## Synchronized

1. **Synchronized使用场景**

Synchronized一般用在对象方法，类方法，和同步块中。但是在方法上使用Synchronized和在同步块中使用是不一样的，方法上采用的是字节码中标志位ACC\_SYNCHRONIZED来进行同步。而同步代码块则采用对象头中的锁指针指向一个monitor来完成同步。

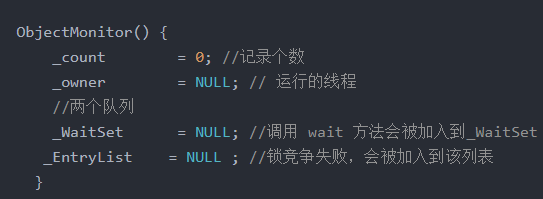
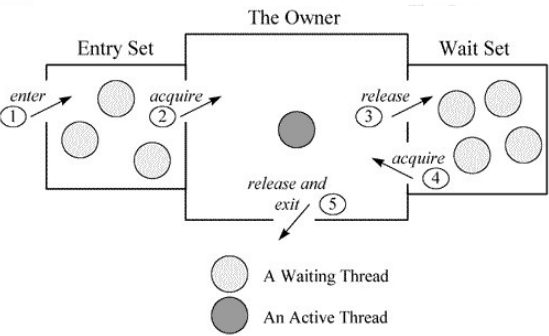
1. **对象与monitor对象**

对象在内存中存储的布局可以分为三块区域：对象头，实例数据，对齐填充。对象头可以分为“Mark Word”和类型指针。“Mark Word”中存储对象自身运行时的数据，如哈希码，GC分代年龄，锁的标志位，线程持有的锁，偏向锁的ID等，为了提高利用率，“Mark Word”是非固定的数据结构：

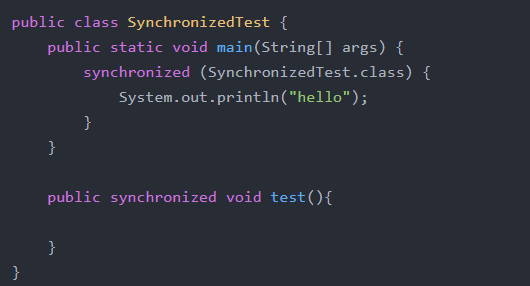


Monitor对象是用来实现重量级锁的关键对象，锁的标志位是10，存放着指向monitor对象的指针。每个对象都有一个monitor对象与之关联。Monitor对象可以和对象一起创建销毁，或者当线程尝试获取对象锁的时候，自动生成。

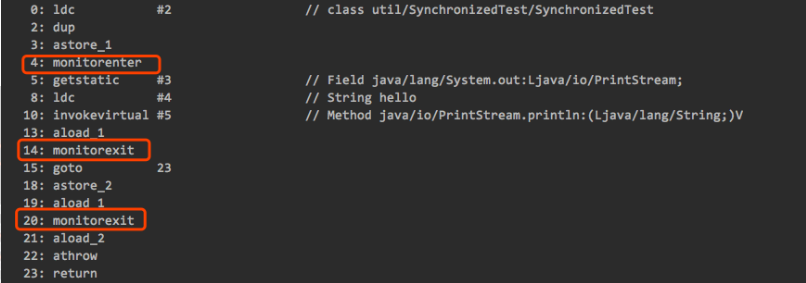
Monitor对象的结构如下：



1. **Synchronized代码块原理**



反编译上述代码得到如下字节码：



当执行monitorenter指令的时候：

* 当前线程尝试去获取monitor的持有权，当monitor的计数器为0，那线程可以成功获取，并将计数器设为1，取锁成功。
* 如果当前线程已经持有了monitor，那么可以重入这个monitor，计数器也会+1。
* 倘若其他线程已经拥有monitor的所有权，那么当前线程就会阻塞，直到别的线程执行monitorexit方法，并将count置为0，其他线程才可以争取该monitor。编译器会确保每个monitorenter都有一个monitorexit与之对应。意味着当synchronized方法里面出现了异常，当前线程会自动放弃持有的锁。

