你真的了解volatile关键字吗？ - 简书

你真的了解volatile关键字吗?

Ruheng

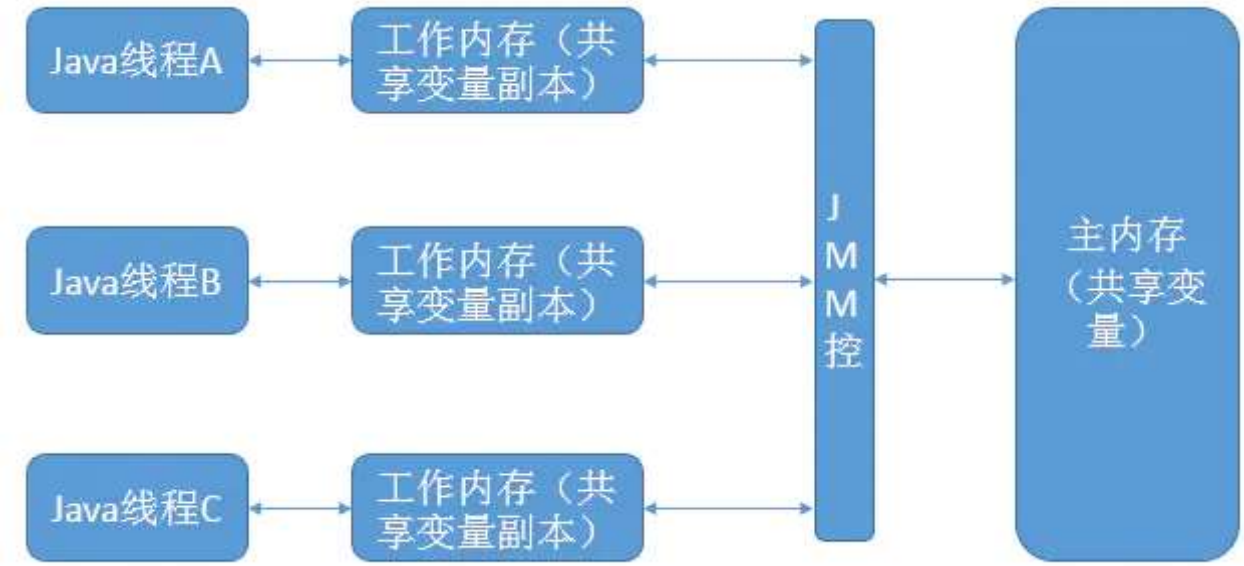
1 2017.03.26 13:49:33 字数 7,051 阅读 9,671

> volatile关键字经常在并发编程中使用，其特性是保证可见性以及有序性，但是关于volatile的使用仍然要小心，这需要明白volatile关键字的特性及实现的原理，这也是本篇文章的主要内容。

一、Java内存模型

想要理解volatile为什么能确保可见性，就要先理解Java中的内存模型是什么样的。

Java内存模型规定了**所有的变量都存储在主内存中。每条线程中还有自己的工作内存，线程的工作内存中保存了被该线程所使用到的变量（这些变量是从主内存中拷贝而来）。线程对变量的所有操作（读取，赋值）都必须在工作内存中进行。不同线程之间也无法直接访问对方工作内存中的变量，线程间变量值的传递均需要通过主内存来完成。**



基于此种内存模型，便产生了多线程编程中的数据"脏读"等问题。

举个简单的例子：在java中，执行下面这个语句：

```
1 | i  = 10;
```

执行线程必须先在自己的工作线程中对变量i所在的缓存行进行赋值操作，然后再写入主存当中。而不是直接将数值10写入主存当中。

比如同时有2个线程执行这段代码，假如初始时i的值为10，那么我们希望两个线程执行完之后i的值变为12。但是事实会是这样吗?

可能存在下面一种情况：初始时，两个线程分别读取i的值存入各自所在的工作内存当中，然后线程1进行加1操作，然后把i的最新值11写入到内存。此时线程2的工作内存当中i的值还是10，进行加1操作之后，i的值为11，然后线程2把i的值写入内存。

最终结果i的值是11，而不是12。这就是著名的缓存一致性问题。通常称这种被多个线程访问的变量为共享变量。

那么如何确保共享变量在多线程访问时能够正确输出结果呢?

