聊聊并发（一）深入分析Volatile的实现原理

本文属于作者原创，原文发表于InfoQ：http://www.infoq.com/cn/articles/ftf-java-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。

参考资料

- JVM执行篇：使用HSDIS插件分析JVM代码执行细节
- 内存屏障和并发
- Intel 64和IA-32架构软件开发人员手册

About . Latest Posts

方 腾飞

花名清英，并发网创始人，蚂蚁金服技术专家。目前工作于支付宝微贷事业部，关注互联网金融，并发编程和敏捷实践。

Related Posts:

发表评论 RSS订阅评论 Trackback (1) 评论 (20)

BlueIceQ
2013/01/27 11:22上午

登录以回复 | 引用

有些地方不太明白，上文提到："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"，这里是不是要有一个前提，就是头节点和尾节点的字节数之和也不超过64字节，因为一个缓存行的大小就是64字节，超过了一个缓存行的大小了。

方 腾飞
2013/01/27 11:33下午

登录以回复 | 引用

头节点和尾节点的字节数肯定不超过64的。一个对象的引用占4个字节，头＋尾节电才占8字节。

surest
2013/10/17 1:53下午

登录以回复 | 引用

上文提到"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"，但在HotSpot VM中每个对象的对象头占用8byte，这样算来一个PaddedAtomicReference对象占用了70byte的大小，超过了64byte的cache line大小；这样是否会导致 head 和 tail 共享 cahce line ?

anonymous
2013/10/30 4:57下午

登录以回复 | 引用

这里的head和tail其实是头和尾的索引位置

Tuscany
2013/02/22 4:35下午

登录以回复 | 引用

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。

~~~~~~

文中的jdk7是指哪个版本？

我机器上jdk1.7.0_15 x86_64并没有看到文中所说的 PaddedAtomicReference

方 腾飞
2013/02/22 4:38下午

登录以回复 | 引用

未公开的版本，现在的版本应该已经去掉了。因为Java7做了优化。

Tuscany
2013/02/22 4:45下午

登录以回复 | 引用

做了哪些优化呢？
~~~~~~

除了cache line padding之外还有哪些方法能解决false sharing？

多谢！

方 腾飞
2013/02/22 5:01下午

登录以回复 | 引用

见 http://ifeve.com/false-shareing-java-7-cn/

Tuscany
2013/02/22 5:10下午

引用

这里面说JDK7会有优化掉代码中没有使用的代码，所以呢自己加的padding部分就被"优化"掉了，不能达到cache line padding的目的，所以需要再加个"无用"的方法以免cache line padding部分被优化掉。

那也就是说jdk7的优化是针对无用代码的，而不是针对cache line padding的。

方 腾飞
2013/02/22 5:12下午

引用

是的。

逍遥K杰
2013/04/14 2:25下午

登录以回复 | 引用

Volatile在多线程下不能保证变量的原子性，并且会产生脏数据，那么能线程间共享又有啥意义呢

方 腾飞
2013/04/14 11:08下午

登录以回复 | 引用

结合CAS算法一起使用可以保证原子性。

sun shanghai
2013/04/26 10:56上午

登录以回复 | 引用

请教一个问题， 似乎不用volatile也能及时的获得变量被更新后的值呢。 我测试是这样的， 不大理解，特请教。

```
import java.lang.*;
/**
* 我觉得如果不加volatile修饰符，则第1个线程是永远不会停止的，
* 但是实际上线程2把flag设置为true后，线程1就停止了
* 这个是什么原因呢？不明白。
*/
public class Counter {
//
private static Boolean flag = new Boolean(false);

// private volatile static Boolean flag = new Boolean(false);

public static void main(String[] args) {

// 线程1
new Thread() {
int i = 0;

public void run() {
while (!flag.booleanValue()) {
System.out.println(i++);
}
}
}.start();

// 线程2
new Thread() {
public void run() {
try {
Thread.sleep(1000);
flag = new Boolean(true);
} catch (InterruptedException e) {
e.printStackTrace();
```

```
}
}
}.start();
}
}
```

丁一
2013/04/28 9:55下午

登录以回复 | 引用

可见性的具体实例可参见这篇：http://ifeve.com/concurrency-visibility/

G
2014/03/31 11:36上午

登录以回复 | 引用

volatile在编译器的实现上是否分为强弱两种方式，一种是明确变量是内存变量而不是寄存器变量；另外一种则表明是强制的内存变量每次都要从内存中读取不允许使用缓存。

而c语言中（centos gcc）下的volatile变量的操作则不会带有lock前缀，不知道这个情况和文中所说是否矛盾？

匿名
2014/07/01 6:00下午

登录以回复 | 引用

请教个问题，看了两篇关于伪共享的文章，这篇中提到的是填补15个引用类型已达到4*15 + 1*4 = 64目的，另外一篇文章中http://ifeve.com/false-shareing-java-7-cn/使用的确是6个long类型来进行填充，理由是每个对象都有一个对象头。请问这两种方式哪种是正确的还是说有适用范围的？

sunny
2014/09/22 5:01下午

登录以回复 | 引用

我觉得value的2头是不是都需要Padding?否则还是可能被伪共享？

例如

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

volatile v;

Object p10, p11, p12, p13, p14, p15, p16, p17, p18, p19, p1a, p1b, p1c, p1d, p1e;

yangzihao
2014/10/08 8:19下午

登录以回复 | 引用

请问作者，synchronized和锁之间的关系解释的很模糊啊

方腾飞
2014/10/08 8:23下午

登录以回复 | 引用

看下这篇文章能否回答你的问题 http://ifeve.com/java-synchronized/

viviju1989
2015/06/01 6:13下午

登录以回复 | 引用

问个问题，在文章中有这么两处提到了volatile 可见性的底层实现机制：