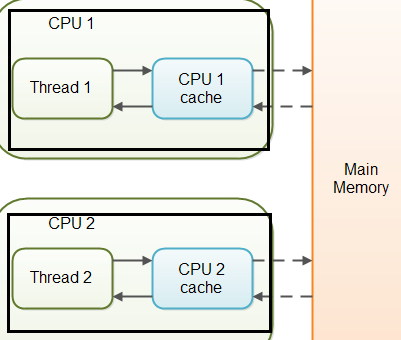
# Volatile可见性

①基本概念：线程之间的可见性，一个线程修改的状态对另一个线程时可见的，也就是一个线程修改的结果，另一个线程马上就能看到。

②实现原理：

当对非volatile变量进行读写的时候，每个线程先从主存拷贝变量到线程工作内存（cache），如果计算机有多个CPU，每个线程可能在不同的CPU上被处理，这就意味着每个线程可以拷贝到不同的CPU的cache中

Volatile变量不会被缓存到在寄存器或者对其他处理器不可见的地方，保证了每次读写变量都从主存中读，跳过了CPU cache这一步，当一个线程修改了该变量的值，对于其他线程是可以立即得知的



# 禁止指令重排

①基本概念：

指令重排序是JVM为了优化指令、提高程序运行效率，在不影响单线程程序执行结果的前提下，尽可能地提高并行度，指令重排包括编译时重排序和运行时重排序。

例如：

|  |
| --- |
| double r = 2.1; //(1)  double pi= 3.14; //(2)  double area = p \* r \* r; //(3) |

虽然代码语句的定义顺序为1-2-3，但是计算顺序1-2-3与2-1-3对结果并没有影响，所以编译时和运行时可以根据需要对1,2语句进行重排序

②指令重排带来的问题

如果一个操作不是原子的，就会给JVM留下重拍的机会

例如：

|  |
| --- |
| Thread1{  sum = count();  inited = true;  }  Thread2{  If(inited){  func(sum);  }  } |

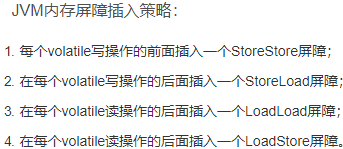
如果Thread1中的指令发生重排，那么Thread2中可能拿到一个未被初始化或者初始化未完成的sum变量，从而引发程序错误

Volatile在双重检查加锁（DCL）的单例中的使用

|  |
| --- |
| public class Singleton {  public static volatile Singleton singleton;  /\*\*  \* 构造函数私有，禁止外部实例化  \*/  private Singleton() {};  public static Singleton getInstance() {  if (singleton == null) {  synchronized (singleton) {  if (singleton == null) {  singleton = new Singleton();  }  }  }  return singleton;  }  }  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。 |

③禁止指令重排的原理

volatile关键字提供内存屏障的方式来防止指令被重排，编译器在生成字节码文件时，会在指令序列中插入内存屏障来禁止特定类型的处理器重排序。



# 适用场景

（1）volatile是轻量级同步机制。在访问volatile变量时不会执行加锁操作，因此也就不会使执行线程阻塞，是一种比synchronized关键字更轻量级的同步机制。

（2）volatile\*\*无法同时保证内存可见性和原子性。加锁机制既可以确保可见性又可以确保原子性，而volatile变量只能确保可见性\*\*。

（3）volatile不能修饰写入操作依赖当前值的变量。声明为volatile的简单变量如果当前值与该变量以前的值相关，那么volatile关键字不起作用，也就是说如下的表达式都不是原子操作：“count++”、“count = count+1”。

（4）当要访问的变量已在synchronized代码块中，或者为常量时，没必要使用volatile；

（5）volatile屏蔽掉了JVM中必要的代码优化，所以在效率上比较低，因此一定在必要时才使用此关键字。

# Volatile的线程安全性

下面用i++的例子进行分析，再讲线程安全之前，理解一下i=i++的内存执行过程

|  |
| --- |
| 2 public static void main(String[] args){  3 int i = 234;  4 i = i++; |
| }  编译后的字节码文件  0: sipush 234//将常量234压入操作数栈  3: istore\_1//将操作数栈出栈，值赋值给局部变量区的1号位置即i  4: iload\_1//然后将变量1的i的值，压入操作数栈  5: iinc 1, 1//将局部变量区的一号变量i数值上加1  8: istore\_1//将操作数栈出栈，值赋值给局部变量区的1号位置  使用局部变量区和操作数栈进行分析  局部变量区   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 0: sipush 234 | 3: istore\_1 | 4: iload\_1 | 5: iinc 1, 1 | 8: istore\_1 | | 1号变量i | 234 | 234 | 234+1 | 234 |   操作数栈   |  |  |  |  |  | | --- | --- | --- | --- | --- | | 234 | 出栈 | 234 | 234 | 出栈 | |

所以i=i++的值是不会发生变化的。

Volatile只能保证变量的可见性，无法保证对变量的操作的原子性。

i++的执行过程其实包含三个步骤

①从内存中读取i当前的值

②局部变量区变量i加1

③把修改后的值刷新到内存中

这三个步骤不是原子性操作，volatile只能保证步骤一和步骤三的改变立即可见，但是无法决定步骤二，当多线程同时执行的时候，所出现的交叉修改，所以无法保证线程安全性。