参考：<http://blog.csdn.net/u011116672/article/details/50098791>

[**Java并发编程系列之一：并发机制的底层原理**](http://blog.csdn.net/u011116672/article/details/50098791)

版权声明：本文为博主原创文章，转载请注明出处。

**前言**

并发编程的目的是让程序运行更快（me：这句话不是特别的准确，应该是使得程序执行的效率更高），但是使用并发并不定会使得程序运行更快，只有当程序的并发数量达到一定的量级的时候才能体现并发编程的优势。所以谈并发编程在高并发量的时候才有意义。虽然目前还没有开发过高并发量的程序，但是学习并发是为了更好理解一些分布式架构。那么当程序的并发量不高，比如是单线程的程序，单线程的执行效率反而比多线程更高。这又是为什么呢？熟悉操作系统的应该知道，CPU是通过给每个线程分配时间片的方式实现多线程的。这样，当CPU从一个任务切换到另一个任务的时候，会保存上一个任务的状态，当执行完这个任务的时候CPU就会继续上一个任务的状态继续执行。这个过程称为**上下文切换**。

在Java多线程中，volatile关键字和synchronized关键字扮演了重要的角色，它们都可以实现线程的同步，但是在底层是如何实现的呢？

**volatile关键字**

我们知道，Java虚拟机把Java源文件编译成.class文件，然后把class文件加载到内存中，JVM执行字节码，最终转化为汇编指令在CPU上执行。

volatile定义：

Java编程语言允许线程访问共享变量，为了确保共享变量能够准确一致地更新，线程确保通过排他锁单独获得这个变量。

Java中的volatile关键字就是这个定义的体现。如果一个变量被声明为volatile，那么确保这个变量是“可见的”。可见性的意思是**当线程修改一个共享变量的时候，另外一个线程能够读到这个修改的值**。由于底层的实现涉及处理器（这里以X86处理器作为举例），所以有必要先了解一些CPU术语：

| **术语** | **描述** |
| --- | --- |
| 内存屏障 | 是一组处理器指令，用于实现内存操作的顺序限制 |
| 缓存行 | 缓存中可以分配的最小存储单位 |
| 原子操作 | 不可中断的一系列操作 |
| 缓存行填充 | 当处理器识别到从内存中读取的操作数是可缓存的，处理器读取整个缓存行到适当的缓存 |

下面，volatile是如何保证可见性的呢？当一个变量被声明为volatile的时候，底层（更具体是处理器）会帮我们完成下面的事情，有两句话很重要：

1. **将当前处理器缓存行的数据写回到系统内存中**
2. **这个写回到内存的数据会使得其他CPU（比如是多核处理器）里缓存了该内存地址的数据无效**

为了提高处理速度，保证内存可见性的实现，CPU之间会使用缓存一致性协议，具体讲就是：如果对声明为volatile关键字的变量进行写操作，那么JVM就会向处理器发送一条Lock前缀的指令，从而将缓存行中的数据写回到系统内存中。是不是这样就够了呢？因为即使写回到系统内存中，而其他处理器仍然使用其缓存中的数据怎么办，所以每个处理器会通过嗅探技术在总线上传播数据来检查自己缓存中数据是不是最新的，如果发现不是最新的就把缓存行设置为无效，处理器下次就会直接从系统内存中取数据，因为系统内存中数据是最新的。

* 如何让缓存行中数据写回到系统内存中？之前的处理器是通过**锁总线**的方式独占共享内存，这样做的缺点很明显：导致其他处理器不能访问总线，这样就不能访问内存。现在的处理器已经是使用**缓存锁定**的方式来保证原子性，具体就是：如果执行的写操作的内存区域已经缓存在处理器的内部，就会锁定这块内存区域的缓存并写回到内存中，并使用缓存一致性协议确保修改的原子性。
* 怎样使得其他处理器缓存行中的数据无效？简要概括就是**MESI（修改、独占、共享、无效）**。这是一个控制协议，可以维护内部缓存和其他处理器缓存的一致性。一个处理器通过嗅探技术得知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（意味着其他处理可以访问），那么正在嗅探的处理器会把其缓存行设置为无效。

