**12楼：ying**  
你的博客太NB了，多谢马丁，我关于填充有两个疑问：  
1.long占8字节，对象引用占16字节，但是这个实现是

public final static class VolatileLong // 16byte</pre>

**{**

public volatile long value **=** 0L**;** // 8 byte

public long p1**,** p2**,** p3**,** p4**,** p5**,** p6**;** // 6\*8 = 48byte

**}**

看起来好像是72个字节啊。  
2.你是发现这个问题的？是去查汇编代码吗？

12.1楼：马丁  
我不希望在缓存行中的标记字在取出锁或者垃圾回收器在老化对象的时候被修改。  
就算默认启用64位模式的压缩指针，它还是会包含类指针在对象的头部。  
<https://wikis.oracle.com/display/HotSpotInternals/CompressedOops>  
这个伪共享的问题我是在多年前发现的，但是我为一个应用做性能测试，发现性能时高时低，追查原因下去发现是伪共享问题。

12.2楼：ying  
那你是如何缩小问题的范围最后发现问题的呢？需要深入分析汇编代码么？

11.3楼：马丁  
汇编代码是不会显示出问题的，你需要去追查为什么CPU的2级缓存总是不命中，追查下去就知道了。  
———————————————–

**13楼：Joachim**

关于@Contended注解的提案在这里:  
<http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007309.html>  
牛文啊，赞！

***原创文章，转载请注明：****转载自*[*并发编程网 – ifeve.com*](http://ifeve.com/)***本文链接地址:***[*Java 7与伪共享的新仇旧恨*](http://ifeve.com/false-shareing-java-7-cn/)

伪共享和缓存行填充，从Java 6, Java 7 到Java 8

关于伪共享的文章已经很多了，对于多线程编程来说，特别是多线程处理列表和数组的时候，要非常注意伪共享的问题。否则不仅无法发挥多线程的优势，还可能比单线程性能还差。随着JAVA版本的更新，再各个版本上减少伪共享的做法都有区别，一不小心代码可能就失效了，要注意进行测试。这篇文章总结一下。

**JAVA 6下的方案**

解决伪共享的办法是使用缓存行填充，使一个对象占用的内存大小刚好为64bytes或它的整数倍，这样就保证了一个缓存行里不会有多个对象。《[剖析Disruptor:为什么会这么快？(三)伪共享](http://developer.51cto.com/art/201306/398232.htm)》提供了缓存行填充的例子：

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

**}**

**}**

VolatileLong通过填充一些无用的字段p1,p2,p3,p4,p5,p6，再考虑到对象头也占用8bit, 刚好把对象占用的内存扩展到刚好占64bytes（或者64bytes的整数倍）。这样就避免了一个缓存行中加载多个对象。但这个方法现在只能适应JAVA6 及以前的版本了。

（注：如果我们的填充使对象size大于64bytes，比如多填充16bytes– public long p1, p2, p3, p4, p5, p6, p7, p8;。理论上同样应该避免伪共享问题，但事实是这样的话执行速度同样慢几倍，只比没有使用填充好一些而已。还没有理解其原因。所以测试下来，必须是64bytes的整数倍）

**JAVA 7下的方案**

上面这个例子在JAVA 7下已经不适用了。因为JAVA 7会优化掉无用的字段，可以参考《[False Sharing && Java 7](http://ifeve.com/false-shareing-java-7-cn/)》。

因此，JAVA 7下做缓存行填充更麻烦了，需要使用继承的办法来避免填充被优化掉，《[False Sharing && Java 7](http://ifeve.com/false-shareing-java-7-cn/)》里的例子我觉得不是很好，于是我自己做了一些优化，使其更通用：

public final class FalseSharing **implements** Runnable **{**

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

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

private final int arrayIndex**;**

private static VolatileLong**[]** longs**;**

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

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

**}**

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

Thread**.**sleep**(**10000**);**

System**.**out**.**println**(**"starting...."**);**

**if** **(**args**.**length **==** 1**)** **{**

NUM\_THREADS **=** Integer**.**parseInt**(**args**[**0**]);**

**}**

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

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

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

**}**

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 class VolatileLongPadding **{**

public volatile long p1**,** p2**,** p3**,** p4**,** p5**,** p6**;** // 注释

**}**

public class VolatileLong **extends** VolatileLongPadding **{**

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

**}**

把padding放在基类里面，可以避免优化。（这好像没有什么道理好讲的，JAVA7的内存优化算法问题，能绕则绕）。不过，这种办法怎么看都有点烦，借用另外一个博主的话：做个java程序员真难。