1. **偏向锁：**

当锁对象第一次被线程获取的时候，虚拟机将会把对象头中的标志位设为“01”，即偏向模式。同时使用CAS操作把获取到这个锁的线程ID记录在对象的Mark Word之中，同时会在栈帧中记录偏向的锁的threadId。如果CAS成功，持有偏向锁的线程每次进入这个锁相关的同步块，虚拟机可以不做任何同步。

当另一个线程尝试获取这个锁的时候，偏向模式就宣告结束。根据偏向对象目前是否处于被锁定的状态，撤销偏向锁后恢复到未锁定（01）或轻量级锁定（00）状态。

1. **轻量级锁：**

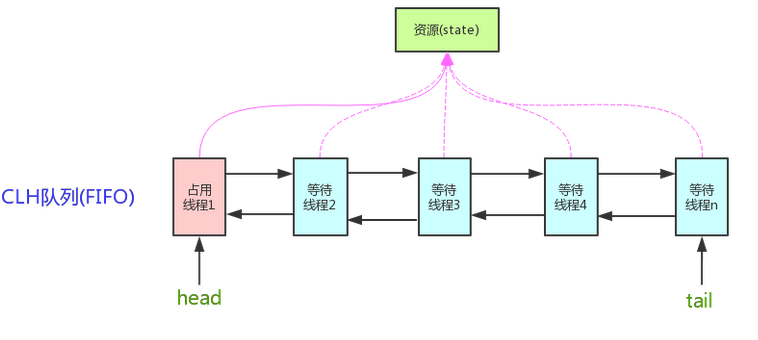
虚拟机首先将在当前线程的栈帧中建立一个名为锁的记录（Lock Record）的空间，用于存储锁对象目前的Mark Word的拷贝，然后虚拟机使用CAS操作尝试将对象的Mark Word更新为指向Lock Record的指针。如果成功，那么这个线程就拥有了该对象锁，并且对象Mard Word的标志位转变为00，即表示此对象处于轻量级锁定状态。如果这个操作失败，虚拟机首先检查对象的Mark Word是否指向当前线程的栈帧，如果指向，说明当前线程已经拥有了这个对象的锁，那就可以直接进入同步块，否则说明这个锁对象已经被其他线程抢占。如果有两条以上的线程用同一个锁，那么轻量级锁就不再生效，要膨胀为重量级锁。

1. **自旋锁与适应性自旋锁**

当线程尝试获取同步资源失败的时候，自旋锁会让线程进入自旋，意味着不放弃CPU片段。如果在自旋完成后，前面锁定同步资源的线程已经释放了锁，那么当前线程可以不必要阻塞，而是直接获取同步资源，从而避免了线程切换的开销。自旋锁虽然避开了线程切换，但是要占用处理器时间，如果被锁住的时间比较长，那么自旋锁就会白白浪费处理器资源。所以自旋等待的时间必须要有限度，如果超过了一定值，那么就挂起该线程。

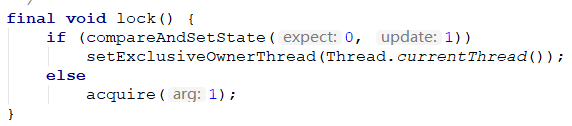
适应性自旋锁：适应性意味着自旋的次数不再固定，而是由前面的自旋成功的几率来决定。

## AQS



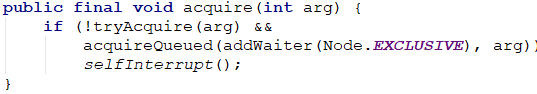
AQS全称为AbstractQueuedSynchronizer，是一个抽象类，提供了一些借口供子类使用。在ReentrantLock中，创建了两个内部类NonfairSync和FairSync，这两个类称为同步器，均继承自AbstractQueuedSynchronizer，分别用于实现公平锁和非公平锁。我们以NoFairSync为例，根据他的情况来一步步的解析AQS内部原理。

