随笔 - 87  文章 - 1  评论 - 292

# [线程安全(上)--彻底搞懂volatile关键字](https://www.cnblogs.com/kubidemanong/p/9505944.html)

对于volatile这个关键字，相信很多朋友都听说过，甚至使用过，这个关键字虽然字面上理解起来比较简单，但是要用好起来却不是一件容易的事。  
这篇文章将从多个方面来讲解volatile，让你对它更加理解。

误解：volatile变量对所有线程是立即可见的，所以对volatile变量的所有修改(写操作)都立刻能反应到其他线程中。或者换句话说：volatile变量在各个线程中是一致的，所以基于volatile变量的运算在并发下是线程安全的。  
这个观点的论据是正确的，但是根据论据得出的结论是错误的，并不能得出这样的结论。volatile的规则，保证了read、load、use的顺序和连续行，同理assign、store、write也是顺序和连续的。也就是这几个动作是原子性的，但是对变量的修改，或者对变量的运算，却不能保证是原子性的。如果对变量的修改是分为多个步骤的，那么多个线程同时从主内存拿到的值是最新的，但是经过多步运算后回写到主内存的值是有可能存在覆盖情况发生的。

## **计算机中为什么会出现线程不安全的问题**

volatile既然是与线程安全有关的问题，那我们先来了解一下计算机在处理数据的过程中为什么会出现线程不安全的问题。  
大家都知道，计算机在执行程序时，每条指令都是在CPU中执行的，而执行指令过程中会涉及到数据的读取和写入。由于程序运行过程中的临时数据是存放在主存（物理内存）当中的，这时就存在一个问题，由于CPU执行速度很快，而从内存读取数据和向内存写入数据的过程跟CPU执行指令的速度比起来要慢的多，因此如果任何时候对数据的操作都要通过和内存的交互来进行，会大大降低指令执行的速度。  
为了处理这个问题，在CPU里面就有了高速缓存(Cache)的概念。当程序在运行过程中，会将运算需要的数据从主存复制一份到CPU的高速缓存当中，那么CPU进行计算时就可以直接从它的高速缓存读取数据和向其中写入数据，当运算结束之后，再将高速缓存中的数据刷新到主存当中。  
我举个简单的例子，比如cpu在执行下面这段代码的时候,

t = t + 1;

会先从高速缓存中查看是否有t的值，如果有，则直接拿来使用，如果没有，则会从主存中读取，读取之后会复制一份存放在高速缓存中方便下次使用。之后cup进行对t加1操作，然后把数据写入高速缓存，最后会把高速缓存中的数据刷新到主存中。

这一过程在单线程运行是没有问题的，但是在多线程中运行就会有问题了。在多核CPU中，每条线程可能运行于不同的CPU中，因此每个线程运行时有自己的高速缓存（对单核CPU来说，其实也会出现这种问题，只不过是以线程调度的形式来分别执行的，本次讲解以多核cup为主）。这时就会出现同一个变量在两个高速缓存中的**值**不一致问题了。  
例如：  
两个线程分别读取了t的值，假设此时t的值为0，并且把t的值存到了各自的高速缓存中，然后线程1对t进行了加1操作，此时t的值为1，并且把t的值写回到主存中。但是线程2中高速缓存的值还是0，进行加1操作之后，t的值还是为1，然后再把t的值写回主存。  
此时，就出现了线程不安全问题了。

## **Java中的线程安全问题**

上面那种线程安全问题，可能对于不同的操作系统会有不同的处理机制，例如Windows操作系统和Linux的操作系统的处理方法可能会不同。  
我们都知道，Java是一种夸平台的语言，因此Java这种语言在处理线程安全问题的时候，会有自己的处理机制，例如volatile关键字，synchronized关键字，并且这种机制适用于各种平台。  
Java内存模型规定所有的变量都是存在主存当中（类似于前面说的物理内存），每个线程都有自己的工作内存（类似于前面的高速缓存）。线程对变量的所有操作都必须在工作内存中进行，而不能直接对主存进行操作。并且每个线程不能访问其他线程的工作内存。  
由于java中的每个线程有自己的工作空间，这种工作空间相当于上面所说的高速缓存，因此多个线程在处理一个**共享变量**的时候，就会出现线程安全问题。

