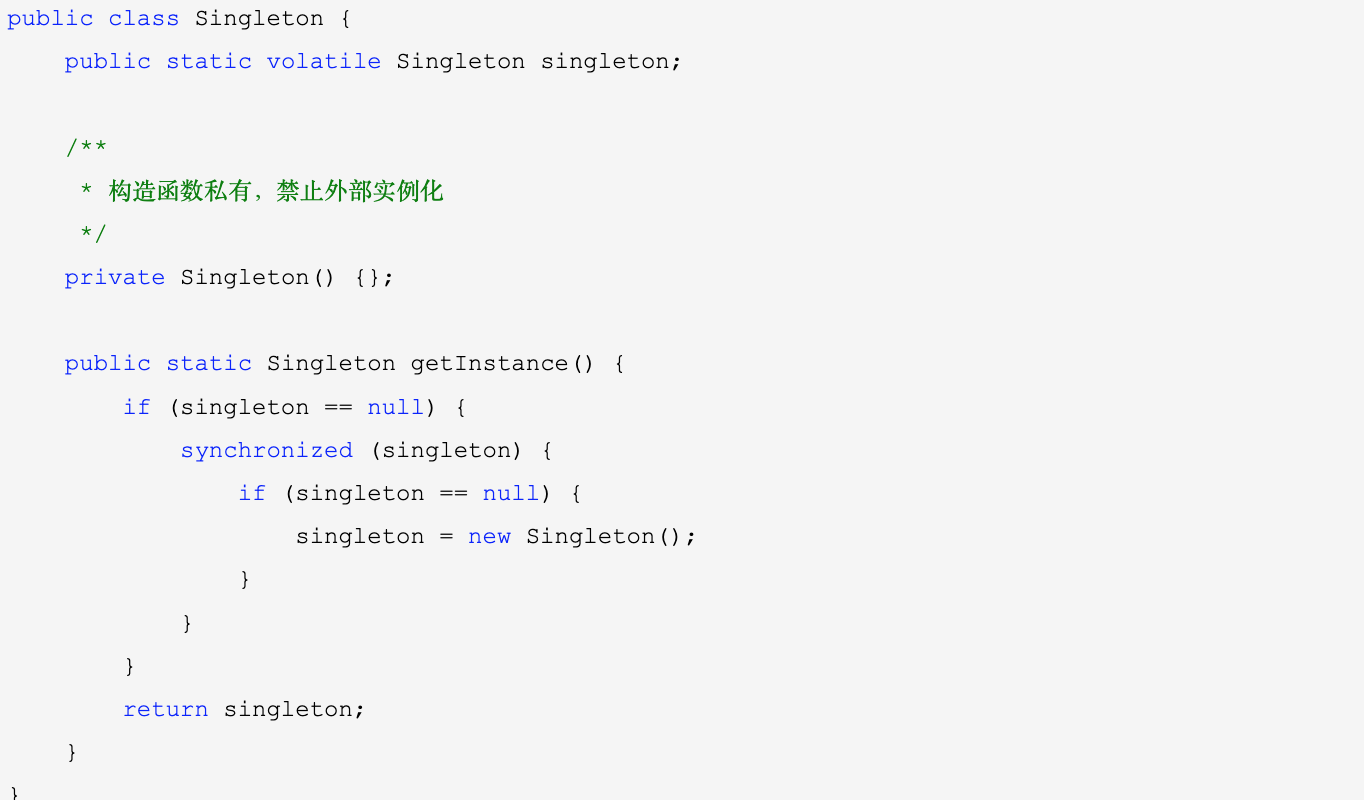
volatile关键字就是Java中提供的另一种解决可见性和有序性问题的方案。对于原子性，需要强调一点，也是大家容易误解的一点：对volatile变量的单次读/写操作可以保证原子性的，如long和double类型变量，但是并不能保证i++这种操作的原子性，因为本质上i++是读、写两次操作。

防止重排：

我们从一个最经典的例子来分析重排序问题。大家应该都很熟悉单例模式的实现，而在并发环境下的单例实现方式，我们通常可以采用双重检查加锁（DCL）的方式来实现。其源码如下：



现在我们分析一下为什么要在变量singleton之间加上volatile关键字。要理解这个问题，先要了解对象的构造过程，实例化一个对象其实可以分为三个步骤：

　　（1）分配内存空间。

　　（2）初始化对象。

　　（3）将内存空间的地址赋值给对应的引用。

但是由于操作系统可以对指令进行重排序，所以上面的过程也可能会变成如下过程：

　　（1）分配内存空间。

　　（2）将内存空间的地址赋值给对应的引用。

　　（3）初始化对象

如果是这个流程，多线程环境下就可能将一个未初始化的对象引用暴露出来，从而导致不可预料的结果。因此，为了防止这个过程的重排序，我们需要将变量设置为volatile类型的变量。

