### 1、volatile的作用

我们已经知道可见性、有序性及原子性问题，通常情况下我们可以通过Synchronized关键字来解决这些个问题，不过如果对Synchronized原理有了解的话，应该知道Synchronized是一个比较重量级的操作，对系统的性能有比较大的影响，所以，如果有其他解决方案，我们通常都避免使用Synchronized来解决问题。

而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。

### 2、volatile的使用

几种使用方式和场景

#### 2.1 防止重排序

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。

public class Singleton {

public static volatile Singleton *singleton*;  
  
 private Singleton() {  
 }  
  
 public static Singleton getInstance() {  
 if (*singleton* == null) {  
 synchronized (*singleton*) {  
 if (*singleton* == null) {  
 *singleton* = new Singleton();  
 }  
 }  
 }  
 return *singleton*;  
 }  
}

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：

* 分配内存空间。
* 初始化对象。
* 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：

* 分配内存空间。
* 将内存空间的地址赋值给对应的引用。
* 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。

#### 2.2 实现可见性

可见性问题主要指一个线程修改了共享变量值，而另一个线程却看不到。引起可见性问题的主要原因是每个线程拥有自己的一个高速缓存区——线程工作内存。

volatile关键字能有效的解决这个问题，我们看下下面的例子，就可以知道其作用：

public class VolatileTest {

volatile int a = 1;  
 volatile int b = 2;  
  
 public void change() {  
 a = 3;  
 b = a;  
 }  
  
 public void print() {  
 System.*out*.println("b=" + b + ";a=" + a);  
 }  
  
 public static void main(String[] args) {  
 while (true) {  
 final VolatileTest test = new VolatileTest();  
 new Thread(() -> {  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 test.change();  
 }).start();  
  
 new Thread(() -> {  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 test.print();  
 }).start();  
  
 }  
 }  
}

#### 2.3 保证原子性

volatile只能保证对单次读/写的原子性。

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。

关于volatile变量对原子性保证，有一个问题容易被误解。现在我们就通过下列程序来演示一下这个问题：

public class VolatileTest01 {

volatile int i;  
  
 public void addI() {  
 i++;  
 }  
  
 public static void main(String[] args) throws InterruptedException {  
 final VolatileTest01 test01 = new VolatileTest01();  
 for (int n = 0; n < 1000; n++) {  
 new Thread(() -> {  
 try {  
 Thread.*sleep*(10);  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 test01.addI();  
 }).start();  
 }  
  
 Thread.*sleep*(10000);  
  
 System.*out*.println(test01.i);  
 }  
}

i++其实是一个复合操作，包括三步骤：

* 读取i的值。
* 对i加1。
* 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。

注：上面几段代码中多处执行了Thread.sleep()方法，目的是为了增加并发问题的产生几率，无其他作用。

### 3、Volatile的原理

#### 3.1 可见性实现

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：

* 修改volatile变量时会强制将修改后的值刷新到主内存中。
* 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。

通过这两个操作，就可以解决volatile变量的可见性问题。

#### 3.2 有序性实现

在解释这个问题前，我们先来了解一下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）。happen-before规则：

* 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。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。

下面是JMM针对volatile变量所规定的重排序规则表：



#### 3.3 内存屏障

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

下面是完成上述规则所要求的内存屏障：



* 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。

* 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。

* 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。

* 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。

最后我可以通过一个实例来说明一下JVM中是如何插入内存屏障的：

### 4、总结

总体上来说volatile的理解还是比较困难的，如果不是特别理解，也不用急，完全理解需要一个过程，在后续的文章中也还会多次看到volatile的使用场景。这里暂且对volatile的基础知识和原来有一个基本的了解。

总体来说，volatile是并发编程中的一种优化，在某些场景下可以代替Synchronized。但是，volatile的不能完全取代Synchronized的位置，只有在一些特殊的场景下，才能适用volatile。总的来说，必须同时满足下面两个条件才能保证在并发环境的线程安全：

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