1. **NonfairSync中的lock：**

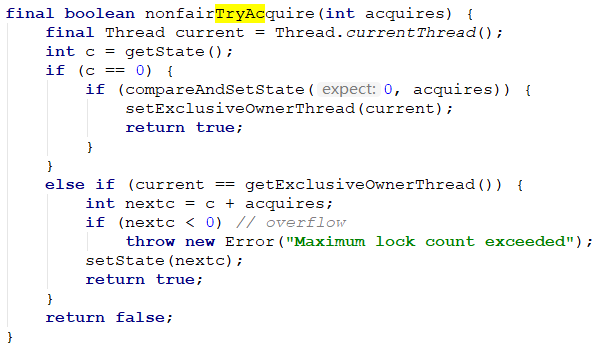


对于非公平锁，在进入同步块之前就会尝试去获取锁，如果获取成功，那么就将锁中的线程设置为当前线程，否则调用acquire方法。

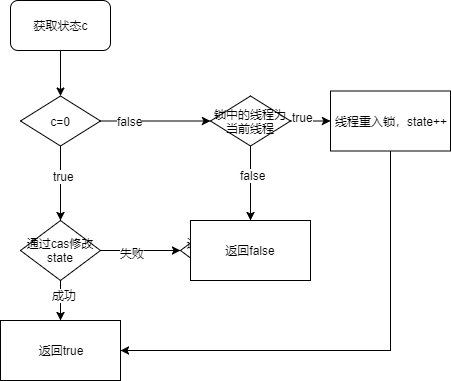
acquire方法：



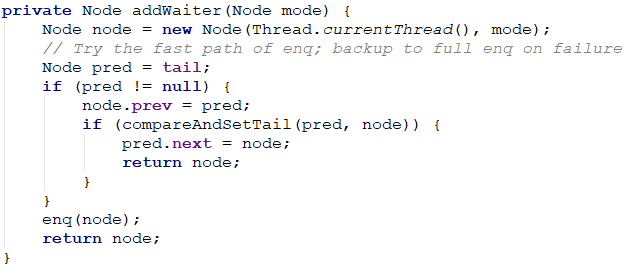
tryAcquire方法会尝试进行锁的获取，如果同步器中的状态==0，那么说明可以尝试去获取该锁，通过CAS来进行锁的获取，成功了那么当前线程就持有锁；如果同步器状态不 !=0 ，再判断锁中的线程是否是当前线程，如果是，那么就是锁的重入，状态量++，否则返回false，获取失败。



流程图如下：

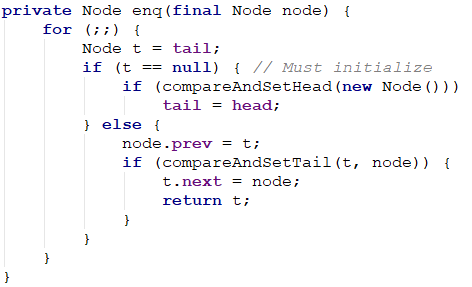


当尝试获取失败之后，会调用addWaiter方法，将当前节点加入到队列中



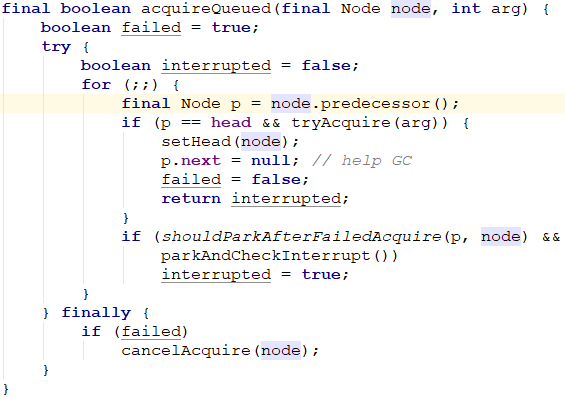
首先获取队尾节点，如果不为null，就尝试通过CAS在队尾添加node，添加成功就返回node，如果添加失败，或者说队尾node为null，那么就调用enq方法，enq方法能够确保node节点能够插入到链表中，调用完后返回node节点。

