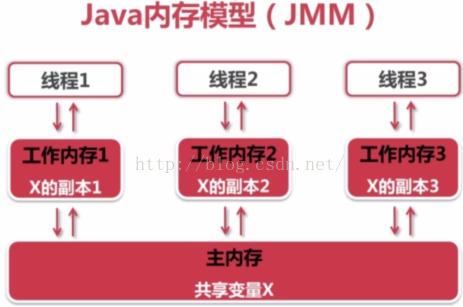
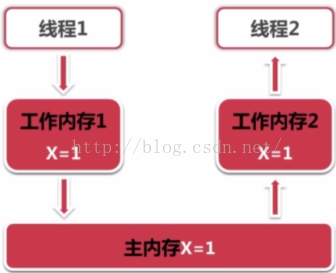
**Java多线程之内存可见性和原子性：Synchronized和Volatile的比较**

      在说明Java多线程内存可见性之前，先来简单了解一下Java内存模型。

     （1）Java所有变量都存储在主内存中  
     （2）每个线程都有自己独立的工作内存，里面保存该线程的使用到的变量副本（该副本就是主内存中该变量的一份拷贝）



   （1）线程对共享变量的所有操作都必须在自己的工作内存中进行，不能直接在主内存中读写  
   （2）不同线程之间无法直接访问其他线程工作内存中的变量，线程间变量值的传递需要通过主内存来完成。  
线程1对共享变量的修改，要想被线程2及时看到，必须经过如下2个过程：  
   （1）把工作内存1中更新过的共享变量刷新到主内存中  
   （2）将主内存中最新的共享变量的值更新到工作内存2中



**可见性与原子性**

**可见性：一个线程对共享变量的修改，更够及时的被其他线程看到  
   原子性：原子操作是不可分割的,在执行完毕不会被任何其它任务或事件中断。比如赋值或者return。**比如"a = 1;"和 "return a;"这样的操作都具有原子性。类似"a += b"这样的操作不具有原子性，在某些JVM中"a += b"可能要经过这样三个步骤：  
①　取出a和b  
②　计算a+b  
③　将计算结果写入内存

**(1)Synchronized：保证可见性和原子性**  
    Synchronized能够实现原子性和可见性；在Java内存模型中，synchronized规定，**线程在加锁→先清空工作内存→在主内存中拷贝最新变量的副本到工作内存→执行完代码→将更改后的共享变量的值刷新到主内存中→释放互斥锁。**

**(2)Volatile：保证可见性，但不保证操作的原子性**  
    Volatile实现内存可见性是通过**store和load指令完成的**；也就是对volatile变量执行写操作时，会在写操作后加入一条**store指令，即强迫线程将最新的值刷新到主内存中**；而在读操作时，会加入一条load指令，**即强迫从主内存中读入变量的值**。但volatile不保证volatile变量的原子性，例如：

**[java]** [view plain](https://blog.csdn.net/guyuealian/article/details/52525724) [copy](https://blog.csdn.net/guyuealian/article/details/52525724)

1. Private **int** Num=0;
2. Num++;//Num不是原子操作

    Num不是原子操作，因为其可以分为：读取Num的值，将Num的值+1，写入最新的Num的值。  
    对于Num++;操作，线程1和线程2都执行一次，最后输出Num的值可能是：1或者2  
   【解释】输出结果1的解释：当线程1执行Num++;语句时，先是读入Num的值为0，倘若此时让出CPU执行权，线程获得执行，线程2会重新从主内存中，读入Num的值还是0，然后线程2执行+1操作，最后把Num=1刷新到主内存中； 线程2执行完后，线程1由开始执行，但之前已经读取的Num的值0，所以它还是在0的基础上执行+1操作，也就是还是等于1，并刷新到主内存中。所以最终的结果是1

    一般在多线程中使用volatile变量，为了安全，对变量的写入操作**不能依赖当前变量的值：如Num++或者Num=Num\*5这些操作。比如用于if判断，boolean A ==true；这种就可以用volatile。**

