第 2 章　Java 并发机制的底层实现原理

Java 代码在编译后会变成 Java 字节码，字节码被类加载器加载到 JVM 里，JVM 执行字节码，最终需要转化为汇编指令在 CPU 上执行，Java 中所使用的并发机制依赖于 JVM 的实现和 CPU 的指令。本章我们将深入底层一起探索下 Java 并发机制的底层实现原理。

2.1　volatile 的应用

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。如果 volatile 变量修饰符使用恰当的话，它比 synchronized 的使用和执行成本更低，因为它不会引起线程上下文的切换和调度。本文将深入分析在硬件层面上 Intel 处理器是如何实现 volatile 的，通过深入分析帮助我们正确地使用 volatile 变量。我们先从了解 volatile 的定义开始。

2.1.1.volatile 的定义与实现原理

Java 语言规范第 3 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地更新，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，在某些情况下比锁要更加方便。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。

在了解 volatile 实现原理之前，我们先来看下与其实现原理相关的 CPU 术语与说明。表 2-1 是 CPU 术语的定义。

术　　语	英文单词	术语描述
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，而不是从内存读取
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 是如何来保证可见性的呢？让我们在 X86 处理器下通过工具获取 JIT 编译器生成的汇编指令来查看对 volatile 进行写操作时，CPU 会做什么事情。

Java 代码如下。

```
instance = new Singleton();                    // instance 是 volatile 变量
```

转变成汇编代码，如下。

```
0x01a3de1d: movb $0×0,0×1104800(%esi);0x01a3de24: lock addl $0×0,(%esp);
```

有 volatile 变量修饰的共享变量进行写操作的时候会多出第二行汇编代码，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[1]。

- 将当前处理器缓存行的数据写回到系统内存。
- 这个写回内存的操作会使在其他 CPU 里缓存了该内存地址的数据无效。

为了提高处理速度，处理器不直接和内存进行通信，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，但操作完不知道何时会写到内存。如果对声明了 volatile 的变量进行写操作，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，当处理器对这个数据进行修改操作的时候，会重新从系统内存中把数据读到处理器缓存里。下面来具体讲解 volatile 的两条实现原则。

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，处理器可以独占任何共享内存[2]。但是，在最近的处理器里，LOCK＃信号一般不锁总线，而是锁缓存，毕竟锁总线开销的比较大。在 8.1.4 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 和目前的处理器中，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，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由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。处理器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，而这个地址当前处于共享状态，那么正在嗅探的处理器将使它的缓存行无效，在下次访问相同内存地址时，强制执行缓存行填充。

2.1.2 volatile 的使用优化

著名的 Java 并发编程大师 Doug lea 在 JDK 7 的并发包里新增一个队列集合类 LinkedTransferQueue，它在使用 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。LinkedTransferQueue 的代码如下。

```
/**
 * 队列中的头部节点
 */
private transient final PaddedAtomicReference<QNode> head;
/**
 * 队列中的尾部节点
 */
private transient final PaddedAtomicReference<QNode> tail;

static final class PaddedAtomicReference<T> extends AtomicReference T>

{
    // 使用很多 4 个字节的引用追加到 64 个字节
   Object p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, pa, pb, pc, pd, pe;
    PaddedAtomicReference(T r) {
    super(r);
}
}

public class AtomicReference<V> implements java.io.Serializable {
    private volatile V value;
    // 省略其他代码
}
```

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节点（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，就是将共享变量追加到 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。

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，以及 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节点时，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则需要不停修改头节点和尾节点，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，避免头节点和尾节点加载到同一个缓存行，使头、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。

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。

共享变量不会被频繁地写。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，如果共享变量不被频繁写的话，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。

不过这种追加字节的方式在 Java 7 下可能不生效，因为 Java 7 变得更加智慧，它会淘汰或重新排列无用字段，需要使用其他追加字节的方式。除了 volatile，Java 并发编程中应用较多的是 synchronized，下面一起来看一下。

[1] *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 8 章）中有详细阐述。*

[2] *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。*

2.2 synchronized 的实现原理与应用

在多线程并发编程中 synchronized 一直是元老级角色，很多人都会称呼它为重量级锁。但是，随着 Java SE 1.6 对 synchronized 进行了各种优化之后，有些情况下它就并不那么重了。本文详细介绍 Java SE 1.6 中为了减少获得锁和释放锁带来的性能消耗而引入的偏向锁和轻量级锁，以及锁的存储结构和升级过程。

先来看下利用 synchronized 实现同步的基础：Java 中的每一个对象都可以作为锁。具体表现为以下 3 种形式。

- 对于普通同步方法，锁是当前实例对象。
- 对于静态同步方法，锁是当前类的 Class 对象。
- 对于同步方法块，锁是 Synchonized 括号里配置的对象。

当一个线程试图访问同步代码块时，它首先必须得到锁，退出或抛出异常时必须释放锁。那么锁到底存在哪里呢？锁里面会存储什么信息呢？