**JAVA 8下的方案**

在JAVA 8中，缓存行填充终于被JAVA原生支持了。JAVA 8中添加了一个@Contended的注解，添加这个的注解，将会在自动进行缓存行填充。以上的例子可以改为：

public final class FalseSharing **implements** Runnable **{**

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

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

private final int arrayIndex**;**

private static VolatileLong**[]** longs**;**

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

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

**}**

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

Thread**.**sleep**(**10000**);**

System**.**out**.**println**(**"starting...."**);**

**if** **(**args**.**length **==** 1**)** **{**

NUM\_THREADS **=** Integer**.**parseInt**(**args**[**0**]);**

**}**

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

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

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

**}**

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**;**

**}**

**}**

**}**

**import** sun**.**misc**.**Contended**;**

@Contended

public class VolatileLong **{**

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

**}**

执行时，必须加上虚拟机参数-XX:-RestrictContended，@Contended注释才会生效。很多文章把这个漏掉了，那样的话实际上就没有起作用。

@Contended注释还可以添加在字段上，今后再写文章详细介绍它的用法。

（后记：以上代码基于32位JDK测试，64位JDK下，对象头大小不同，有空再测试一下）

参考

<http://mechanical-sympathy.blogspot.com/2011/07/false-sharing.html>

<http://mechanical-sympathy.blogspot.hk/2011/08/false-sharing-java-7.html>

<http://robsjava.blogspot.com/2014/03/what-is-false-sharing.html>

Java8的伪共享和缓存行填充--@Contended注释

在我的前一篇文章<[伪共享和缓存行填充，从Java 6, Java 7 到Java 8](http://www.cnblogs.com/Binhua-Liu/p/5620339.html)>中, 我们演示了在Java 8中，可以采用@Contended在类级别上的注释，来进行缓存行填充。这样，多线程情况下的伪共享冲突问题。 感兴趣的同学可以查看该文。

其实，@Contended注释还可以应用于字段级别（Field-Level），当应用于字段级别时，被注释的字段将和其他字段隔离开来，会被加载在独立的缓存行上。在字段级别上，@Contended还支持一个“contention group”属性（Class-Level不支持），同一group的字段们在内存上将是连续，但和其他他字段隔离开来。

上面只是泛泛的介绍一下。关于@Contended应用于Field-Level特别是contention group的相关的资料很少，源代码中的注释中有一些，还有关于[JEP-142](http://openjdk.java.net/jeps/142)（即关于增加@Contended的提议）的邮件讨论组中的描述（<http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007309.html>），其中的讲解是非常详细的（由于该讨论发生在@Contended实现的最初阶段，不能保证和现在的实现完全一致）， 我摘录和翻译如下：

*@Contended注释的行为如下所示：*

*A，在类上应用Contended：*

@Contended

public static class ContendedTest2 {

private Object plainField1;

private Object plainField2;

private Object plainField3;

private Object plainField4;

}

*将使整个字段块的两端都被填充：（以下是使用 –XX:+PrintFieldLayout的输出）（翻译注：注意前面的@140表示字段在类中的地址偏移）*

TestContended$ContendedTest2: field layout

Entire class is marked contended

@140 --- instance fields start ---

@140 "plainField1" Ljava.lang.Object;

@144 "plainField2" Ljava.lang.Object;

@148 "plainField3" Ljava.lang.Object;

@152 "plainField4" Ljava.lang.Object;

@288 --- instance fields end ---

@288 --- instance ends ---

*注意，我们使用了128bytes的填充 -- 2倍于大多数硬件缓存行的大小 -- 来避免相邻扇区预取导致的伪共享冲突。*

B，在字段上应用Contended：

public static class ContendedTest1 {

@Contended

private Object contendedField1;

private Object plainField1;

private Object plainField2;

private Object plainField3;

private Object plainField4;

}

*将导致该字段从连续的字段块中分离开来并高效的添加填充：*

TestContended$ContendedTest1: field layout

@ 12 --- instance fields start ---

@ 12 "plainField1" Ljava.lang.Object;

@ 16 "plainField2" Ljava.lang.Object;

@ 20 "plainField3" Ljava.lang.Object;

@ 24 "plainField4" Ljava.lang.Object;

@156 "contendedField1" Ljava.lang.Object; (contended, group = 0)

@288 --- instance fields end ---

@288 --- instance ends ---

*C, 注解多个字段使他们分别被填充：*

public static class ContendedTest4 {

@Contended

private Object contendedField1;

@Contended

private Object contendedField2;

private Object plainField3;

private Object plainField4;

}

*被注解的2个字段都被独立地填充：*

TestContended$ContendedTest4: field layout

@ 12 --- instance fields start ---

@ 12 "plainField3" Ljava.lang.Object;

