

郑州的文武

世界需要变得更美好

博客园

首页

订阅

~ Java中Volatile关键字详解

阅读目录

- 一、基本概念
- 二、Volatile原理

一、基本概念

先补充一下概念：Java 内存模型中的可见性、原子性和有序性。

可见性：

可见性是一种复杂的属性，因为可见性中的错误总是会违背我们的直觉。通常，我们无法确保执行读操作的线程能适时地看到其他线程写入的值，有时甚至是根本不可能的事情。为了确保多个线程之间对内存写入操作的可见性，必须使用同步机制。

可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的。也就是一个线程修改的结果。另一个线程马上就能看到。比如：用volatile修饰的变量，就会具有可见性。volatile修饰的变量不允许线程内部缓存和重排序，即直接修改内存。所以对其他线程是可见的。但是这里需要注意一个问题，volatile只能让被他修饰内容具有可见性，但不能保证它具有原子性。比如 volatile int a = 0；之后有一个操作 a++；这个变量a具有可见性，但是a++ 依然是一个非原子操作，也就是这个操作同样存在线程安全问题。

在 Java 中 volatile、synchronized 和 final 实现可见性。

原子性：

原子是世界上的最小单位，具有不可分割性。比如 a=0；（a非long和double类型）这个操作是不可分割的，那么我们说这个操作时原子操作。再比如：a++；这个操作实际是a = a + 1；是可分割的，所以他不是一个原子操作。非原子操作都会存在线程安全问题，需要我们使用同步技术（synchronized）来让它变成一个原子操作。一个操作是原子操作，那么我们称它具有原子性。java的concurrent包下提供了一些原子类，我们可以通过阅读API来了解这些原子类的用法。比如：AtomicInteger、AtomicLong、AtomicReference等。

在 Java 中 synchronized 和在 lock、unlock 中操作保证原子性。

有序性：

Java 语言提供了 `volatile` 和 `synchronized` 两个关键字来保证线程之间操作的有序性，`volatile` 是因为其本身包含“禁止指令重排序”的语义，`synchronized` 是由“一个变量在同一个时刻只允许一条线程对其进行 `lock` 操作”这条规则获得的，此规则决定了持有同一个对象锁的两个同步块只能串行执行。

下面内容摘录自《Java Concurrency in Practice》：

下面一段代码在多线程环境下，将存在问题。

[+ View code](#)

```
1 /**
2  * @author zhengbinMac
3  */
4 public class NoVisibility {
5     private static boolean ready;
6     private static int number;
7     private static class ReaderThread extends Thread {
8         @Override
9         public void run() {
10             while(!ready) {
11                 Thread.yield();
12             }
13             System.out.println(number);
14         }
15     }
16     public static void main(String[] args) {
17         new ReaderThread().start();
18         number = 42;
19         ready = true;
20     }
21 }
```



`NoVisibility`可能会持续循环下去，因为读线程可能永远都看不到`ready`的值。甚至`NoVisibility`可能会输出0，因为读线程可能看到了写入`ready`的值，但却没有看到之后写入`number`的值，这种现象被称为“重排序”。只要在某个线程中无法检测到重排序情况（即使在其他线程中可以明显地看到该线程中的重排序），那么就无法确保线程中的操作将按照程序中指定的顺序来执行。当主线程首先写入`number`，然后在没有同步的情况下写入`ready`，那么读线程看到的顺序可能与写入的顺序完全相反。

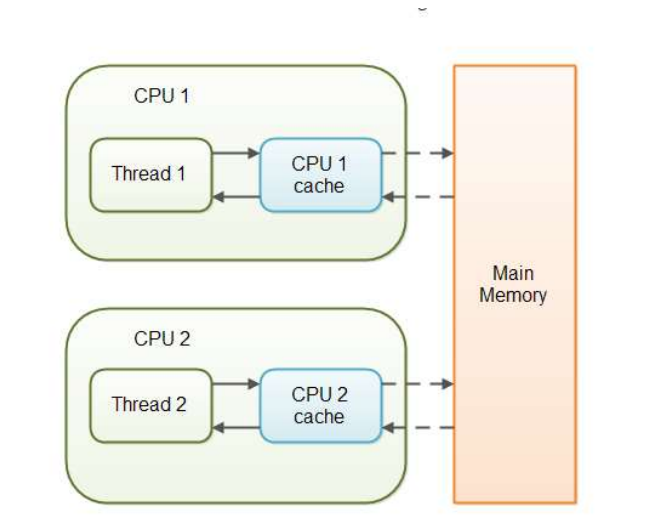
在没有同步的情况下，编译器、处理器以及运行时等都可能对操作的执行顺序进行一些意想不到的调整。在缺乏足够同步的多线程程序中，要想对内存操作的执行顺序进行判断，无法得到正确的结论。

