synchronized、ReentrantLock、volatile、atomic原理分析

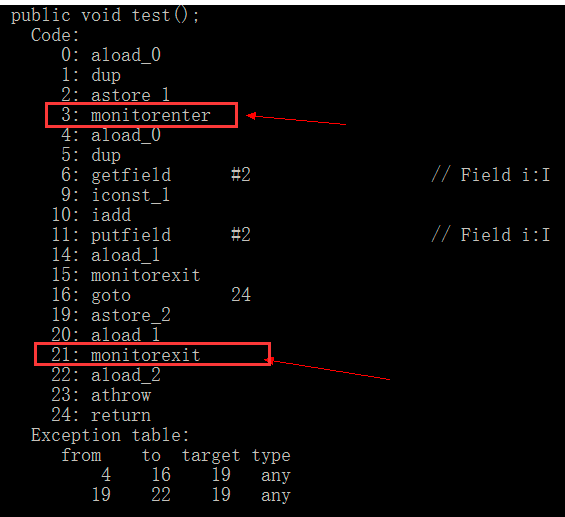
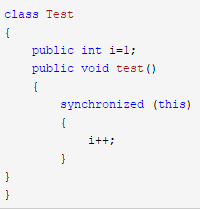
# synchronized

## 作用：隐式锁，实现同步，保证线程安全。

## 原理：

* synchronized关键字是通过**字节码指令**来实现的；synchronized关键字**编译后**会在同步块前后形成**monitorenter和monitorexit**两个字节码指令；
* 执行monitorenter指令时需要先获得对象的锁（每个对象有一个监视器锁monitor），如果这个对象没被锁或者当前线程已经获得此锁（也就是重入锁），那么锁的计数器+1。如果获取失败，那么当前线程阻塞，直到锁被对另一个线程释放；执行monitorexit指令时，计数器减一，当为0的时候锁释放。

测试代码：



# ReentrantLock：显式加锁

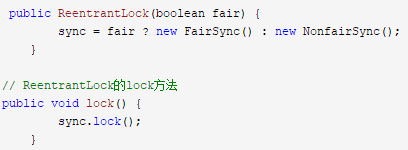
## 作用：显式加锁

## 原理(重点)

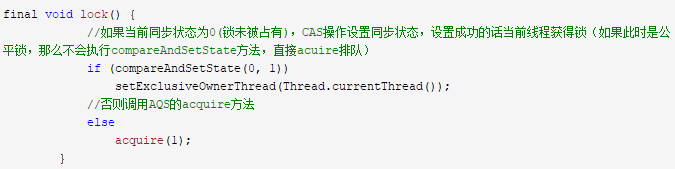
通过**同步器AQS**（**AbstractQueuedSynchronized类**）来实现的，AQS根本上是通过一个双向队列来实现的；

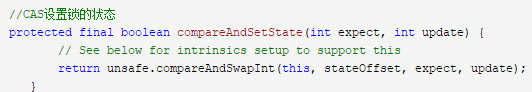
线程构造成一个节点Node，一个线程先尝试获得锁，如果获取锁失败，就将该线程加到队列尾部；

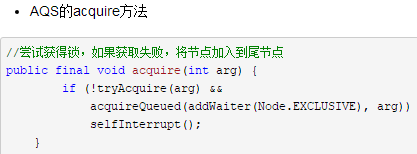
非公平锁的lock方法,调用的**sync（NonfairSync和fairSync的父类）的lock方法。**



