volatile关键字虽然从字面上理解起来比较简单，但是要用好不是一件容易的事情。由于**volatile关键字是与Java的内存模型有关的**，因此在了解volatile关键字时，可以先看一下上一篇博客了解一下与内存模型相关的概念和知识，然后此片文章才分析volatile关键字的实现原理，在给出了几个使用volatile关键字的场景。

**深入剖析volatile关键字**

在前面讲述了很多东西，其实都是为讲述volatile关键字作铺垫，那么接下来我们就进入主题。

**1.volatile关键字的两层语义**

**一旦一个共享变量（类的成员变量、类的静态成员变量）被volatile修饰之后，那么就具备了两层语义：**

　　1）保证了不同线程对这个变量进行操作时的可见性，即一个线程修改了某个变量的值，这新值对其他线程来说是立即可见的。

　　2）禁止进行指令重排序。

　　先看一段代码，假如线程1先执行，线程2后执行：

//线程1

boolean stop = false;

while(!stop){

    doSomething();

}

//线程2

stop = true;

　　这段代码是很典型的一段代码，很多人在中断线程时可能都会采用这种标记办法。但是事实上，这段代码会完全运行正确么？即一定会将线程中断么？不一定，也许在大多数时候，这个代码能够把线程中断，但是**也有可能会导致无法中断线程**（虽然这个可能性很小，但是只要一旦发生这种情况就会造成死循环了）。

　　下面解释一下这段代码为何有可能导致无法中断线程。在前面已经解释过，每个线程在运行过程中都有自己的工作内存，那么线程1在运行的时候，**会将stop变量的值拷贝一份放在自己的工作内存当中。**

　　那么当线程2更改了**stop变量的值之后，但是还没来得及写入主存当中**，线程2转去做其他事情了，那么**线程1由于不知道线程2对stop变量的更改，因此还会一直循环下去。**

　　但是用volatile修饰之后就变得不一样了：

　　第一：**使用volatile关键字会强制将修改的值立即写入主存；**

　　第二：**使用volatile关键字的话，当线程2进行修改时，会导致线程1的工作内存中缓存变量stop的缓存行无效**（反映到硬件层的话，就是CPU的L1或者L2缓存中对应的缓存行无效）；

　　第三：由于线程1的工作内存中缓存变量stop的缓存行无效，所以**线程1再次读取变量stop的值时会去主存读取。**

　　那么在线程2修改stop值时（当然这里包括2个操作，**修改线程2工作内存中的值，然后将修改后的值写入内存**），会使得线程1的工作内存中缓存变量stop的缓存行无效，然后线程1读取时，发现自己的缓存行无效，它会等待缓存行对应的主存地址被更新之后，然后去对应的主存读取最新的值。

**那么线程1读取到的就是最新的正确的值。**

**2.volatile保证原子性吗？**

　　从上面知道volatile关键字**保证了操作的可见性**，但是volatile能保证对变量的操作是原子性吗？

　　下面看一个例子：

|  |
| --- |
| public class Test {      public volatile int inc = 0;        public void increase() {          inc++;      }        public static void main(String[] args) {          final Test test = new Test();          for(int i=0;i<10;i++){              new Thread(){                  public void run() {                      for(int j=0;j<1000;j++)                          test.increase();                  };              }.start();          }            while(Thread.activeCount()>1)  //保证前面的线程都执行完              Thread.yield();          System.out.println(test.inc);      }  } |

 　　大家想一下这段程序的输出结果是多少？也许有些朋友认为是10000。但是事实上运行它会发现每次运行结果都不一致，都是**一个小于10000的数字**。

　　可能有的朋友就会有疑问，不对啊，上面是对变量inc进行自增操作，由于volatile保证了可见性，那么在每个线程中对inc自增完之后，在其他线程中都能看到修改后的值啊，所以有**10个线程分别进行了1000次操作，**那么最终inc的值应该是1000\*10=10000。

　　这里面就有一个误区了，**volatile关键字能保证可见性没有错，但是上面的程序错在没能保证原子性。可见性只能保证每次读取的是最新的值，但是volatile没办法保证对变量的操作的原子性。**

　　在前面已经提到过，自增操作是不具备原子性的，它**包括读取变量的原始值、进行加1操作、写入工作内存**。那么就是说自增操作的三个子操作可能会分割开执行，就有可能导致下面这种情况出现：