在解决这个问题之前，我们要先了解并发编程的三大概念：**原子性，有序性，可见性**。

二、原子性

1.定义

原子性：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。

2.实例

一个很经典的例子就是银行账户转账问题：

比如从账户A向账户B转1000元，那么必然包括2个操作：从账户A减去1000元，往账户B加上1000元。

试想一下，如果这2个操作不具备原子性，会造成什么样的后果。假如从账户A减去1000元之后，操作突然中止。这样就会导致账户A虽然减去了1000元，但是账户B没有收到这个转过来的1000元。

所以这2个操作必须要具备原子性才能保证不出现一些意外的问题。

同样地反映到并发编程中会出现什么结果呢?

举个最简单的例子，大家想一下假如为一个32位的变量赋值过程不具备原子性的话，会发生什么后果?

```
1 | i = 9;
```

假若一个线程执行到这个语句时，我暂且假设为一个32位的变量赋值包括两个过程：为低16位赋值，为高16位赋值。

那么就可能发生一种情况：当将低16位数值写入之后，突然被中断，而此时又有一个线程去读取i的值，那么读取到的就是错误的数据。

3.Java中的原子性

在Java中，**对基本数据类型的变量的读取和赋值操作是原子性操作**，即这些操作是不可被中断的，要么执行，要么不执行。

上面一句话虽然看起来简单，但是理解起来并不是那么容易。看下面一个例子i：

请分析以下哪些操作是原子性操作：

```
x = 10;         //语句1
y = x;          //语句2
x++;            //语句3
x = x + 1;      //语句4
```

咋一看，可能会说上面的4个语句中的操作都是原子性操作。其实只有语句1是原子性操作，其他三个语句都不是原子性操作。

语句1是直接将数值10赋值给x，也就是说线程执行这个语句的会直接将数值10写入到工作内存中。

语句2实际上包含2个操作，它先要去读取x的值，再将x的值写入工作内存，虽然读取x的值以及将x的值写入工作内存 这2个操作都是原子性操作，但是合起来就不是原子性操作了。

同样的，**x++和 x = x+1包括3个操作：读取x的值，进行加1操作，写入新的值。**

所以上面4个语句只有语句1的操作具备原子性。

也就是说，**只有简单的读取、赋值（而且必须是将数字赋值给某个变量，变量之间的相互赋值不是原子操作）才是原子操作。**

从上面可以看出，Java内存模型只保证了基本读取和赋值是原子性操作，**如果要实现更大范围操作的原子性，可以通过synchronized和Lock来实现。由于synchronized和Lock能够保证任一时刻只有一个线程执行该代码块，那么自然就不存在原子性问题了，从而保证了原子性。**

关于synchronized和Lock的使用，参考：关于synchronized和ReentrantLock之多线程同步详解

三、可见性

1.定义

可见性是指当多个线程访问同一个变量时，一个线程修改了这个变量的值，其他线程能够立即看得到修改的值。

2.实例

举个简单的例子，看下面这段代码：

```
//线程1执行的代码
int i = 0;
i = 10;

//线程2执行的代码
j = i;
```

由上面的分析可知，当线程1执行 i =10这句时，会先把i的初始值加载到工作内存中，然后赋值为10，那么在线程1的工作内存当中i的值变为10了，却没有立即写入到主存当中。

此时线程2执行 j = i，它会先去主存读取i的值并加载到线程2的工作内存当中，注意此时内存当中i的值还是0，那么就会使得j的值为0，而不是10.

这就是可见性问题，线程1对变量i修改了之后，线程2没有立即看到线程1修改的值。

3.Java中的可见性

对于可见性，Java提供了volatile关键字来保证可见性。

当一个共享变量被volatile修饰时，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值。

而普通的共享变量不能保证可见性，**因为普通共享变量被修改之后，什么时候被写入主存是不确定的，当其他线程去读取时，此时内存中可能还是原来的旧值，因此无法保证可见性。**

另外，通过synchronized和Lock也能够保证可见性，synchronized和Lock能保证同一时刻只有一个线程获取锁然后执行同步代码，并且**在释放锁之前会将对变量的修改刷新到主存当中**。因此可以保证可见性。

