深入浅出Java并发包—锁机制(一) - 人生设计师 - 博客园

深入浅出Java并发包—锁机制(一)

前面我们看到了Lock和synchronized都能正常的保证数据的一致性（上文例子中执行的结果都是20000000），也看到了Lock的优势，那究竟他们是什么原理来保障的呢？今天我们就来探讨下Java中的锁机制！

Synchronized是基于JVM来保证数据同步的，而Lock则是在硬件层面，依赖特殊的CPU指令实现数据同步的，那究竟是如何来实现的呢？我们一一看来！

一、synchronized的实现方案

synchronized比较简单，语义也比较明确，尽管Lock推出后性能有较大提升，但是基于其使用简单，语义清晰明了，使用还是比较广泛的，其应用层的含义是把任意一个非NULL的对象当作锁。当synchronized作用于方法时，锁住的是对象的实例(this)，当作用于静态方法时，锁住的是Class实例，又因为Class的相关数据存储在永久带，因此静态方法锁相当于类的一个全局锁，当synchronized作用于一个对象实例时，锁住的是对应的代码块。在Sun的HotSpot JVM实现中，其实synchronized锁还有一个名字：对象监视器。

当多个线程一起访问某个对象监视器的时候，对象监视器会将这些请求存储在不同的容器中。

1、 Contention List：竞争队列，所有请求锁的线程首先被放在这个竞争队列中

2、 Entry List：Contention List中那些有资格成为候选资源的线程被移动到Entry List中

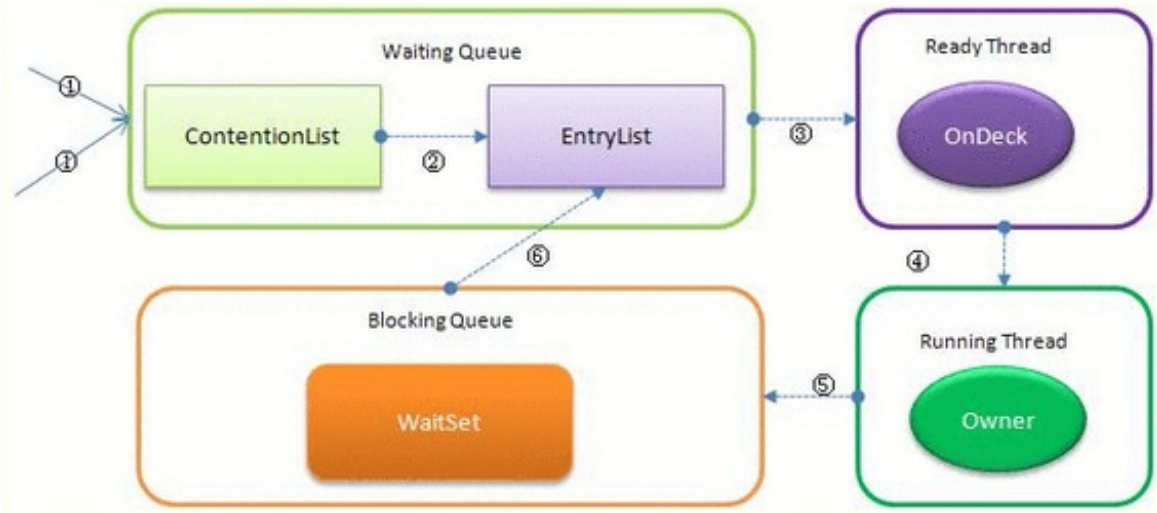
3、 Wait Set：哪些调用wait方法被阻塞的线程被放置在这里

4、 OnDeck：任意时刻，最多只有一个线程正在竞争锁资源，该线程被成为OnDeck

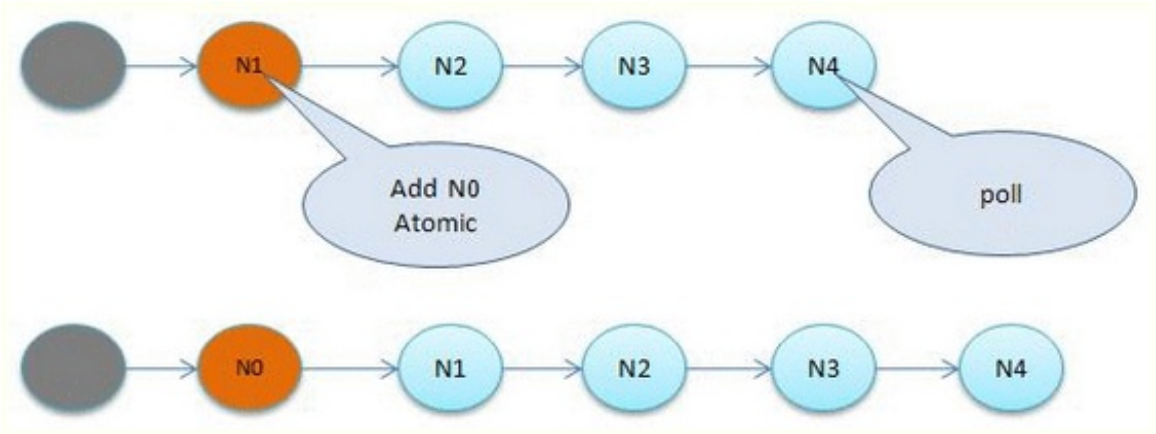
5、 Owner：当前已经获取到所资源的线程被称为Owner

6、 !Owner：当前释放锁的线程

下图展示了他们之前的关系



ContentionList并不是真正意义上的一个队列。仅仅是一个虚拟队列，它只有Node以及对应的Next指针构成，并没有Queue的数据结构。每次新加入Node会在队头进行，通过CAS改变第一个节点为新增节点，同时新增阶段的next指向后续节点，而取数据都在队列尾部进行。