这个看上去像是一个失败的设计，但却能使JVM充分地利用现代多核处理器的强大性能。例如，在缺少同步的情况下，Java内存模型允许编译器对操作顺序进行重排序，并将数值缓存在寄存器中。此外，它还允许CPU对操作顺序进行重排序，并将数值缓存在处理器特定的缓存中。

二、Volatile原理

Java语言提供了一种稍弱的同步机制，即volatile变量，用来确保将变量的更新操作通知到其他线程。当把变量声明为volatile类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile变量不会被缓存在寄存器或者对其他处理器不可见的地方，因此在读取volatile类型的变量时总会返回最新写入的值。

在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，因此volatile变量是一种比synchronized关键字更轻量级的同步机制。



当对非 volatile 变量进行读写的时候，每个线程先从内存拷贝变量到CPU缓存中。如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这意味着每个线程可以拷贝到不同的 CPU cache 中。

而声明变量是 volatile 的，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步。

当一个变量定义为 volatile 之后，将具备两种特性：

1. 保证此变量对所有的线程的可见性，这里的“可见性”，如本文开头所述，当一个线程修改了这个变量的值，volatile 保证了新值能立即同步到主内存，以及每次使用前立即从主内存刷新。但普通变量做不到这点，普通变量的值在线程间传递均需要通过主内存（详见：Java内存模型）来完成。


2. 禁止指令重排序优化。有volatile修饰的变量，赋值后多执行了一个“load addl \$0x0, (%esp)”操作，这个操作相当于一个内存屏障（指令重排序时不能把后面的指令重排序到内存屏障之前的位置），只有一个CPU访问内存时，并不需要内存屏障；（什么是指令重排序：是指CPU采用了允许将多条指令不按程序规定的顺序分开发送给各相应电路单元处理）。

volatile 性能：

volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行。

梦想要一步步来！

分类: Java基础, 备战阿里, 并发

 郑州的文武

关注 - 31

粉丝 - 63

+加关注

« 上一篇：Java自增原子性问题（测试Volatile、AtomicInteger）

» 下一篇：乐观锁与悲观锁

posted @ 2016-07-08 22:50 郑州的文武 阅读(15194) 评论(11) 收藏

评论列表

#1楼 2016-08-11 10:38 、soul。小叶子

浅显易懂

支持(1) 反对(0)

#2楼[楼主] 2016-09-04 22:31 郑州的文武

@ 、soul。小叶子

谢谢~

支持(0) 反对(0)

#3楼 2017-02-09 11:39 tancp

我喜欢简洁通俗描述问题的语言，而你这篇文章的语言正是我喜欢的这种

支持(1) 反对(0)

#4楼[楼主] 2017-02-09 11:53 郑州的文武

@ tancp

感谢支持😊

支持(0) 反对(0)

#5楼 2017-02-21 11:41 Gamehu

3q

支持(1) 反对(0)

#6楼 2017-02-21 16:21 Bazinga~

请问我遇到一个问题，首先你这段代码运行的时候 基本不会有死循环情况出现，因为主线程优先级一般比较高，在子线程启动前ready就已经被赋值为true了。

所以为了出现更好的效果 我在线程start后，加上一行Thread.sleep(10)

但是还是不会出现你说的死循环的情况，按理说会出现的才对

然而如果把 Thread.yield() 去掉，就会有死循环出现。

这是什么原因，请博主赐教

支持(1) 反对(0)

#7楼[楼主] 2017-02-21 19:05 郑州的文武

@ Bazinga~

非常感谢能指出问题，当时运行代码也发现存在这个问题，但没有深入下去。

1) “因为主线程优先级一般比较高”：这个JavaAPI文档中有解释：‘新创建线程的优先级被设定为创建该线程的线程的优先级，即当前正在运行的线程的优先级。’，也就是说主线程和子线程优先级是相同的，并且新创建的线程优先级默认都为NORM_PRIORITY (5)。

2) “没有出现死循环”：如果你在 while 循环之前和循环里输出 ready 的值，多次运行，就会发现存在循环外输出 ready 的值为false，循环里输出也为 false，但是接下来的 yield，会让当前线程暂停，让其他线程开始执行，也就是暂停子线程，继续主线程，这时主线程将 ready 赋值为 true，最终也会结束程序运行；

但是正如你所说，如果将 yield 注释掉，多次运行，观察加入注释的输出结果，会发现循环里的输出确实输出了很多次，但是最终还是会跳出循环，也就是说子线程获得了主线程修改的最新 ready 值。根据 Java 内存模型，子线程确实从主内存获取到了最新的值。

关于重排序，这个例子确实没有说明清楚重排序是个什么情况，研究透彻后，我会再修改的。

支持(0) 反对(0)

