# Java多线程工作内存

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。

Java的内存模型分为主内存（物理内存），和工作内存。**工作内存是寄存器和高速缓存的抽象。涵盖了缓存、写缓冲区、寄存器、其它硬件和编译器优化。**

主内存是所有的线程所共享的，**工作内存是每个线程自己有一个，不是共享的**。线程工作时将要用到的变量从主内存拷贝到自己的工作内存，然后在工作内存中进行读和写。

|  |
| --- |
|  |

# Java中线程安全问题

Java是一种跨平台的语言，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。

# volatile保证变量可见性

### 可见性

多线程环境下，某个变量被某个线程修改，其它线程能够立即知道该线程被修改，当其它线程读取该变量的值时最终会去内存中读取，而不是从自己的**工作内存**中读取。此时该变量具有可见性。

|  |
| --- |
| x = x + 1; |

情形：线程A读取X的值为1并执行自增操作，此时工作内存X值为2，并写入主存X=2。线程B读取自己工作内存中X的值为1，并执行加1操作，此时工作内存为2，并写入主存X=2。最终主存中X的值为2。而我们期待的结果是3。

**当使用volatile保证变量可见性后，线程A更改了主存中X的值，此时线程B能立即知道X的已被修改，线程B计算前会直接从主存中读取X的值为2，再执行加1，最终结果为3。**

### volatile保证变量可见性的原理

# volatile保证代码有序性

### 有序性

|  |
| --- |
| int shu = 6; int xin = 6; int sheng = 6; |

上面这三行代码，无论谁先执行，最终结果都不会变，所以Java虚拟机编译时会进行顺序重排（为什么要重排？），比如先执行int xin = 6;当重排后代码不具有有序性，相反代码具有有序性。

### 重排后的线程安全问题

虽然重排后不会对变量值产生影响，但是会存在线程安全问题。

|  |
| --- |
| public class Test *{* private static boolean *isReady*;  private static int *myNumber*;   public static void main*(*String*[]* args*) {* new Thread*(()* -> *{* while *(*!*isReady)*;  System.*out*.println*(myNumber)*;  *})*.start*()*;  *myNumber* = 66;  *isReady* = true;  *} }* |

理想情况：myNumber=66， isReady = true，结束循环打印66

重排序后：isReady=true ，结束循环打印 0，myNumber = 66

### volatile保证有序性

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。例如把上面的*myNumber*声明为volatile，那么*myNumber* = 66;一定会比*isReady* = 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不能完全保证一个变量的线程安全

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

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。

举例：

|  |
| --- |
| 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：

从内存中读取b的值。

进行a = b + 1这个运算

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。

### volatile声明的变量无法保证线程安全举例

|  |
| --- |
| public class Test *{* public static volatile int *t* = 0;  public static void main*(*String*[]* args*)* throws InterruptedException *{* CountDownLatch countDownLatch = new CountDownLatch*(*10*)*;  for*(*int i = 0; i < 10; i++*){  //每个线程对t进行1000次加1的操作* new Thread*(()* -> *{* for*(*int j = 0; j < 1000; j++*){  t* = *t* + 1;  *}* countDownLatch.countDown*()*;  *})*.start*()*;  *}  //等待所有累加线程都结束* countDownLatch.await*()*;  *//打印t的值* System.*out*.println*(t)*;  *} }* |

运行该代码：打印结果多半不是 1000\*10

问题就出现在t = t + 1这句代码中。我们来分析一下

|  |
| --- |
| 线程A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B，注意，刚才线程B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B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

1. **变量的写操作不依赖于当前值。**
2. **该变量没有包含在具有其他变量的不变式中。**

# volatile的使用场景有哪些？