JVM每次从队列的尾部取出一个数据用于锁竞争候选者（OnDeck），但是并发情况下，ContentionList会被大量的并发线程进行CAS访问，为了降低对尾部元素的竞争，JVM会将一部分线程移动到EntryList中作为候选竞争线程。Owner线程会在unlock时，将ContentionList中的部分线程迁移到EntryList中，并指定EntryList中的某个线程为OnDeck线程（一般是最先进去的那个线程）。Owner线程并不直接把锁传递给OnDeck线程，而是把锁竞争的权利交个OnDeck，OnDeck需要重新竞争锁。这样虽然牺牲了一些公平性，但是能极大的提升系统的吞吐量，在JVM中，也把这种选择行为称之为“竞争切换”。

OnDeck线程获取到锁资源后会变为Owner线程，而没有得到锁资源的仍然停留在EntryList中。如果Owner线程被wait方法阻塞，则转移到WaitSet队列中，直到某个时刻通过notify或者notifyAll唤醒，会重新进去EntryList中。

处于ContentionList、EntryList、WaitSet中的线程都处于阻塞状态，该阻塞是由操作系统来完成的（Linux内核下采用pthread_mutex_lock内核函数实现的）。该线程被阻塞后则进入内核调度状态，会导致系统在用户和内核之间进行来回切换，严重影响锁的性能。为了缓解上述性能问题，JVM引入了自旋锁。原理非常简单，如果Owner线程能在很短时间内释放锁资源，那么哪些等待竞争锁的线程可以稍微等一等（自旋）而不是立即阻塞，当Owner线程释放锁后可立即获取锁，进而避免用户线程和内核的切换。但是Owner可能执行的时间会超过设定的阈值，争用线程在一定时间内还是获取不到锁，这是争用线程会停止自旋进入阻塞状态。基本思路就是先自旋等待一段时间看能否成功获取，如果不成功再执行阻塞，尽可能的减少阻塞的可能性，这对于占用锁时间比较短的代码块来说性能能大幅度的提升！

但是有个头大的问题，何为自旋？其实就是执行几个空方法，稍微等一等，也许是一段时间的循环，也许是几行空的汇编指令，其目的是为了占着CPU的资源不释放，等到获取到锁立即进行处理。但是如何去选择自旋的执行时间呢？如果自旋执行时间太长，会有大量的线程处于自旋状态占用CPU资源，进而会影响整体系统的性能。因此自旋的周期选的额外重要！