**NonfairSync**的**lock**方法，acquire的是Sync的父类AQS的acquire方法。

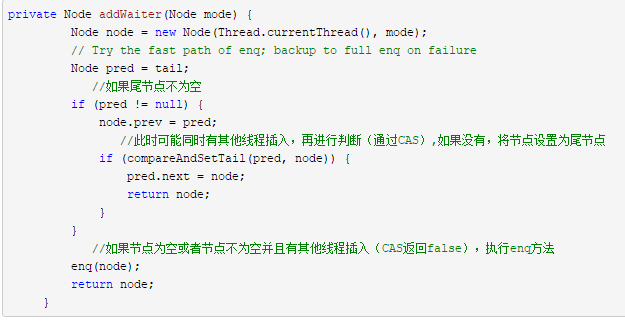




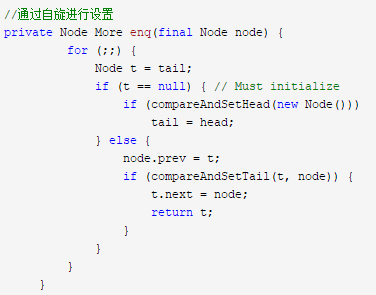




如果获取锁失败，将节点加入尾节点



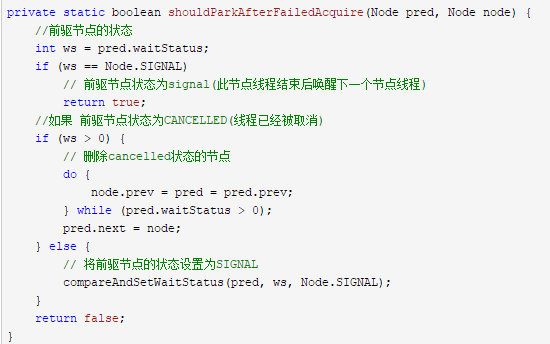
如果节点为空或者节点不为空并且有其他线程插入（CAS返回false），执行enq



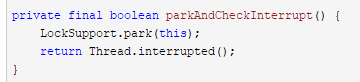
进入队列的线程尝试获得锁



线程是否可以挂起



挂起当前线程,返回线程中断状态并重置



# volatile关键字

## 作用：保证变量的可见性。

* **volatile关键字的作用**：保证变量对所有的线程的可见性，当一个线程修改了这个变量的值，其他线程可以立即知道这个新值（之所以有可见性的问题，是因为java的内存模型）；

## 原理：

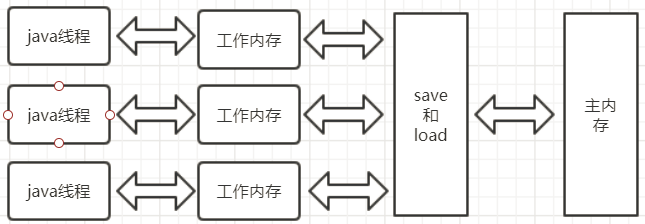
* **volatile关键字的原理**：

所有变量都存在**主内存**，每条线程有**自己的工作内存**，工作内存保存了被该线程使用的变量的主内存副本拷贝；线程对变量的所有操作都必须在**工作内存中**进行，不能直接读写主内存的变量，也就是必须先通过工作内存。

一个线程不能访问另一个线程的工作内存；

volatile保证了变量更新的时候**能够立即同步到主内存**，**使用变量的时候能立即从主内存刷新到工作内存**，这样就**保证了变量的可见性**；

实际上是通过**内存屏障**来实现的。语义上，内存屏障之前的**所有写操作**都要写入内存；内存屏障之后的读操作都可以获得同步屏障之前的写操作的结果。



# Atomic：

## 作用

* **Atomic的作用**：当有多个线程同时对单个（包括基本类型及引用类型）变量进行操作时，具有排他性，即当多个线程同时对该变量的值进行更新时，仅有一个线程能成功，而未成功的线程可以像自旋锁一样，继续尝试，一直等到执行成功。

**自我感觉：由于Atomic的变量都是volatile修饰，可以看做是volatile的升级版。**

## 原理

**CAS操作（compare and swap 比较并交换），**是通过**一个cpu指令**实现的，这个指令是一个**原子指令**，指令有3个操作数ABC，A为内存位置，B为预期值，C为新值，如果A符合旧预期值B，那么用V更新A的值，如果不符合就不更新，这个过程是原子操作。

所以我们并没有通过代码来实现同步，而是通过**硬件级别的cpu指令**来实现的，并不像synchronized一样阻塞线程。

具体参见AtomicInteger的源码：



如果此时有两个线程，线程A得到current值为1，线程B得到current值也为2，此时线程A执行CAS操作，成功将值改为2，而此时线程B执行CAS操作，发现此时内存中的值并不是读到current值1，所以返回false，此时线程B继续进行循环，最后成功加1