enq方法如下：

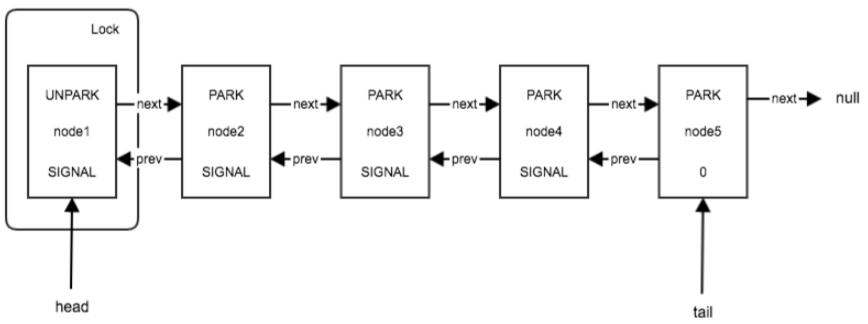


进入方法会直接进入死循环，判断队尾节点是否为null，如果为null，那么就创建一个队头元素，然后将尾节点指向队头元素，进入下一个循环。如果队尾元素不为null，那么就将node节点添加到队尾，最后将队尾的前一个元素返回（这个无所谓，因为调用者并没有接受enq方法的返回值）。

在调用完addWaiter方法后，会调用acquireQueued方法，该方法的参数就是之前加入队列的node节点，作用是决定当前的线程是阻塞还是获取锁。



方法也是一个死循环，首先获取当前节点的前置节点，如果说前置节点是head节点，并且tryAcquire调用成功，获取到了锁，那么就将当前节点设置为头结点，并返回。如果前置节点不是头结点，或者获取锁失败了，那么就将当前线程阻塞起来。下次被唤醒的时候就可以接着在acquireQueued方法里面执行了。多次调用shouldParkAfterFailedAcquire方法后，最终队列收敛后如下：



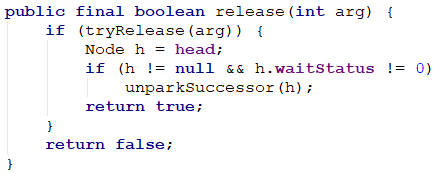
**除了头节点，剩余节点都会返回true，表示需要阻塞等待**。

**除了尾节点，剩余节点都满足waitStatus==SIGNAL，表示释放后需要唤醒后继节点**。

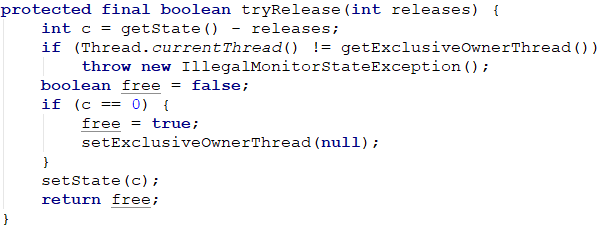
1. **Unlock**

ReentrantLock中Unlock和lock是对偶出现的，有多少个lock就有多少个unlock。

当调用unlock的时候会调用release方法：



首先调用tryRelease方法：



获取状态之后，然后将状态的值-1，如果状态值为0，那么就可以释放锁，将锁的线程设置为null，否则说明该线程多次重入锁，修改状态值，返回修改结果。

回到release方法，当tryRealse返回false的时候，release方法也返回false，说明这个时候是线程重入的情况；当tryRelease返回的是true的时候，说明当前线程释放锁成功，那么就获取头结点，在头结点不为null并且头结点的waitstatus不为0的时候，去唤醒头结点的后面一个节点。这边就和之前acquireQueued方法中阻塞的线程呼应起来，acquireQueued方法里阻塞的线程就会接着判断其前置节点是否等于head并且能够成功获取锁，如果都可以，那么就将头节点设置为当前的node节点。

那么不公平锁是怎么形成的呢？每次唤醒的都是阻塞队列头结点的下一个节点，明明是公平的嘛，先来先服务。这里需要看下最一开始的lock方法，假设当前线程A持有锁，线程B、C、D在阻塞队列中，A执行完了，释放锁，按理说要唤醒B，但是在A释放完锁之后，唤醒B之前，来了个线程E，调用了lock方法，发现state为0，那么E就修改state并持有锁了，A唤醒B之后，B尝试去获取锁，发现已经被占用了，这个时候B就只能接着阻塞了。非公平的意思就是这样。