JVM对于自旋周期的选择，基本认为一个线程上下文切换的时间是最佳的一个时间，同时JVM还针对当前CPU的负荷情况做了较多的优化

1、 如果平均负载小于CPUs则一直自旋

2、 如果有超过(CPUs/2)个线程正在自旋，则后来线程直接阻塞

3、 如果正在自旋的线程发现Owner发生了变化则延迟自旋时间（自旋计数）或进入阻塞

4、 如果CPU处于节电模式则停止自旋

5、 自旋时间的最坏情况是CPU的存储延迟（CPU A存储了一个数据，到CPU B得知这个数据直接的时间差）

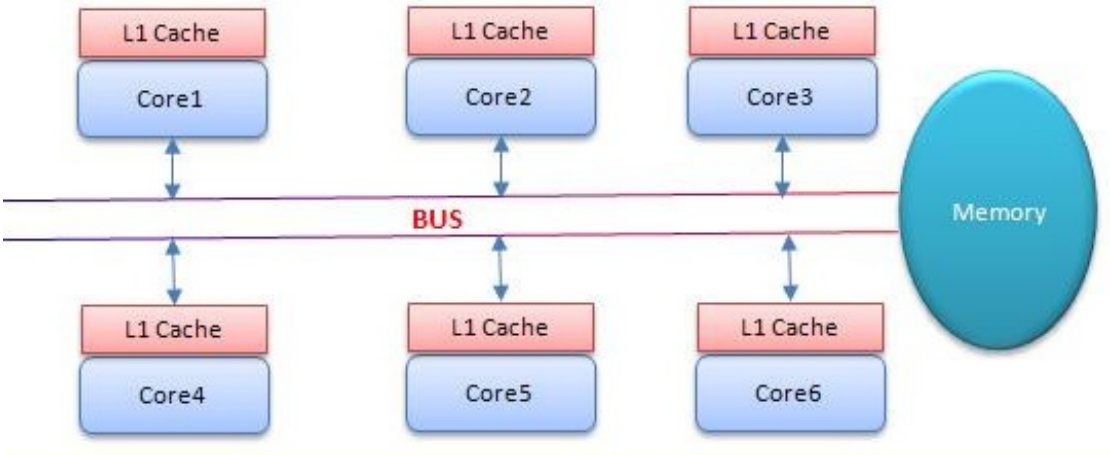
6、 自旋时会适当放弃线程优先级之间的差异

Synchronized在线程进入ContentionList时，等待的线程就通过自旋先获取锁，如果获取不到就进入ContentionList，这明显对于已经进入队列的线程是不公平的，还有一个不公平的事情就是自旋获取锁的线程还可能直接抢占OnDeck线程的锁资源。

在JVM6以后还引入了一种偏向锁，主要用于解决无竞争下面锁的性能问题。我们首先来看没有这个会有什么样子的问题。

现在基本上所有的锁都是可重入的，即已经获取锁的线程可以多次锁定/解锁监视对象，但是按照之前JVM的设计，每次加锁解锁都采用CAS操作，而CAS会引发本地延迟（下面会讲原因），因此偏向锁希望线程一旦获取到监视对象后，之后让监视对象偏向这个锁，进而避免多次CAS操作，说白了就是设置了一个变量，发现是这个线程过来的就避免再走加锁解锁流程。

那CAS为什么会引发本地延迟呢？这要从多核处（SMP）理架构说起（前面有提到过--JVM内存模型），下图基本上表明了多核处理的架构



多核CPU会共享一条系统总线，靠总线和主存通讯，但是每个CPU又有自己的一级缓存，而CAS是一条原子指令，其作用是让CPU比较，如果相同则进行数据更新，而这些是基于硬件实现的（JVM只是封装了硬件的汇编调用，AtomicInteger其实是通过调用这些封装后的接口实现的）。多核运算时，由于线程切换，很有可能第二次取值是在另外一核CPU上执行的。假设Core1和Core2把对应的某个值加载到自己的一级缓存时，某个时刻，core1更新了这个数据并通过总线通知主存，此时core2的一级缓存中的数据就失效了，他需要从主存中重新加载一次到一级缓存中，大家通过总线通讯被称之为一致性流量，总线的通讯能力有限，当缓存一致性流量过大时，总线会成为瓶颈，而当Core1和Core2的数据再次一致时，被称为缓存一致性！

而CAS要保证数据的一致性，恰好会引发比较多的一致性流量，如果有很多线程共享一个对象，当某个线程成功执行一次CAS时会引发总线风暴，这就是本地延迟，而偏向锁就是为了消除CAS，降低Cache一致性流量！

当然并不是所有的CAS都会引发总线风暴，这和Cache一致性协议有关系的。但是偏向锁的引入却带来了另外一个问题，在很多线程竞争使用中，如果一个线程持有偏向锁，另外一个线程想争用偏向对象，拥有者想释放这个偏向锁，释放会带来额外的性能开销，但是总体来说偏向锁带来的好处还是大于CAS的代价的。