#8楼 2017-03-09 15:14 张文浩

我可不可以这样理解：

不使用volatile时，变量赋值需要读取、赋值、写回，赋值改变的是读取到工作内存的一个拷贝，需要写回到主内存之后其他线程才能看见这个新的值；

加了volatile，（赋值+写回）是原子的，直接会改变主内存的值，中间不会插入其他线程的操作，是安全的，所以一旦赋值其他线程立马就能看见。

volatile之所以不能线程安全，是因为虽然读取的是新的值，是线程安全的，但如果读完之后，其他线程改变了这个值，该线程不会重新读取，依然用的是之前读取的值，从而出现线程同步问题。

支持(0) 反对(0)

#9楼[楼主] 2017-03-09 16:28 郑州的文武

@ 张文浩

下面先指出，我对你以上三个结论存在不同见解的

第2个：“中间不会插入其他线程的操作”、“所以一旦赋值其他线程立马就能看见”，结论是对的，但是

引出结论的原因是错误的。

第3个：“但如果读完之后，其他线程改变了这个值，该线程不会重新读取，依然用的是之前读取的值，从而出现线程同步问题。”，描述不太完整。

下面重新再阐述下volatile域所遵循的规则：

- 1.线程所见的值在使用之前总会从主内存中读出来；
- 2.线程所写的值总会在指令完成之前被刷回到主内存中；

现在再看你的第2个结论，并不是“中间不会插入其他线程的操作”，volatile变量在各个线程的工作内存中存在不一致性的问题，但是，在每次使用前都要刷新，所以执行引擎并看不到不一致的情况，所以可以认为不存在不一致性问题。以上两个规则，保证了可见性。

再看第3个结论，我再补充一些。不安全，指的是Java运算并非原子操作，这是导致volatile变量的运算在并发下不安全的原因。比如“i++”操作，生成四个字节码指令：getstatic, iconst, iadd, putstatic, volatile只保证了getstatic将i值读取到操作栈顶时i值是正确的，但是进行后三个操作前，其他线程可能进行了i++操作，操作栈顶的i值就是无效的了，这时再putstatic，就产生了线程不安全。

支持(0) 反对(0)

#10楼 2017-05-02 16:29 框架搬运工

这个例子是摘自Java并发实战的例子，我看过一点这本书，对这个例子有印象。坦白说，可能我对CPU的底层原理不清楚，所以我一直到现在也明白这个例子的意义在哪里？

主线程在读取线程启动后，将ready设置为true，并且为number赋值。按作者的意思，我理解的是：作者想说主线程还未修改ready的值，读取线程已经在执行了，此时ready为false，所以会无限的循环下去。

可是代码测试之后发现并不是这样，我加了一个属性count用来记录while循环执行的次数，然后在读取线程启动后，让主线程休眠100毫秒。此时ready的值肯定为false，那么循环开始，当100毫秒过后，ready值被改为true了，读取线程也停止了，我输出count大约在6000-9000之间。也就是说100毫秒的休眠循环执行了7000次左右。之后ready的值变为true，读取线程就输出number退出了。并没有无限循环。

同样我让主线程1000毫秒，count大概在11万左右。区别也只是while多执行了N次的空循环而已。

但是依然会在某个点读取到ready为true，并且正确退出。完全不会有无限循环的事情发生。那么我想问问，这个例子到底在说明什么？

按我个人的理解，其实就是一个自定义线程，一个主线程，主线程改变了某个属性，而自定义线程中的循环依赖这个属性值，那么主线程在执行完毕更改操作后，将这个值写回内存中，

而在值被写回内存后，自定义线程就能读取到被更改后的值了。所以不会有死循环产生。不知道博主对这个例子是如何看待的。那本书中有的例子真的看的是一头雾水不明白

如果博主有其他理解，还望指出。谢谢了。

支持(1) 反对(0)

8

0

#11楼 2017-05-10 10:18 骑着单车的程序猿

@ 框架搬运工

引用

这个例子是摘自Java并发实战的例子，我看过一点这本书，对这个例子有印象。坦白说，可能我对CPU的底层原理不清楚，所以我一直到现在也明白这个例子的意义在哪里？

主线程在读取线程启动后，将ready设置为true，并number赋值。按作者的意思，我理解的是：作者想说主线程还未修改ready的值，读取线程已经在执行了，此时ready为false，所以会无限的循环下

去。
可是代码测试之后发现并不是这样，我加了一个属性count用来记录while循环执行的次数，然后在读取线程启动后，让主线程休眠100毫秒。此时ready的值肯定为false，那么循环开始，当100毫秒过后，ready
值被改为t...

我也发现了，根本不会无限循环的结果出现，如果让main线程休眠100毫秒也只是让while循环多执行几次而已，最终main线程中改变的ready值还是能被子线程看到，最终退出执行！这个例子好像说明不了重排序这个问题！

支持(0) 反对(0)

[刷新评论](#) [刷新页面](#) [返回顶部](#)

注册用户登录后才能发表评论，请 [登录](#) 或 [注册](#)，[访问网站首页](#)。



8

0