Java中的happen-before规则：

通俗一点说就是如果a happen-before b，则a所做的任何操作对b是可见的。（这一点大家务必记住，因为happen-before这个词容易被误解为是时间的前后）。

同一个线程中的，前面的操作 happen-before 后续的操作。（即单线程内按代码顺序执行。但是，在不影响在单线程环境执行结果的前提下，编译器和处理器可以进行重排序，这是合法的。换句话说，这一是规则无法保证编译重排和指令重排）。

监视器上的解锁操作 happen-before 其后续的加锁操作。（Synchronized 规则）

对volatile变量的写操作 happen-before 后续的读操作。（volatile 规则）

线程的start() 方法 happen-before 该线程所有的后续操作。（线程启动规则）

线程所有的操作 happen-before 其他线程在该线程上调用 join 返回成功后的操作。

如果 a happen-before b，b happen-before c，则a happen-before c（传递性）。

这里我们主要看下第三条：volatile变量的保证有序性的规则。重排序分为编译器重排序和处理器重排序。为了实现volatile内存语义，JMM会对volatile变量限制这两种类型的重排序。

为了实现volatile可见性和happen-befor的语义。JVM底层是通过一个叫做“内存屏障”的东西来完成。内存屏障，也叫做内存栅栏，是一组处理器指令，用于实现对内存操作的顺序限制。

（1）LoadLoad 屏障

执行顺序：Load1—>Loadload—>Load2

确保Load2及后续Load指令加载数据之前能访问到Load1加载的数据。

（2）StoreStore 屏障

执行顺序：Store1—>StoreStore—>Store2

确保Store2以及后续Store指令执行前，Store1操作的数据对其它处理器可见。

（3）LoadStore 屏障

执行顺序： Load1—>LoadStore—>Store2

确保Store2和后续Store指令执行前，可以访问到Load1加载的数据。

（4）StoreLoad 屏障

执行顺序: Store1—> StoreLoad—>Load2

确保Load2和后续的Load指令读取之前，Store1的数据对其他处理器是可见的。

public class MemoryBarrier {

int a, b;

volatile int v, u;

void f() {

int i, j;

i = a;

j = b;

i = v;

//LoadLoad

j = u;

//LoadStore

a = i;

b = j;

//StoreStore

v = i;

//StoreStore

u = j;

//StoreLoad

i = u;

//LoadLoad

//LoadStore

j = b;

a = i;

}

}

原子性：

因为long和double两种数据类型的操作可分为高32位和低32位两部分，因此普通的long或double类型读/写可能不是原子的。因此，鼓励大家将共享的long和double变量设置为volatile类型，这样能保证任何情况下对long和double的单次读/写操作都具有原子性。

i++其实是一个复合操作，包括三步骤：

　　（1）读取i的值。

　　（2）对i加1。

　　（3）将i的值写回内存。

volatile是无法保证这三个操作是具有原子性的，我们可以通过AtomicInteger或者Synchronized来保证+1操作的原子性。

可见性：

如果一个字段被声明成volatile,那么java线程内存模型将确保所有线程看到的这个变量的值都是一致的.

线程本身并不直接与主内存进行数据的交互，而是通过线程的工作内存来完成相应的操作。这也是导致线程间数据不可见的本质原因。因此要实现volatile变量的可见性，直接从这方面入手即可。对volatile变量的写操作与普通变量的主要区别有两点：

　　（1）修改volatile变量时会强制将修改后的值刷新的主内存中。

（2）修改volatile变量后会导致其他线程工作内存中对应的变量值失效。因此，再读取该变量值的时候就需要重新从读取主内存中的值。

先看一段JIT编译器生成的汇编指令：

//Java代码如下

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

//反汇编后

0x01a3de1d: movb $0x0,0x1104800(%esi);

0x01a3de24: lock add1 $0x0,(%esp);