*这里简单解释下****共享变量****，上面我们所说的t就是一个共享变量，也就是说，能够被多个线程访问到的变量，我们称之为共享变量。在java中共享变量包括实例变量，静态变量，数组元素。他们都被存放在堆内存中。*

## **volatile关键字**

上面扯了一大堆，都没提到volatile关键字的作用，下面开始讲解volatile关键字是如何保证线程安全问题的。

#### **可见性**

##### **什么是可见性？**

意思就是说，在多线程环境下，某个共享变量如果被其中一个线程给修改了，其他线程能够立即知道这个共享变量已经被修改了，当其他线程要读取这个变量的时候，最终会去内存中读取，而不是从自己的**工作空间**中读取。  
例如我们上面说的，当线程1对t进行了加1操作并把数据写回到主存之后，线程2就会知道它自己工作空间内的t已经被修改了，当它要执行加1操作之后，就会去主存中读取。这样，两边的数据就能一致了。  
假如一个变量被声明为volatile，那么这个变量就具有了**可见性**的性质了。这就是volatile关键的作用之一了。

##### **volatile保证变量可见性的原理**

当一个变量被声明为volatile时，在编译成会变指令的时候，会多出下面一行：

0x00bbacde: lock add1 $0x0,(%esp);

这句指令的意思就是在寄存器执行一个加0的空操作。不过这条指令的前面有一个lock(锁)前缀。  
当处理器在处理拥有lock前缀的指令时：  
在之前的处理中，lock会导致传输数据的总线被锁定，其他处理器都不能访问总线，从而保证处理lock指令的处理器能够独享操作数据所在的内存区域，而不会被其他处理所干扰。  
但由于总线被锁住，其他处理器都会被堵住，从而影响了多处理器的执行效率。为了解决这个问题，在后来的处理器中，处理器遇到lock指令时不会再锁住总线，而是会检查数据所在的内存区域，如果该数据是在处理器的内部缓存中，则会锁定此缓存区域，处理完后把缓存写回到主存中，并且会利用**缓存一致性协议**来保证其他处理器中的缓存数据的一致性。

##### **缓存一致性协议**

刚才我在说可见性的时候，说“如果一个共享变量被一个线程修改了之后，当其他线程要读取这个变量的时候，最终会去内存中读取，而不是从自己的**工作空间**中读取”，实际上是这样的：  
线程中的处理器会一直在总线上嗅探其内部缓存中的内存地址在其他处理器的操作情况，一旦嗅探到某处处理器打算修改其内存地址中的值，而该内存地址刚好也在自己的内部缓存中，那么处理器就会强制让自己对该缓存地址的无效。所以当该处理器要访问该数据的时候，由于发现自己缓存的数据无效了，就会去主存中访问。

#### **有序性**

实际上，当我们把代码写好之后，虚拟机不一定会按照我们写的代码的顺序来执行。例如对于下面的两句代码：

int a = 1;int b = 2;

对于这两句代码，你会发现无论是先执行a = 1还是执行b = 2，都不会对a,b最终的值造成影响。所以虚拟机在编译的时候，是有可能把他们进行**重排序**的。  
为什么要进行重排序呢？  
你想啊，假如执行 int a = 1这句代码需要100ms的时间，但执行int b = 2这句代码需要1ms的时间，并且先执行哪句代码并不会对a,b最终的值造成影响。那当然是先执行int b = 2这句代码了。  
所以，虚拟机在进行代码编译优化的时候，对于那些改变顺序之后不会对最终变量的值造成影响的代码，是有可能将他们进行重排序的。  
更多代码编译优化可以看我写的另一篇文章：  
[虚拟机在运行期对代码的优化策略](https://mp.weixin.qq.com/s/NHBSv6AmpsKTZPmhj_sU6A)

那么重排序之后真的不会对代码造成影响吗？  
实际上，对于有些代码进行重排序之后，虽然对变量的值没有造成影响，但有可能会出现线程安全问题的。具体请看下面的代码

public class NoVisibility{

private static boolean ready;

private static int number;

private static class Reader extends Thread{

public void run(){

while(!ready){

Thread.yield();

}

System.out.println(number);

}

}

public static void main(String[] args){

new Reader().start();

number = 42;

ready = true;

