**Java Volatile Keyword**

* [Variable Visibility Problems](http://tutorials.jenkov.com/java-concurrency/volatile.html#variable-visibility-problems)
* [The Java volatile Visibility Guarantee](http://tutorials.jenkov.com/java-concurrency/volatile.html#the-java-volatile-visibility-guarantee)
  + [Full volatile Visibility Guarantee](http://tutorials.jenkov.com/java-concurrency/volatile.html#full-volatile-visibility-guarantee)
* [Instruction Reordering Challenges](http://tutorials.jenkov.com/java-concurrency/volatile.html#instruction-reordering-challenges)
* [The Java volatile Happens-Before Guarantee](http://tutorials.jenkov.com/java-concurrency/volatile.html#the-java-volatile-happens-before-guarantee)
* [volatile is Not Always Enough](http://tutorials.jenkov.com/java-concurrency/volatile.html#volatile-is-not-always-enough)
* [When is volatile Enough?](http://tutorials.jenkov.com/java-concurrency/volatile.html#when-is-volatile-enough)
* [Performance Considerations of volatile](http://tutorials.jenkov.com/java-concurrency/volatile.html#performance-considerations-of-volatile)

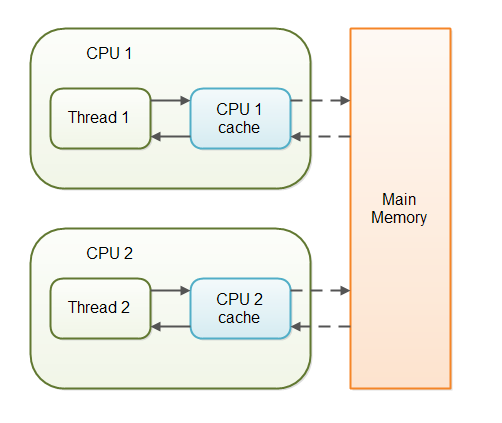
Java的volatile关键字用于将Java变量标记为“存储在主内存中”。更准确地说，这意味着volatile变量的每次读取都将从计算机的主存储器而不是从CPU缓存中读取，对volatile变量的每次写入都将被写入主存储器，而不仅仅是CPU缓存。

实际上，自从Java 5，易失性关键字保证的不仅仅是易失性变量被写入主内存和从主内存读取。我将在下面的章节中解释这一点。

**Variable Visibility Problems**

Java关键字volatile 保证了变量对线程的变化的可见性。这听起来有点抽象，所以让我详细说明一下。

在一个线程对非volatile变量进行操作的多线程应用程序中，线程对非易失性变量进行操作，出于性能原因，每个线程可能会在处理这些变量的时候将变量从主存储器复制到CPU缓存中。如果您的计算机包含多个CPU，则每个线程可能在不同的CPU上运行。也就是说，每个线程可以将变量复制到不同CPU的CPU缓存中。说明如下：



对于非volatile变量，什么时候Java虚拟机（JVM）从主存储器读取数据到CPU缓存或从CPU缓存写入数据到主存储器，没有任何保证。这可能会导致几个问题，我将在下面的章节中解释。

设想一种情况，其中两个或多个线程可以访问共享对象，该对象包含这样声明的counter变量：

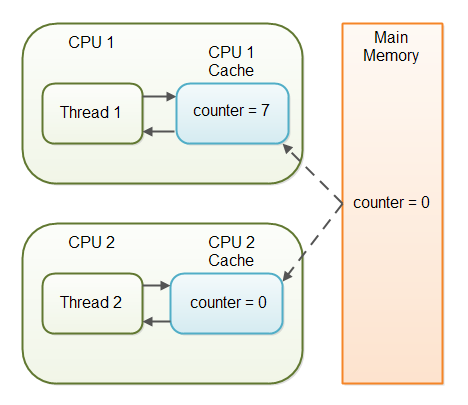
public class SharedObject {

public int counter = 0;

}

同样可以想象，只有线程1递增counter变量，但是线程1和线程2都可以不时地读取counter变量。

如果counter变量没有被声明为volatile，则无法保证计数器变量的值何时从CPU缓存写入主存储器。这意味着CPU缓存中的counter变量值可能与主存中的counter值不一样。这里说明了这种情况：