二、Lock的实现

与synchronized不同的是，Lock书纯Java实现的，与底层的JVM无关。
在java.util.concurrent.locks包中有很多Lock的实现类，常用的有ReentrantLock、ReadWriteLock（实现类ReentrantReadWriteLock），其实现都依赖java.util.concurrent.AbstractQueuedSynchronizer类（简称AQS），实现思路都大同小异，因此我们以ReentrantLock作为讲解切入点。

分析之前我们先来花点时间看下AQS。AQS是我们后面将要提到的CountDownLatch/FutureTask/ReentrantLock/RenntrantReadWriteLock/Semaphore的基础，因此AQS也是Lock和Excutor实现的基础。它的基本思想就是一个同步器，支持获取锁和释放锁两个操作。

获取锁：首先判断当前状态是否允许获取锁，如果是就获取锁，否则就阻塞操作或者获取失败，也就是说如果是独占锁就可能阻塞，如果是共享锁就可能失败。另外如果是阻塞线程，那么线程就需要进入阻塞队列。当状态位允许获取锁时就修改状态，并且如果进了队列就从队列中移除。

```
while(synchronization state does not allow acquire){

   enqueue current thread if not already queued;

   possibly block current thread;

}

dequeue current thread if it was queued;
```

释放锁:这个过程就是修改状态位，如果有线程因为状态位阻塞的话，就唤醒队列中的一个或者更多线程。

```
update synchronization state;

if(state may permit a blocked thread to acquire)

   unlock one or more queued threads;
```

要支持上面两个操作就必须有下面的条件：

1、 状态位必须是原子操作的

2、 阻塞和唤醒线程

3、 一个有序的队列，用于支持锁的公平性

怎么样才能满足这几个条件呢？

1、 原子操作状态位，前面我们已经提到了，实际JDK中也是通过一个32bit的整数位进行CAS操作

来实现的。

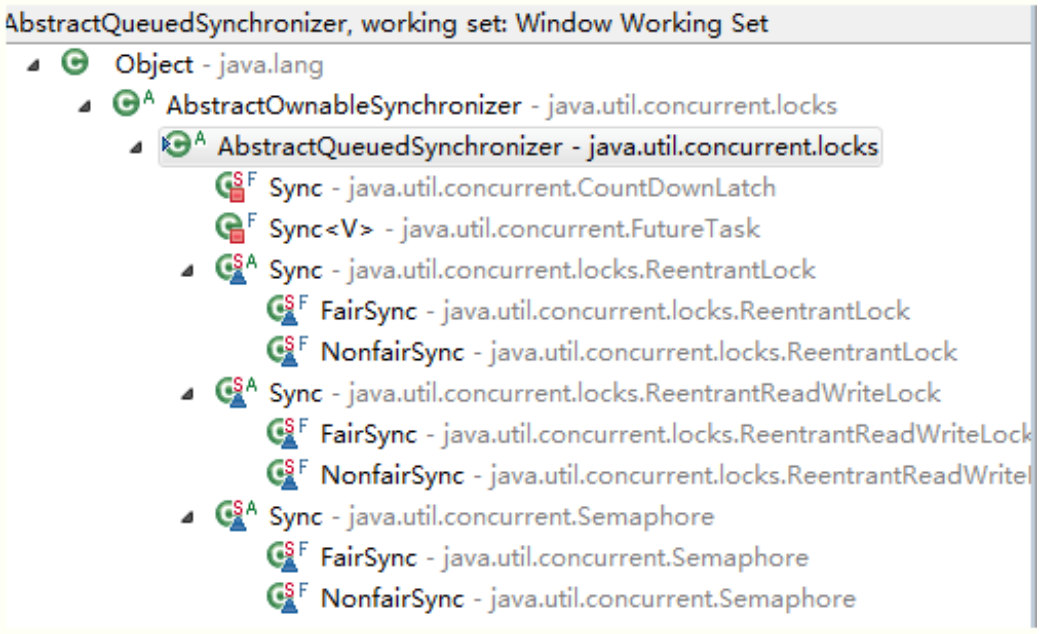
2、 阻塞和唤醒，JDK1.5之前的API中并没有阻塞一个线程，然后在将来的某个时刻唤醒它（wait/notify是基于synchronized下才生效的，在这里不算），JDK5之后利用JNI在LockSupport 这个类中实现了相关的特性！