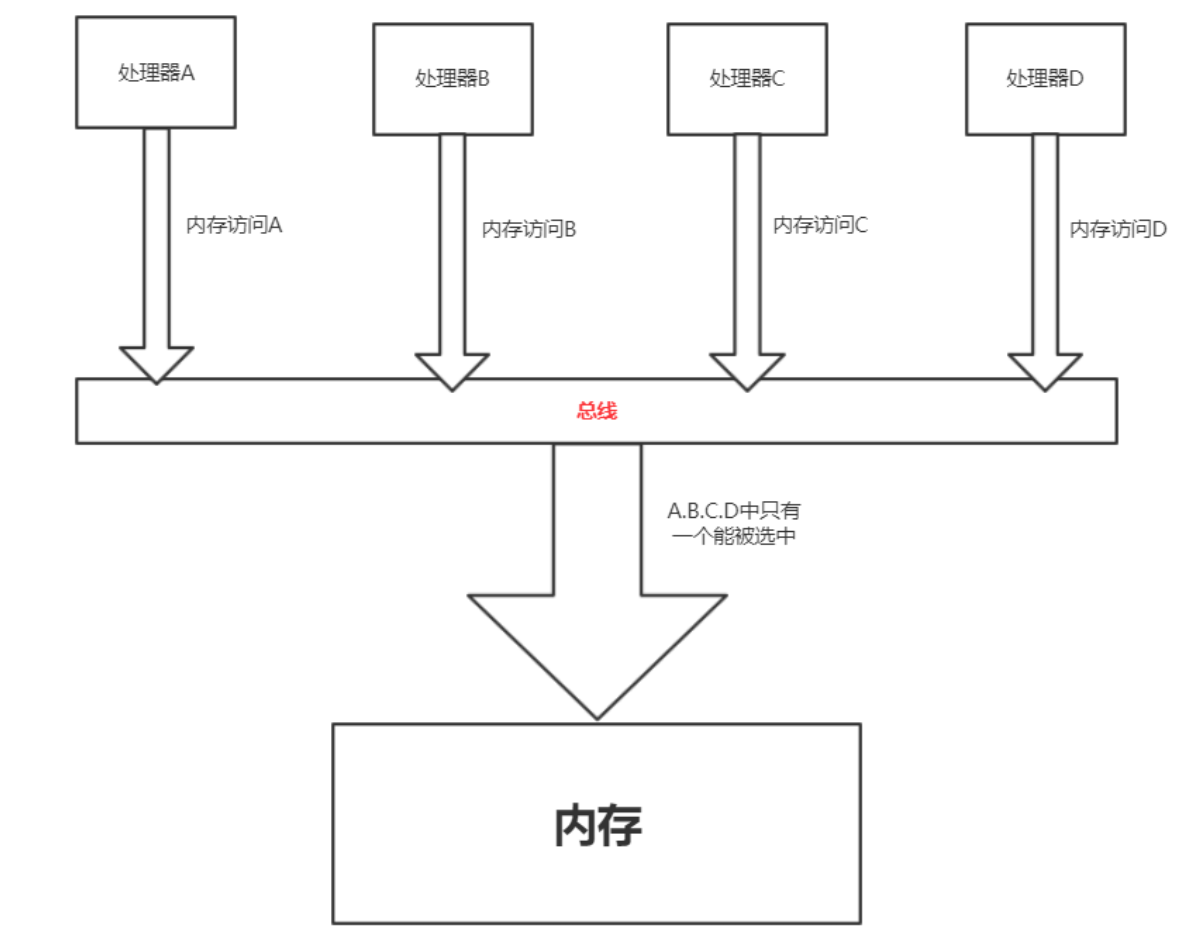
有volatile修饰的变量在进行写操作时会出现第二行反汇编代码,重点在lock这个指令.它有两个目的:

立即回写当前处理器缓存行的值到内存.

其他所有cpu缓存了该地址的数据将会失效.

这里大家也许会有疑问,有没有可能存在多个cpu一起回写数据?

答案是不会的.虽然cpu鼓励多个处理器可以有竞争,但是总线会对竞争做出裁决,只会有一个cpu获取优先权.其他处理器会被总线禁止,处于阻塞状态.如下图:



对于第二点,其他cpu缓存该地址的数据失效后想要再次使用的话就必须得从主内存中重新读取,这样就能保证再次执行计算时所获取的值是最新的,也可以认为所有CPU的缓存是一致的,这也就证明了volatile修饰的字段是可见的.

如何正确的运用volatile

要想运用好volatile修饰符,需要保证运用场景符合下述规则:

运算结果不依赖变量的当前值.

该变量不需要和其他变量共同参与约束.

例如使用volatile变量来控制并发就很合适:

volatile boolean shutdownWork;

public void shutdowm(){

shutdownWork = true;

}

public void doWork(){

while (!shutdownWork){

//execute task

}

}

上面这段代码运行结果并无需依赖shutdownWork的值,但是只要shutdownWork的值一旦经过改变,便会立即被其他所有线程所感知,然后停止执行任务.

