synchronized

synchronized可作用于一段代码或方法，既可以保证可见性，又能够保证原子性

可见性体现在：通过synchronized或者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中

原子性表现在：要么不执行，要么执行到底

1. **不能继承**

**虽然可以使用synchronized来定义方法，但synchronized并不属于方法定义的一部分，因此，synchronized关键字不能被继承。如果在父类中的某个方法使用了synchronized关键字，而在子类中覆盖了这个方法，在子类中的这个方法默认情况下并不是同步的，而必须显式地在子类的这个方法中加上synchronized关键字才可以。当然，还可以在子类方法中调用父类中相应的方法**

1. **synchronized(this)**//this当前类的对象
2. **当一个线程访问对象的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该对象中的非synchronized(this)同步代码块**
3. **无论synchronized关键字加在方法上还是对象上，如果它作用的对象是非静态的，则它取得的锁是对象；如果synchronized作用的对象是一个静态方法或一个类(test.class)，则它取得的锁是类锁，该类所有的对象同一把锁**

**synchronized关键字的用法？**

synchronized关键字可以将对象或者方法标记为同步，以实现对对象和方法的互斥访问可以用synchronized(对象) { … }定义同步代码块

synchronized用于锁对象

锁的对象包括：

* This（当前对象）
* 临界资源对象(锁的是堆内存中的对象，而不是引用)

Object object = new Object();

T methodName(){

synchronized(object){

//do something

}

* Class 类对象（当前类型的类对象xxx.class）

**synchronized 和java.util.concurrent.locks.Lock的异同？**

相同点：Lock 能完成synchronized所实现的所有功能；

不同点：一个是关键字一个是类

synchronized会自动释放锁，而Lock一定要求手工释放，最好在finally 块中释放（这是释放外部资源的最好的地方）

lock可判断锁状态，具有公平性（长时间没运行的线程可优先运行）

*Volatile*

volatile是变量修饰符，其修饰的变量具有可见性

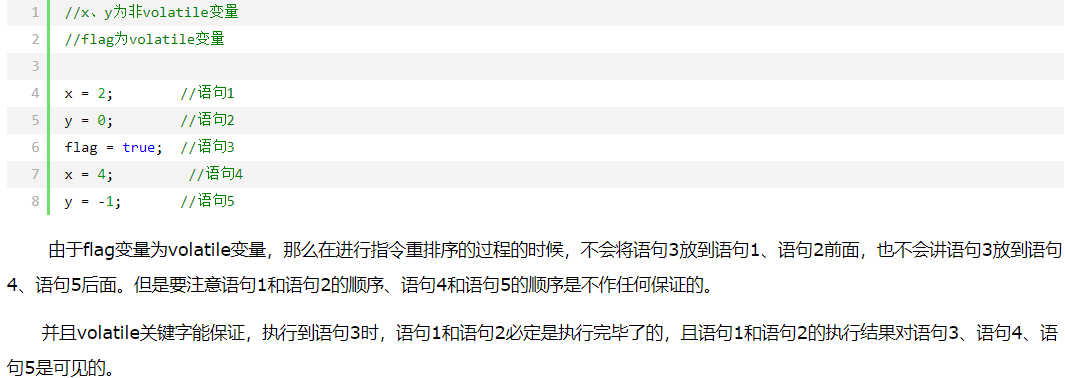
可见性也就是说一旦某个线程修改了该被volatile修饰的变量，它会保证修改的值会立即被更新到主存，当有其他线程需要读取时，它会去内存中读取新值(这新值对其他线程来说是立即可见的)

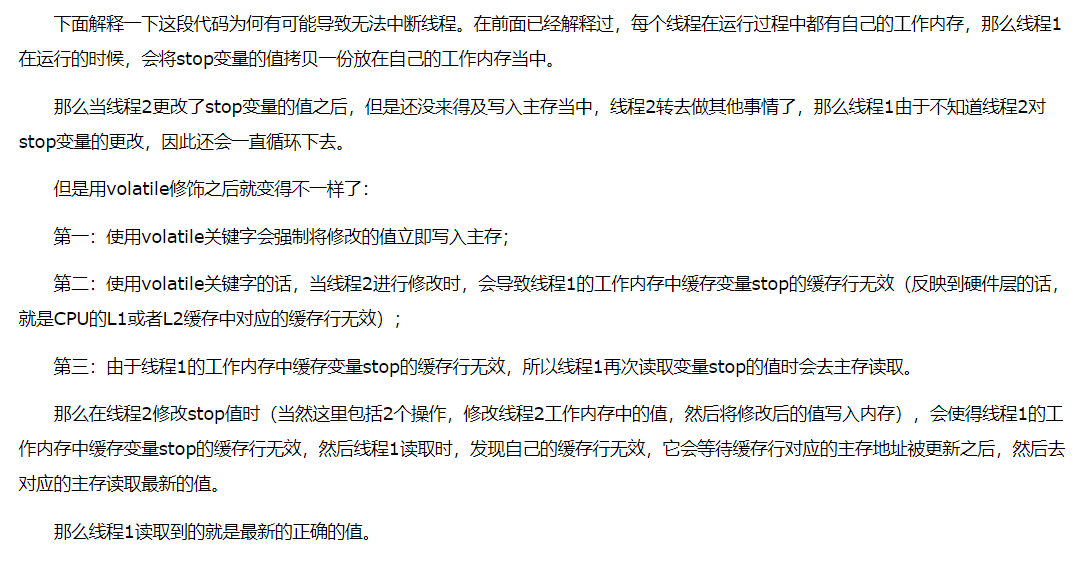
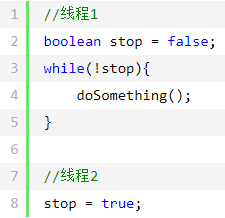
在Java中为了加快程序的运行效率，对一些变量的操作通常是在该线程的寄存器或是CPU缓存上进行的，之后才会同步到主存中，而加了volatile修饰符的变量则是直接读写主存

volatile可以禁止进行指令重排

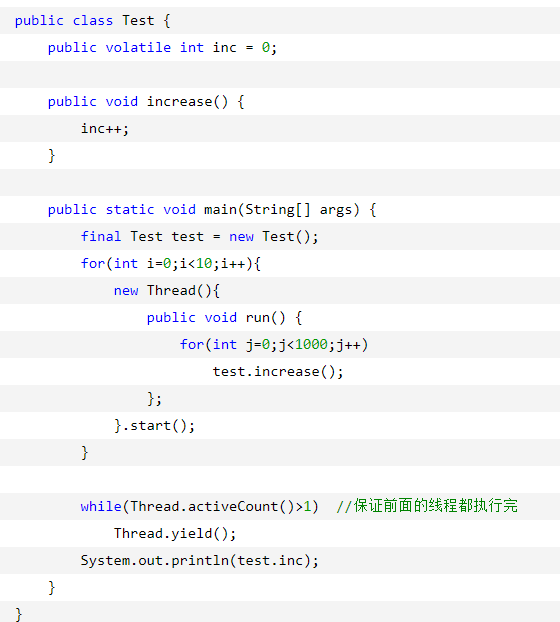
指令重排序不会影响单个线程的执行，但是会影响到线程并发执行的正确性。

1. 程序执行到volatile修饰变量的读操作或者写操作时，在其前面的操作肯定已经完成，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





无原子性



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