方法摘要	
static Object	getBlocker(Thread t) 返回提供给最近一次尚未解除阻塞的 park 方法调用的 blocker 对象，如果该调用不受阻塞，则返回 null。
static void	park() 为了线程调度，禁用当前线程，除非许可可用。
static void	park(Object blocker) 为了线程调度，在许可可用之前禁用当前线程。
static void	parkNanos(long nanos) 为了线程调度禁用当前线程，最多等待指定的等待时间，除非许可可用。
static void	parkNanos(Object blocker, long nanos) 为了线程调度，在许可可用前禁用当前线程，并最多等待指定的等待时间。
static void	parkUntil(long deadline) 为了线程调度，在指定的时限前禁用当前线程，除非许可可用。
static void	parkUntil(Object blocker, long deadline) 为了线程调度，在指定的时限前禁用当前线程，除非许可可用。
static void	unpark(Thread thread) 如果给定线程的许可尚不可用，则使其可用。

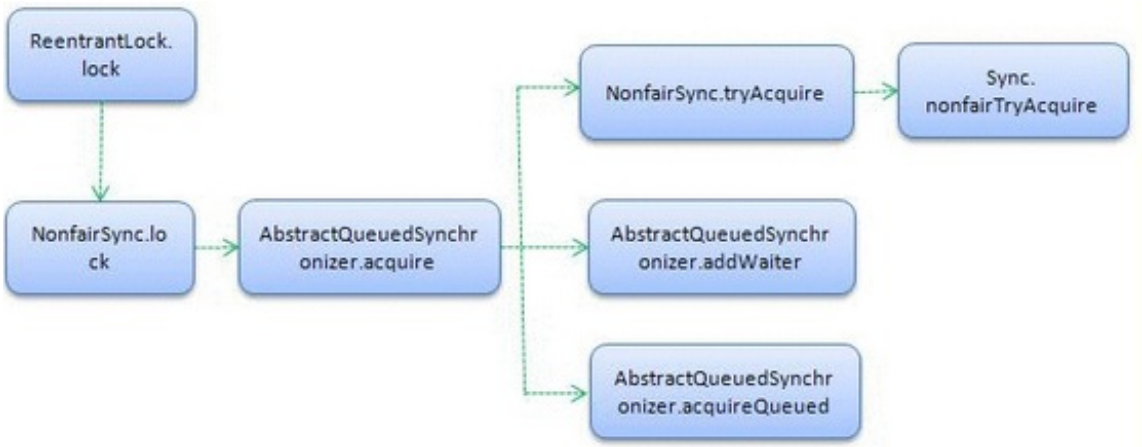
3、 有序队列：在AQS中采用ＣＬＨ队列来解决队列的有序问题。

我们来看下ReentrantLock的调用过程

经过源码分析，我们看到ReentrantLock把所有的Lock都委托给Sync类进行处理，该类继承自AQS，其类关系图如下

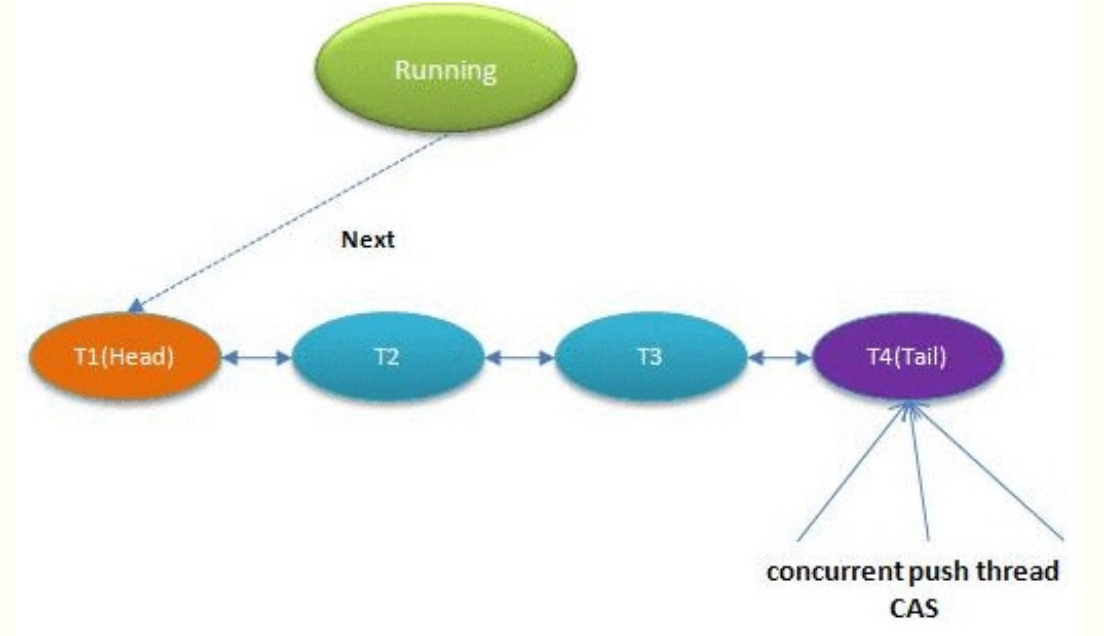


其中Sync又有两个final static的子类NonfairSync和FairSync用于支持非公平锁和公平锁。我们先来挑一个看下对应Reentrant.lock()的调用过程（默认为非公平锁）



这些模版很难让我们直观的看到整个调用过程，但是通过上面的过程图和AbstractQueuedSynchronizer的注释可以看出，AbstractQueuedSynchronizer抽象了大多数Lock的功能，而只把tryAcquire(int)委托给子类进行多态实现。tryAcquire用于判断对应线程事都能够获取锁，无论成功与否，AbstractQueuedSynchronizer都将处理后面的流程。

简单来讲，AQS会把所有请求锁的线程组成一个CLH的队列，当一个线程执行完毕释放锁(Lock.unlock())的时候，AQS会激活其后继节点，正在执行的线程不在队列当中，而那些等待的线程全部处于阻塞状态，经过源码分析，我们可以清楚的看到最终是通过LockSupport.park()实现的，而底层是调用sun.misc.Unsafe.park()本地方法，再进一步，HotSpot在Linux中中通过调用pthread_mutex_lock函数把线程交给系统内核进行阻塞。其运行示意图如下