　　假如某个时刻变量inc的值为10，

　　线程1对变量进行自增操作，线程1先读取了变量inc的原始值，然后线程1被阻塞了；

　　然后线程2对变量进行自增操作，线程2也去读取变量inc的原始值，由于线程1只是对**变量inc进行读取操作，而没有对变量进行修改操作**，所以不会**导致线程2的工作内存中缓存变量inc的缓存行无效**，所以线程2会直接去主存读取inc的值，发现inc的值时10，然后进行加1操作，并把11写入工作内存，最后写入主存。

　　然后线程1接着进行加1操作，由于**已经读取了inc的值，注意此时在线程1的工作内存中inc的值仍然为10**，所以线程1对inc进行加1操作后inc的值为11，然后将11写入工作内存，最后写入主存。

　　那么两个线程分别进行了一次自增操作后，**inc只增加了1**。

解释到这里，可能有朋友会有疑问，不对啊，前面不是保证一个变量在修改volatile变量时，会让缓存行无效吗？然后其他线程去读就会读到新的值，对，这个没错。这个就是上面的happens-before规则中的volatile变量规则（**写先于读，但是只读没写，其他的线程也就无法看到，那么都读完，再一起写，等于只存入了慢写的那个值**），但是要注意，线程1对变量进行读取操作之后，被阻塞了的话，并没有对inc值进行修改。然后虽然volatile能保证线程2对变量inc的值读取是从内存中读取的，但是**线程1没有进行修改，所以线程2根本就不会看到修改的值。**

**此处的解释应该多一句：可见性只是保证每次去读的是最新值，是别的线程修改更新过的，然而如果已经读好了，但是没操作，然后别人已经操作过了，你此时再修改完inc再去覆盖掉别人修改过的，那前面线程2的操作就作废了。这就不同于前面文章的那个例子：**

**Boolean b = false；//线程1**

**While（！b）｛**

**｝**

**b=true;//线程2**

这个是每次线程1在循环的时候，都要去读b的值的，就是每次判断！b之前，都是一次新的操作（要经过读取b的值，判断b，执行，这个过程），都是要去读取看b在主内存是怎么样的一种形态。而上面说到的b++，已经读取了，只是没有操作++和写入而已，完全是两码事。

　　根源就在这里，自增操作不是原子性操作，而且volatile也无法保证对变量的任何操作都是原子性的。

　　把上面的代码改成以下任何一种都可以达到效果：

　　采用synchronized：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

public class Test {

public int inc = 0;

public synchronized void increase() {

inc++;

}

public static void main(String[] args) {

final Test test = new Test();

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

new Thread(){

public void run() {

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

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

复制代码

　　采用Lock：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

public class Test {

public int inc = 0;

Lock lock = new ReentrantLock();

public void increase() {

lock.lock();

try {

inc++;

} finally{

lock.unlock();

}

}

public static void main(String[] args) {

final Test test = new Test();

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

new Thread(){

public void run() {

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

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

复制代码

　　采用AtomicInteger：

http://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

复制代码

public class Test {

public AtomicInteger inc = new AtomicInteger();

public void increase() {

inc.getAndIncrement();

}

public static void main(String[] args) {

final Test test = new Test();

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

new Thread(){

public void run() {

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

test.increase();

};

}.start();

}

while(Thread.activeCount()>1) //保证前面的线程都执行完

Thread.yield();

System.out.println(test.inc);

}

}

复制代码

　　在java 1.5的java.util.concurrent.atomic包下提供了一些原子操作类，即对基本数据类型的 **自增（加1操作），自减（减1操作）、以及加法操作（加一个数），减法操作（减一个数）**进行了封装，**保证这些操作是原子性操作**。**atomic是利用CAS来实现原子性操作的（Compare And Swap）**，CAS实际上是利用处理器提供的CMPXCHG指令实现的，而处理器执行CMPXCHG指令是一个原子性操作。

**3.volatile能保证有序性吗？**

　　在前面提到volatile关键字能禁止指令重排序，所以volatile能**在一定程度上保证有序性。**

　　volatile关键字禁止指令重排序有两层意思：

　　1）当程序执行到volatile变量的读操作或者写操作时，在**其前面的操作的更改肯定全部已经进行**，且结**果已经对后面的操作可见；在其后面的操作肯定还没有进行**；