四、有序性

1.定义

有序性：即程序执行的顺序按照代码的先后顺序执行。

2.实例

举个简单的例子，看下面这段代码：

```
int i = 0;

boolean flag = false;

i = 1;                //语句1
flag = true;          //语句2
```

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操

上面代码定义了一个int型变量，定义了一个boolean类型变量，然后分别对两个变量进行赋值操作。从代码顺序上看，语句1是在语句2前面的，那么JVM在真正执行这段代码的时候会保证语句1一定会在语句2前面执行吗？不一定，为什么呢？这里可能会发生指令重排序（Instruction Reorder）。

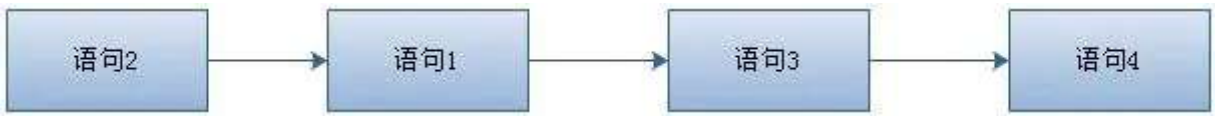
下面解释一下什么是指令重排序，**一般来说，处理器为了提高程序运行效率，可能会对输入代码进行优化，它不保证程序中各个语句的执行先后顺序同代码中的顺序一致，但是它会保证程序最终执行结果和代码顺序执行的结果是一致的。**

比如上面的代码中，语句1和语句2谁先执行对最终的程序结果并没有影响，那么就有可能在执行过程中，语句2先执行而语句1后执行。

但是要注意，虽然处理器会对指令进行重排序，但是它会保证程序最终结果会和代码顺序执行结果相同，那么它靠什么保证的呢？再看下面一个例子：

```
int a = 10;    //语句1
int r = 2;    //语句2
a = a + 3;    //语句3
r = a*a;     //语句4
```

这段代码有4个语句，那么可能的一个执行顺序是：



那么可不可能是这个执行顺序呢： 语句2 语句1 语句4 语句3

不可能，因为处理器在进行重排序时是会考虑指令之间的数据依赖性，如果一个指令Instruction 2必须用到Instruction 1的结果，那么处理器会保证Instruction 1会在Instruction 2之前执行。

虽然重排序不会影响单个线程内程序执行的结果，但是多线程呢？下面看一个例子：

```
//线程1:

context = loadContext();   //语句1
inited = true;             //语句2

 //线程2:
while(!inited ){
   sleep()
}
doSomethingwithconfig(context);
```

上面代码中，由于语句1和语句2没有数据依赖性，因此可能会被重排序。假如发生了重排序，在线程1执行过程中先执行语句2，而此是线程2会以为初始化工作已经完成，那么就会跳出while循环，去执行doSomethingwithconfig(context)方法，而此时context并没有被初始化，就会导致程序出错。

从上面可以看出，**指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。**

也就是说，**要想并发程序正确地执行，必须要保证原子性、可见性以及有序性。只要有一个没有被保证，就有可能会导致程序运行不正确。**

3.Java中的有序性

在Java内存模型中，允许编译器和处理器对指令进行重排序，但是重排序过程不会影响到单线程程序的执行，却会影响到多线程并发执行的正确性。

在Java里面，可以通过volatile关键字来保证一定的“有序性”。另外可以通过synchronized和Lock来保证有序性，很显然，synchronized和Lock保证每个时刻是有一个线程执行同步代码，相当于是让线程顺序执行同步代码，自然就保证了有序性。

另外，Java内存模型具备一些先天的“有序性”，**即不需要通过任何手段就能够得到保证的有序性，这个通常也称为 happens-before 原则。如果两个操作的执行次序无法从happens-before原则推导出来，那么它们就不能保证它们的有序性，虚拟机可以随意地对它们进行重排序。**

下面就来具体介绍下happens-before原则（先行发生原则）：

①程序次序规则：一个线程内，按照代码顺序，书写在前面的操作先行发生于书写在后面的操作

②锁定规则：一个unLock操作先行发生于后面对同一个锁额lock操作

③volatile变量规则：对一个变量的写操作先行发生于后面对这个变量的读操作

④传递规则：如果操作A先行发生于操作B，而操作B又先行发生于操作C，则可以得出操作A先行发生于操作C

⑤线程启动规则：Thread对象的start()方法先行发生于此线程的每个一个动作

⑥线程中断规则：对线程interrupt()方法的调用先行发生于被中断线程的代码检测到中断事件的发生

⑦线程终结规则：线程中所有的操作都先行发生于线程的终止检测，我们可以通过Thread.join()方法结束、Thread.isAlive()的返回值手段检测到线程已经终止执行