与synchronized相同的是，这个也是一个虚拟队列，并不存在真正的队列示例，仅存在节点之前的前后关系。（注：原生的CLH队列用于自旋锁，JUC将其改造为阻塞锁）。和synchronized还有一点相同的是，就是当获取锁失败的时候，不是立即进行阻塞，而是先自旋一段时间看是否能获取锁，这对那些已经在阻塞队列里面的线程显然不公平（非公平锁的实现，公平锁通过有序队列强制线程顺序进行），但会极大的提升吞吐量。如果自旋还是获取失败了，则创建一个节点加入队列尾部，加入方法仍采用CAS操作，并发对队尾CAS操作有可能会发生失败，AQS是采用自旋循环的方法，知道CAS成功！下面我们来看下锁的实现细节！

锁的实现依赖与lock()方法，Lock()方法首先是调用acquire(int)方法，不管是公平锁还是非公平锁

```
public final void acquire(int arg) {
        if (!tryAcquire(arg) &&
            acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
            selfInterrupt();
    }
```

Acquire()方法默认首先调用tryAcquire(int)方法，而此时公平锁和不公平锁的实现就不一样了。

1、Sync.NonfairSync.TryAcquire（非公平锁）

nonfairTryAcquire方法是lock方法间接调用的第一个方法，每次调用都会首先调用这个方法，我们来看下对应的实现代码：

```
final boolean nonfairTryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        int c = getState();
        if (c == 0) {
            if (compareAndSetState(0, acquires)) {
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
```

该方法首先会判断当前线程的状态，如果c==0 说明没有线程正在竞争锁。（反过来，如果c!=0则说明已经有其他线程已经拥有了锁）。如果c==0，则通过CAS将状态设置为acquires(独占锁的acquires为1)，后续每次重入该锁都会+1，每次unlock都会-1，当数据为0时则释放锁资源。其中精妙的部分在于：并发访问时，有可能多个线程同时检测到c为0，此时执行compareAndSetState(0, acquires))设置，可以预见，如果当前线程CAS成功，则其他线程都不会再成功，也就默认当前线程获取了锁，直接作为running线程，很显然这个线程并没有进入等待队列。如果c!=0，首先判断获取锁的线程是不是当前线程，如果是当前线程，则表明为锁重入，继续+1，修改state的状态，此时并没有锁竞争，也非CAS，因此这段代码也非常漂亮的实现了偏向锁。