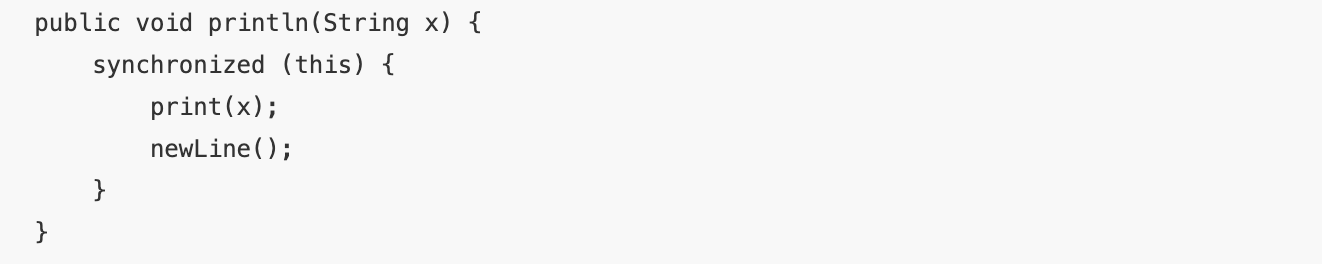
在多处理器下,为了保证各个处理器的缓存是一致的,处理器会使用嗅探技术来保证它的内部缓存,系统内存和其他处理器的缓存的数据在总线上保持一致.如果通过嗅探检测到其他处理器打算写内存地址,而这个地址当前处于共享状态,那么正在嗅探的处理器将使它的缓存无效,在下次访问相同的内存地址时,强制执行缓存行填充,也就是从内存中重新读取该内存地址指向的值.

当没有添加volidate 修饰属性的时候，数据什么时候从缓存行刷新到主存？

曾经有遇到过这样一个问题，有一个共享变量keepRunning=true,线程A中执行while (keepRunning)；，线程B中执行keepRunning = false;，在main函数中同时开启A，B线程，然后会发现程序会一直运行且不会退出。说白了这其实就是一个典型的可见性问题，A线程并不知道keepRunning已经被修改过了，故未将修改后的keepRunning变量的值从主内存中读取到线程缓存中来。



按上述代码直接运行，你会发现在打印完end: false之后，程序并没有正常的退出，而是在一直跑着while (keepRunning)这个死循环。但是我们尝试着将其中注释的代码System.out.println("如果你不注释这一行，程序会正常停止！");给取消掉注释，再运行一次上面的代码，就会发现程序会跑一段时间后正常退出。看到这里大家也许会感到奇怪，在进行System.out.println这个IO操作后，线程t竟然读到了主线程写入的t.keepRunning = false这个值，然后导致while循环退出了。这里就不得不去看下println这个方法的源码了。



这里我们会发现println方法是一个同步的方法。大家都知道用synchronized这个关键字修饰的方法或者代码块能保证代码串行化的执行(同一时间只能有一个线程获取执行权限)，

从本质上来说，当线程释放一个锁时会强制性的将工作内存中之前所有的写操作都刷新到主内存中去，而获取一个锁则会强制性的加载可访问到的值到线程工作内存中来。虽然锁操作只对同步方法和同步代码块这一块起到作用，但是影响的却是线程执行操作所使用的所有字段。这也就解释了为什么加上System.out.println("如果你不注释这一行，程序会正常停止！");这句代码后，线程t能够读取到修改后的keepRunning的值了。对于这个问题上，有些人的说法是：打印是IO操作，而IO操作会引起线程的切换，线程切换会导致线程原本的缓存失效，从而也会读取到修改后的值。这里我认为这种说法也是有道理的，我尝试着将打印换成File file = new File("G://1.txt");这句代码，程序也能够正常的结束。当然，在这里大家也可以尝试将将打印替换成synchronized(NoVisibility\_Demonstration.class){ }这句空同步代码块，发现程序也能够正常结束。

针对上述问题，最起码可以得出一个结论：当进行IO操作或者线程内部调用synchronized修饰的方法或者同步代码块时，线程的缓存会进行刷新，也就是会感知到共享变量的变化。当然这也只是针对非volatile修饰的变量而言，当变量被申明为volatile的时候，每次使用该变量都会从主内存中进行读取。

只有在以下条件下，才能保证一个线程对字段的更改对其他线程可见：

写入线程释放同步锁，读取线程随后获取相同的同步锁。释放锁的时候会强制从线程使用的工作内存中刷新所有写入，并且在获取锁的时候会强制重新加载可访问字段的值。

如果一个字段被声明为volatile，则写入线程会立即将修改后的值同步到主内存。读取线程必须在每次访问时重新加载volatile字段的值。

线程第一次访问一个对象的某个字段时，它会看到字段的初始值或来自某个其他线程写入的值。

当一个线程终止时，所有写入的变量都被刷新到主内存。例如:现有线程A，B，在B线程中调用A.join()，那么在B中可以保证看到A线程产生的影响。