**(3)Synchronized和Volatile的比较**  
    1）Synchronized保证内存可见性和操作的原子性  
    2）Volatile只能保证内存可见性  
    3）Volatile不需要加锁，比Synchronized更轻量级，并不会阻塞线程（volatile不会造成线程的阻塞；synchronized可能会造成线程的阻塞。）  
    4）volatile标记的变量不会被编译器优化（通过volatile保证了有序性）,而synchronized标记的变量可以被编译器优化（如编译器重排序的优化）. （**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**）所以根据before-happen原则，也可以保证有序性。

    5）**volatile是变量修饰符，仅能用于变量，而synchronized是一个方法或块的修饰符。**  
      volatile本质是在告诉JVM当前变量在寄存器中的值是不确定的，使用前，需要先从主存中读取，因此可以实现可见性。而对n=n+1,n++等操作时，volatile关键字将失效，不能起到像synchronized一样的线程同步（原子性）的效果。

【相关习题】

**(1)下列说法不正确的是（）**  
A.当两个并发线程访问同一个对象object中的这个synchronized(this)同步代码块时，一个时间内只能有一个线程得到执行。  
B.当一个线程访问object的一个synchronized(this)同步代码块时，另一个线程仍然可以访问该object中的非synchronized(this)同步代码块。  
C.当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问不会被阻塞。  
D.当一个线程访问object的一个synchronized(this)同步代码块时，它就获得了这个object的对象锁。结果，其它线程对该object对象所有同步代码部分的访问都被暂时阻塞。  
答案：C，当一个线程访问object的一个synchronized(this)同步代码块时，其他线程对object中所有其它synchronized(this)同步代码块的访问将会被阻塞。  
**(2)下面叙述错误的是：**  
A.通过synchronized和volatile都可以实现可见性  
B.不同线程之间可以直接访问其他线程工作内存中的变量  
C.线程对共享变量的所有操作都必须在自己的工作内存中进行  
D.所有的变量都存储在主内存中  
答案：B，不同线程之间无法直接访问其他线程工作内存中的变量

**研究 Java Synchronize 和 Lock 的区别与用法**

在分布式开发中，锁是线程控制的重要途径。

Java为此也提供了2种锁机制，synchronized和lock。做为Java爱好者，自然少不了对比一下这2种机制，也能从中学到些分布式开发需要注意的地方。  
   
我们先从最简单的入手，逐步分析这2种的区别。  
   
一、synchronized和lock的**用法区别**  
   
**synchronized：在需要同步的对象中加入此控制，synchronized可以加在方法上，也可以加在特定代码块中，括号中表示需要锁的对象。**  
   
**lock：需要显示指定起始位置和终止位置。一般使用ReentrantLock类做为锁，多个线程中必须要使用一个ReentrantLock类做为对象才能保证锁的生效。**且在加锁和解锁处需要通过**lock()和unlock()显示指出**。所以**一般会在finally块中写unlock()以防死锁**。  
   
用法区别比较简单，这里不赘述了，如果不懂的可以看看Java基本语法。  
   
二、synchronized和lock**性能区别**  
   
**synchronized是托管给JVM执行的**，而**lock是java写的控制锁的代码**。在Java1.5中，synchronize是性能低效的。因为这是一个重量级操作，需要调用操作接口，导致有可能加锁消耗的系统时间比加锁以外的操作还多。

但是到了**Java1.6，发生了变化。synchronized在语义上很清晰，可以进行很多优化，**有**适应自旋，锁消除，锁粗化，轻量级锁，偏向锁**等等。导致在Java1.6上synchronize的性能并不比Lock差。官方也表示，他们也更支持synchronize，在未来的版本中还有优化余地。  
   