}

}

这段代码最终打印的一定是42吗？如果没有重排序的话，打印的确实会是42，但如果number = 42和ready = true被进行了重排序，颠倒了顺序，那么就有可能打印出0了，而不是42。（因为number的初始值会是0).  
因此，重排序是有可能导致线程安全问题的。

如果一个变量被声明volatile的话，那么这个变量不会被进行重排序，也就是说，虚拟机会保证这个变量之前的代码一定会比它先执行，而之后的代码一定会比它慢执行。  
例如把上面中的number声明为volatile，那么number = 42一定会比ready = true先执行。

*不过这里需要注意的是，虚拟机只是保证这个变量之前的代码一定比它先执行，但并没有保证这个变量之前的代码不可以重排序。之后的也一样。*

volatile关键字能够保证代码的**有序性**，这个也是volatile关键字的作用。  
总结一下，一个被volatile声明的变量主要有以下两种特性保证保证线程安全。

1. 可见性。
2. 有序性。

#### **volatile真的能完全保证一个变量的线程安全吗？**

我们通过上面的讲解，发现volatile关键字还是挺有用的，不但能够保证变量的可见性，还能保证代码的有序性。  
那么，它真的能够保证一个变量在多线程环境下都能被正确的使用吗？  
答案是否定的。原因是因为Java里面的运算并非是**原子操作**。

##### **原子操作**

**原子操作**：即一个操作或者多个操作 要么全部执行并且执行的过程不会被任何因素打断，要么就都不执行。  
也就是说，处理器要嘛把这组操作全部执行完，中间不允许被其他操作所打断，要嘛这组操作不要执行。  
刚才说Java里面的运行并非是原子操作。我举个例子，例如这句代码

int a = b + 1;

处理器在处理代码的时候，需要处理以下三个操作：

1. 从内存中读取b的值。
2. 进行a = b + 1这个运算
3. 把a的值写回到内存中

而这三个操作处理器是不一定就会连续执行的，有可能执行了第一个操作之后，处理器就跑去执行别的操作的。

##### **证明volatile无法保证线程安全的例子**

由于Java中的运算并非是原子操作，所以导致volatile声明的变量无法保证线程安全。  
对于这句话，我给大家举个例子。代码如下:

public class Test{

public static volatile int t = 0;

public static void main(String[] args){

Thread[] threads = new Thread[10];

for(int i = 0; i < 10; i++){

//每个线程对t进行1000次加1的操作

threads[i] new Thread(new Runnable(){

@Override

public void run(){

for(int j = 0; j < 1000; j++){

t = t + 1;

}

}

});

threads[i].start();

}

//等待所有累加线程都结束

while(Thread.activeCount() > 1){

Thread.yield();

}

//打印t的值

System.out.println(t);

}

}

最终的打印结果会是1000 \* 10 = 10000吗？答案是否定的。  
问题就出现在t = t + 1这句代码中。我们来分析一下  
例如：  
线程1读取了t的值，假如t = 0。之后线程2读取了t的值，此时t = 0。  
然后线程1执行了加1的操作，此时t = 1。但是这个时候，处理器还没有把t = 1的值写回主存中。这个时候处理器跑去执行线程2，注意，刚才线程2已经读取了t的值，所以这个时候并不会再去读取t的值了，所以此时t的值还是0，然后线程2执行了对t的加1操作，此时t =1 。  
这个时候，就出现了线程安全问题了，两个线程都对t执行了加1操作，但t的值却是1。所以说，volatile关键字并不一定能够保证变量的安全性。

#### **什么情况下volatile能够保证线程安全**

刚才虽然说，volatile关键字不一定能够保证线程安全的问题，其实，在大多数情况下volatile还是可以保证变量的线程安全问题的。所以，在满足以下两个条件的情况下，volatile就能保证变量的线程安全问题：

1. 运算结果并不依赖变量的当前值，或者能够确保只有单一的线程修改变量的值。
2. 变量不需要与其他状态变量共同参与不变约束。

讲到这里，关于volatile关键字的就算讲完了。如果有哪里讲的不对的地方，非常欢迎你的指点。下篇应该会讲synchronize关键字。  
**完**

参考书籍：

1. 深入理解Java虚拟机(JVM高级特性与最佳实践)。
2. Java并非编程实战

*如果你习惯在微信看文章，欢迎关注公众号：****苦逼的码农****，获取更多原创文章，后台回复”礼包”送你一份资源大礼包。*