⑧对象终结规则：一个对象的初始化完成先行发生于他的finalize()方法的开始

这8条规则中，前4条规则是比较重要的，后4条规则都是显而易见的。

下面我们来解释一下前4条规则：

对于程序次序规则来说，就是一段程序代码的执行**在单个线程中看起来是有序的。**注意，虽然这条规则中提到"书写在前面的操作先行发生于书写在后面的操作"，这个应该是程序看起来执行的顺序是按照代码顺序执行的，**但是虚拟机可能会对程序代码进行指令重排序。**虽然进行重排序，但是最终执行的结果是与程序顺序执行的结果一致的，它只会对不存在数据依赖性的指令进行重排序。因此，**在单个线程中，程序执行看起来是有序执行的**，这一点要注意理解。事实上，**这个规则是用来保证程序在单线程中执行结果的正确性，但无法保证程序在多线程中执行的正确性。**

第二条规则也比较容易理解，也就是说无论在单线程中还是多线程中，**同一个锁如果处于被锁定的状态，那么必须先对锁进行了释放操作，后面才能继续进行lock操作。**

第三条规则是一条比较重要的规则。直观地解释就是，**如果一个线程先去写一个变量，然后一个线程去进行读取，那么写入操作肯定会先行发生于读操作。**

第四条规则实际上就是体现happens-before原则**具备传递性。**

五、深入理解volatile关键字

1.volatile保证可见性

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

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不能确保原子性

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，也不会导致主存中的值刷新，**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。

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。

解决方案：可以通过synchronized或lock，进行加锁，来保证操作的原子性。也可以通过AtomicInteger。

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保证有序性

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变量的读操作或者写操作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。**

六、volatile的实现原理

1 可见性

1.可见性

处理器为了提高处理速度，不直接和内存进行通讯，而是将系统内存的数据独到内部缓存后再进行操作，但操作完后不知什么时候会写到内存。

如果**对声明了volatile变量进行写操作时，JVM会向处理器发送一条Lock前缀的指令，将这个变量所在缓存行的数据写会到系统内存。** 这一步确保了如果有其他线程对声明了volatile变量进行修改，则立即更新主内存中数据。

** 但这时候其他处理器的缓存还是旧的，所以在多处理器环境下，为了保证各个处理器缓存一致，每个处理会通过嗅探在总线上传播的数据来检查 自己的缓存是否过期，**当处理器发现自己缓存行对应的内存地址被修改了，就会将当前处理器的缓存行设置成无效状态，当处理器要对这个数据进行修改操作时，**会强制重新从系统内存把数据读到处理器缓存里。** 这一步确保了其他线程获得的声明了volatile变量都是从主内存中获取最新的。

2.有序性

Lock前缀指令实际上相当于一个内存屏障（也成内存栅栏），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。

七、volatile的应用场景

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

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;
 //线程1
while(!flag){
    doSomething();
}
  //线程2
public void setFlag() {
    flag = true;
}
```

根据状态标记，终止线程。

②.单例模式中的double check

```
class Singleton{
    private volatile static Singleton instance = null;

    private Singleton() {

    }

    public static Singleton getInstance() {
        if(instance==null) {
            synchronized (Singleton.class) {
                if(instance==null)
                    instance = new Singleton();
            }
        }
        return instance;
    }
}
```

为什么要使用volatile 修饰instance?

主要在于instance = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情:

1.给 instance 分配内存

2.调用 Singleton 的构造函数来初始化成员变量

3.将instance对象指向分配的内存空间（执行完这步 instance 就为非 null 了）。

但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。

参考文章

Java并发编程：volatile关键字解析
【死磕Java并发】-----深入分析volatile的实现原理
Java并发机制的底层实现原理
Volatile的实现原理

"小礼物走一走，来简书关注我"


共2人赞赏

Ruheng
总资产1,570　共写了21.1W字　获得8,901个赞　共8,521个粉丝