说到这里，还是想提一下这2中机制的具体区别。据我所知**，synchronized原始采用的是CPU悲观锁机制，即线程获得的是独占锁。**独占锁意味着其他线程只能依靠阻塞来等待线程释放锁。而在CPU转换线程阻塞时会引起线程上下文切换，**当有很多线程竞争锁的时候，会引起CPU频繁的上下文切换导致效率很低。**   
而**Lock用的是乐观锁方式**。所谓乐观锁就是，每次不加锁而是假设没有冲突而去完成某项操作**，如果因为冲突失败就重试，直到成功为止**。乐观锁实现的机制就是CAS操作（Compare and Swap）。我们可以进一步研究ReentrantLock的源代码，会发现其中比较重要的获得锁的一个方法是compareAndSetState。这里其实就是调用的CPU提供的特殊指令。  
   
现代的CPU提供了指令，可以自动更新共享数据，而且能够检测到其他线程的干扰，而 **compareAndSet() 就用这些代替了锁定。这个算法称作非阻塞算法，**意思是一个线程的失败或者挂起不应该影响其他线程的失败或挂起的算法。

三、synchronized和lock用途区别  
   
synchronized原语和ReentrantLock在一般情况下没有什么区别，但是在非常复杂的同步应用中，请考虑使用ReentrantLock，特别是遇到下面几种需求的时候。  
   
1.某个线程在等待一个锁的控制权的这段时间需要中断  
2.需要分开处理一些wait-notify，ReentrantLock里面的Condition应用，能够控制notify哪个线程  
**3.具有公平锁功能，每个到来的线程都将排队等候**  
   
先说第一种情况，ReentrantLock的lock机制有2种，忽略中断锁和响应中断锁，这给我们带来了很大的灵活性。

比如：如果A、B2个线程去竞争锁，A线程得到了锁，B线程等待，但是A线程这个时候实在有太多事情要处理，就是一直不返回，B线程可能就会等不及了，想中断自己，不再等待这个锁了，转而处理其他事情。这个时候ReentrantLock就提供了2种机制，

**第一，B线程中断自己（或者别的线程中断它）**，但是**ReentrantLock不去响应，**继续让B线程等待，你再怎么中断，我全当耳边风（synchronized原语就是如此）；第二，**B线程中断自己（或者别的线程中断它），ReentrantLock处理了这个中断**，并且不再等待这个锁的到来，完全放弃。  
   
这里来做个试验，首先搞一个Buffer类，它有读操作和写操作，为了不读到脏数据，写和读都需要加锁，我们先用synchronized原语来加锁

我们期待“读”这个线程能退出等待锁，可是事与愿违，一旦读这个线程发现自己得不到锁，就一直开始等待了，就算它等死，也得不到锁，因为写线程要21亿秒才能完成 T\_T ，即使我们中断它，它都不来响应下，看来真的要等死了。这个时候，ReentrantLock给了一种机制让我们来响应中断，让“读”能伸能屈，勇敢放弃对这个锁的等待。我们来改写Buffer这个类，就叫BufferInterruptibly吧，可中断缓存。

**这次“读”线程接收到了lock.lockInterruptibly()中断，并且有效处理了这个“异常”。**

至于第二种情况，ReentrantLock可以与Condition的配合使用，Condition为ReentrantLock锁的等待和释放提供控制逻辑。  
   
例如，使用ReentrantLock加锁之后，可以通过它自身的**Condition.await()方法释放该锁，线程在此等待Condition.signal()方法，然后继续执行下去。**await方法需要放在while循环中，因此，在不同线程之间实现并发控制，还需要一个volatile的变量，boolean是原子性的变量。

调用spillDone.await()时可以释放spillLock锁，线程进入阻塞状态，而等待其他线程的 spillDone.signal()操作时，就会唤醒线程，重新持有spillLock锁。  
   
这里可以看出，利用lock可以使我们多线程交互变得方便，而使用synchronized则无法做到这点。  
   
最后呢，**ReentrantLock这个类还提供了2种竞争锁的机制**：**公平锁和非公平锁。**这2种机制的意思从字面上也能了解个大概：即对于多线程来说，**公平锁会依赖线程进来的顺序，后进来的线程后获得锁**。而**非公平锁的意思就是后进来的锁也可以和前边等待锁的线程同时竞争锁资源**。对于效率来讲，当然是非公平锁效率更高，因为公平锁还要判断是不是线程队列的第一个才会让线程获得锁。