**synchronized关键字**

使用synchronized关键字有以下三种使用方式：

1. 同步代码块
2. 同步方法
3. 静态同步方法

对于同步代码块锁的是括号里面的配置对象，同步方法锁的是当前实例对象，静态同步方法锁的是当前类的Class对象。JVM通过一个**监视器对象**实现代码块的同步，对应的指令是monitorenter和monitorexit，方法的同步使用的另一种机制，但仍然使用的是监视器对象来实现同步的。具体是这样的：monitorenter指令插入到同步代码块中的开始位置，而monitorexit指令则被插入到方法结束处和异常处。要注意的是，任何对象都有一个监视器对象与之关联，当一个监视器对象被持有的时候，该对象将处于锁定状态。当线程执行到monitorenter指令的时候，将会尝试获取对象所对应的**监视器对象的所有权，也就是线程的锁**。

**锁的使用**

前面提到了锁的概念，那么锁有哪些呢？在JDK 1.6中引入了“偏向锁”和“轻量级锁“。锁一共有四种状态：无锁、偏向锁、轻量级锁和重量级锁。锁只能升级，不能降级。当对锁的竞争加剧的时候，锁会发生升级。

**1.偏向锁**

之所以引入偏向锁，是为了让线程获得锁的代价更低。当一个线程访问同步块并获取锁的时候，会在对象的对象头（对象头包括两部分的信息：一部分是”Mark Word“，主要存放的是哈希码、对象的分代年龄、锁的标记等信息；另一部分是对象的类型指针）和栈帧中的锁记录中存储锁偏向的ID，以后该线程在进入方法的同步块的时候，就检查这个ID（可以理解为一种标记，是一种身份的标识），如果测试成功，表明对象已经获得了锁；如果测试失败，继续测试偏向锁的标识是否设置为1（1的话就是偏向锁），如果没有则使用CAS（Compare And Swap）锁。

**2.轻量级锁**

分为加锁和解锁。当线程执行到同步块之前，JVM会首先检查当前线程的栈帧中创建用于存储记录锁记录的空间，并将对象头中Mark Word复制到锁记录中，也称为Displaced Mark Word，然后线程尝试使用CAS将对象头中的Mark Word替换为指向锁记录的指针。如果成功，则线程获得锁，否则当前线程尝试使用自旋来获取锁。这就是加锁的过程。

这里多次提到CAS，那么CAS是个什么鬼？CAS是Compare and swap（比较和替换）的简写，具体而言就是：当进行CAS操作的时候，需要输入两个数值，一个是旧值，该旧值是原来的值，另一个是新值，也就是发生改变的值，得到这两个值后，在CAS操作期间会去比较旧值是否发生变化，如果没有发生变化就用新值进行替换，如果发生了变化就不进行替换。

那么解锁的过程又是怎样的呢？就是使用CAS操作将Displaced Mark Word替换回对象头，如果成功，则表示没有竞争发生。如果失败，表示当前锁存在竞争，锁就会膨胀，膨胀的结果是导致锁的升级，并进入阻塞状态。直到需要释放锁的线程释放锁并唤醒其他等待的线程。

**锁的使用场景**   
由于偏向锁在线程存在竞争的时候会带来额外的性能开销，所以偏向锁适用于只有一个线程方法同步快的情况；轻量级锁在线程竞争锁的情况下不会导致线程阻塞，但是会通过自旋消耗CPU，所以轻量级锁适用于追求响应时间的情况。重量级锁线程竞争不会使用自旋，但是线程竞争会导致阻塞，所以响应时间比较慢，重量级锁一般使用在追求吞吐量的情况。