线程没有看到变量的最新值因为它还没有被另一个线程写入主内存的问题，称为“可见性”问题。一个线程的更新对于其他线程是不可见的。

**The Java volatile Visibility Guarantee**

Java的volatile关键字旨在解决变量可见性问题。通过声明counter变量为volatile，所有对counter变量的写入都将立即写入主存储器。同样，counter变量的所有读取将直接从主存储器读取。

下面是counter变量的volatile声明的样子：

public class SharedObject {

public **volatile** int counter = 0;

}

声明变量volatile保证了其他线程对该变量写入的*可见性*。

在上面给出的场景中，一个线程(T1)修改counter，而另一个线程(T2)读取counter(但从不修改它)，声明counter变量volatile足以保证T2对counter变量的写入的可见性。

然而，如果T1和T2都在递增counter变量，那么仅声明counter变量volatile是不够的。稍后再谈。

**Full volatile Visibility Guarantee**

实际上，Java volatile的可见性保证超出了volatile变量本身。能见度保证如下：

* 如果线程A写入volatile变量，而线程B随后读取相同的volatile变量，那么线程A在写入volatile变量之前可见的所有变量在线程B读取易失性变量之后也将可见
* 如果线程A读取了一个volatile变量，那么所有对线程A可见的变量在读取volatile变量时也将重新从内存读取

让我用一个代码示例来说明：

public class MyClass {

private int years;

private int months

private volatile int days;

public void update(int years, int months, int days){

this.years = years;

this.months = months;

this.days = days;

}

}

udpate()方法编写三个变量，其中只有days是volatile的。

完全volatile可见性保证意味着，当一个值被写入days时，线程可见的所有变量也被写入主存储器。这意味着，当一个值被写入days时，years和months的值也被写入主存。

当读取years、months 和days 的值时，你可以这样做：

public class MyClass {

private int years;

private int months

private volatile int days;

**public int totalDays() {**

**int total = this.days;**

**total += months \* 30;**

**total += years \* 365;**

**return total;**

**}**

public void update(int years, int months, int days){

this.years = years;

this.months = months;

this.days = days;

}

}

注意，totalDays()方法是通过将days值读入total变量开始的。在读取days的值时，months、years的值也读入主存储器。因此，您通过上述读取序列可以看到最新的days、months和years的值。

**Instruction Reordering Challenges**

Java VM和CPU被允许重排程序中的指令，出于性能原因，只要指令的语义保持不变。例如，请看下面的说明：

int a = 1;

int b = 2;

a++;

b++;

这些指令可以在不丢失程序语义的情况下重新排序为以下顺序：

'

int a = 1;

a++;

int b = 2;

b++;

然而，当变量之一是volatile变量时，指令重排提出了挑战。让我们看看来自早前这个java volatile教程例子的MyClass类：

public class MyClass {

private int years;

private int months

private volatile int days;

public void update(int years, int months, int days){

this.years = years;

this.months = months;

this.days = days;

}

}

一旦update()方法将一个值写入days，新写入years和months的值也将被写入主内存。但是，如果Java VM重新排序指令会怎么样，比如：

public void update(int years, int months, int days){

this.days = days;

this.months = months;

this.years = years;

}

当修改days变量时，months和years的值仍然写入主存储器，但是这次是在将新值写入months和years之前发生的。因此，新的值不能正确地显示给其他线程。重新排序指令的语义发生了变化。

Java有一个解决这个问题的方法，我们将在下一节中看到。

**The Java volatile Happens-Before Guarantee**

为了解决指令重新排序的挑战，Java volatile 关键字除了可见性保证之外，还提供了一个“happens-before”的保证。“happens-before”保证以下：

* 如果读写操作最初发生在写入volatile变量之前，读写其他变量操作不能被重排发生在写入volatile变量操作之后。对volatile变量的写入操作之前的读/写操作保证在写入volatile变量操作“之前发生”。请注意，仍然有可能将位于对volatile变量写入之后的其他变量的读/写重新排序为在对易失性写入之前发生。只是不能相反。从后到前是允许的，但从前到后是不允许的。
* 如果读取/写入操作最初发生在读取volatile变量操作之后，则对其他变量的读取和写入操作不能被重新排序为在读取volatile变量操作之前发生。注意，在读取volatile变量之前发生的其他变量的读取可能被重新排序为在读取volatile变量之后发生。而不是相反。从前到后是允许的，但从后到前是不允许的。