　　2）在进行指令优化时，不能将在对volatile变量访问的语句放在其后面执行，也不能把volatile变量后面的语句放到其前面执行。

　　可能上面说的比较绕，举个简单的例子：

|  |
| --- |
| //**x、y为非volatile变量**  //**flag为volatile变量**    x = 2;        //语句1  y = 0;        //语句2  flag = true;  //语句3  x = 4;         //语句4  y = -1;       //语句5 |

 　　由于flag变量为volatile变量，那么在进行指令重排序的过程的时候，不会将语句3放到语句1、语句2前面，也不会讲语句3放到语句4、语句5后面。但是要注意语句1和语句2的顺序、语句4和语句5的顺序是不作任何保证的。

　　并且volatile关键字能保证，执行到语句3时，语句1和语句2必定是执行完毕了的，且语句1和语句2的执行结果对语句3、语句4、语句5是可见的。

　　那么我们回到前面举的一个例子：

|  |
| --- |
| //线程1:  context = loadContext();   //语句1  inited = true;             //语句2    //线程2:  while(!inited ){    sleep()  }  doSomethingwithconfig(context); |

 　　前面举这个例子的时候，提到有可能语句2会在语句1之前执行，那么久可能导致context还没被初始化，而线程2中就使用未初始化的context去进行操作，导致程序出错。

　　这里如果用**volatile关键字对inited变量进行修饰，**就不会出现这种问题了，**因为当执行到语句2时，必定能保证context已经初始化完毕。**

**4.volatile的原理和实现机制**

　　前面讲述了源于volatile关键字的一些使用，下面我们来探讨一下volatile到底如何**保证可见性和禁止指令重排序**的。

　　下面这段话摘自《深入理解Java虚拟机》：

　　“观察加入volatile关键字和没有加入volatile关键字时所生成的**汇编代码（其实就是Java虚拟机的指令集）**发现，**加入volatile关键字时，会多出一个lock前缀指令**”

**lock前缀指令实际上相当于一个内存屏障（也叫内存栅栏）**，内存屏障会**提供3个功能：**

　　1）**它确保指令重排序时不会把其后面的指令排到内存屏障之前的位置，也不会把前面的指令排到内存屏障的后面；即在执行到内存屏障这句指令时，在它前面的操作已经全部完成；（也就是上面所说的可以解决有序性问题的方法）**

　　2）**它会强制将对缓存的修改操作立即写入主存；**

3）**如果是写操作，它会导致其他CPU中对应的缓存行无效。（只能重新去主内存读取最新的数据）**

但：

volatile仍然在执行一个从主存加载到工作内存，并且将变更的值写回主存的操作，但是：

1，**volatile保证每次读取该变量前，**都判断当前值是否已经失效（即是否已经与主存不一致），如果已经失效，则从主存load最新的变量；**（但是，如果不读取该变量值，那就没办法了。比如刚刚的自增，A++，A已经读取过了，不需要再次读取，所以该线程就不知道该变量已经失效了）。**

2，volatile保证每次对该变量做出修改时，都立即写入主存；

**使用volatile关键字的场景**

synchronized关键字是防止多个线程同时执行一段代码，那么就会很影响程序执行效率，而volatile关键字在某些情况下性能要优于synchronized，但是要注意volatile关键字是无法替代synchronized关键字的，因为**volatile关键字无法保证操作的原子性**。通常来说，使用volatile必须具备以下2个条件：

**1）对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**

　　实际上，这些条件表明，可以被写入 volatile 变量的这些有效值独立于任何程序的状态，包括变量的当前状态。

事实上，我的理解就是上面的2个条件**需要保证操作是原子性操作，才能保证使用volatile关键字的程序在并发时能够正确执行。**

　　下面列举几个Java中使用volatile的几个场景。

**1.状态标记量**

volatile**boolean** flag = false;

while(!flag){

    doSomething();

}

public void setFlag() {

    flag = true;

}

volatile boolean inited = false;

//线程1:

context = loadContext();

inited = true;

//线程2:

while(!inited ){

sleep()

}

doSomethingwithconfig(context);

**2.double check**

class Singleton{

**private volatile static Singleton instance** = null;//保证读取到是最新值。

    private Singleton() {

    }

    public static Singleton getInstance() {

        if(instance==null) {

            synchronized (Singleton.class) {

                if(instance==null)

                    instance = new Singleton();

            }

        }

        return instance;

    }

}