@ 16 "plainField4" Ljava.lang.Object;

@148 "contendedField1" Ljava.lang.Object; (contended, group = 0)

@280 "contendedField2" Ljava.lang.Object; (contended, group = 0)

@416 --- instance fields end ---

@416 --- instance ends ---

*在有些cases中，你会想对字段进行分组，同一组的字段会和其他字段有访问冲突，但是和同一组的没有。例如，（同一个线程的）代码同时更新2个字段是很常见的情况。如果同时把2个字段都添加@Contended注解是足够的（翻译注：但是太足够了），但我们可以通过去掉他们之间的填充，来优化它们的内存空间占用。为了区分组，我们有一个参数“contention group”来描述：*

*所以：*

public static class ContendedTest5 {

@Contended("updater1")

private Object contendedField1;

@Contended("updater1")

private Object contendedField2;

@Contended("updater2")

private Object contendedField3;

private Object plainField5;

private Object plainField6;

}

内存布局是：

TestContended$ContendedTest5: field layout

@ 12 --- instance fields start ---

@ 12 "plainField5" Ljava.lang.Object;

@ 16 "plainField6" Ljava.lang.Object;

@148 "contendedField1" Ljava.lang.Object; (contended, group = 12)

@152 "contendedField2" Ljava.lang.Object; (contended, group = 12)

@284 "contendedField3" Ljava.lang.Object; (contended, group = 15)

@416 --- instance fields end ---

@416 --- instance ends ---

*注意$contendedField1 和$contendedField2和其他字段之间有填充，但是它们之间是紧挨着的。*

以上是对邮件组中大牛们原始实现解释的翻译。

下面我们来做一个测试，看@Contended在字段级别，并且带分组的情况下，是否能解决伪缓存问题。

import sun.misc.Contended;

public class VolatileLong {

@Contended("group0")

public volatile long value1 = 0L;

@Contended("group0")

public volatile long value2 = 0L;

@Contended("group1")

public volatile long value3 = 0L;

@Contended("group1")

public volatile long value4 = 0L;

}

我们用2个线程来修改字段--

测试1：线程0修改value1和value2；线程1修改value3和value4；他们都在同一组中。

测试2：线程0修改value1和value3；线程1修改value2和value4；他们在不同组中。

测试1：

public final class FalseSharing **implements** Runnable **{**

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

private static VolatileLong volatileLong**;**

private String groupId**;**

public FalseSharing**(**String groupId**)** **{**

**this.**groupId **=** groupId**;**

**}**

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

// Thread.sleep(10000);

System**.**out**.**println**(**"starting...."**);**

volatileLong **=** **new** VolatileLong**();**

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

runTest**();**

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

**}**

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

Thread t0 **=** **new** Thread**(new** FalseSharing**(**"t0"**));**

Thread t1 **=** **new** Thread**(new** FalseSharing**(**"t1"**));**

t0**.**start**();**

t1**.**start**();**

t0**.**join**();**

t1**.**join**();**

**}**

public void run**()** **{**

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

**if** **(**groupId**.**equals**(**"t0"**))** **{**

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

volatileLong**.**value1 **=** i**;**

volatileLong**.**value2 **=** i**;**

**}**

**}** **else** **if** **(**groupId**.**equals**(**"t1"**))** **{**

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

volatileLong**.**value3 **=** i**;**

volatileLong**.**value4 **=** i**;**

**}**

**}**

**}**

**}**

测试2：（基于以上代码修改下面的部分）

public void run**()** **{**

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

**if** **(**groupId**.**equals**(**"t0"**))** **{**

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

volatileLong**.**value1 **=** i**;**

volatileLong**.**value3 **=** i**;**

**}**

**}** **else** **if** **(**groupId**.**equals**(**"t1"**))** **{**

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

volatileLong**.**value2 **=** i**;**

volatileLong**.**value4 **=** i**;**

**}**

**}**

**}**

测试1：

starting....

duration = 16821484056

测试2：

starting....

duration = 39191867777

可以看出，如果同一线程修改的是同一“contention group”中的字段，没有伪共享冲突，比有伪共享冲突的情况要快1倍多。

后记：

测试3：不使用@Contended

public class VolatileLong **{**

public volatile long value1 **=** 0L**;**

public volatile long value2 **=** 0L**;**

public volatile long value3 **=** 0L**;**

public volatile long value4 **=** 0L**;**

**}**

结果：

starting....

duration = 38096777198

参考：

<http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/8-b132/sun/misc/Contended.java>

<http://openjdk.java.net/jeps/142>

<http://mail.openjdk.java.net/pipermail/hotspot-dev/2012-November/007309.html>