上述的“happens-before”保证确保了volatile关键字的可见性保证被强制执行。

**volatile is Not Always Enough**

即使volatile关键字保证volatile 变量的所有读取都直接从主存储器读取，并且对volatile 变量的所有写入都直接写入主存储器，仍然存在声明变量volatile不够的情况。

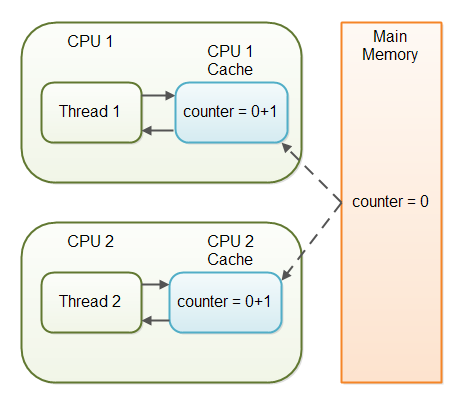
在前面解释的情况中，只有线程1写入共享counter 变量，声明counter 变量volatile就足以确保线程2总是看到最新的写入值。

事实上，多个线程甚至可以写入共享的volatile 变量，并且仍然具有存储在主存储器中的正确值，如果写入变量的新值不依赖于其先前值的话。换言之，如果一个线程向共享的volatile 变量写入一个值并不首先需要读取它的值来找出它的下一个值的话。

一旦线程需要首先读取volatile 变量的值，并且根据该值为共享的volatile 变量生成一个新的值，volatile 变量就不再足以保证正确的可见性。在读取volatile 变量和写入其新值之间的短时间间隔创建了一个竞争条件，其中多个线程可能读取volatile 变量的相同值，为该变量生成一个新值，并且在将值写回主存储器时重写彼此的值。

多个线程递增同一个counter的情况正是volatile变量不足的情况。下面的部分将更详细地解释这一情况。

设想一下，如果线程1将值0的共享counter变量读入其CPU缓存，将其递增到1，而不将更改的值写回主存储器。然后，线程2可以从主存储器中读取相同的counter变量，其中变量的值仍然是0，进入它自己的CPU缓存。线程2也可以将计数器递增到1，也不会将其写入主存储器。这种情况如下图所示：



线程1和线程2现在实际上是不同步的。共享counter 变量的实际值应该是2，但是每个线程的CPU缓存中的变量的值是1，而在主存中，该值仍然是0。真是一团糟！即使线程最终将共享counter 变量的值写入主内存，值也将是错误的。

**When is volatile Enough?**

正如我前面提到的，如果两个线程都在读和写共享变量，那么使用volatile关键字是不够的。在这种情况下，需要使用一个同步([**synchronized**](http://tutorials.jenkov.com/java-concurrency/synchronized.html))来保证变量的读取和写入是原子的。读取或写入volatile变量不会阻止线程读取或写入。要做到这一点，必须使用synchronized关键字环绕临界区间。

作为同步块的替代，您还可以使用[**java.util.concurrentpackage**](http://tutorials.jenkov.com/java-util-concurrent/index.html)中找到的许多原子数据类型中的一种。例如，[**AtomicLong**](http://tutorials.jenkov.com/java-util-concurrent/atomiclong.html)或[**AtomicReference**](http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html)或其他之一。

如果只有一个线程读取和写入易失性变量的值，而其他线程只读取该变量，那么读取线程将确保看到写入volatile变量的最新值。在不使变量volatile的情况下，这是不能保证的。

volatile关键字确保在32位和64位变量上工作。

**Performance Considerations of volatile**

volatile变量的读取和写入导致变量被读取或写入主存储器。读取和写入主内存比访问CPU缓存更昂贵。访问volatile变量还阻止指令重新排序，后者是一种正常的性能增强技术。因此，只有在真正需要强制变量的可见性时，才应使用volatile变量。