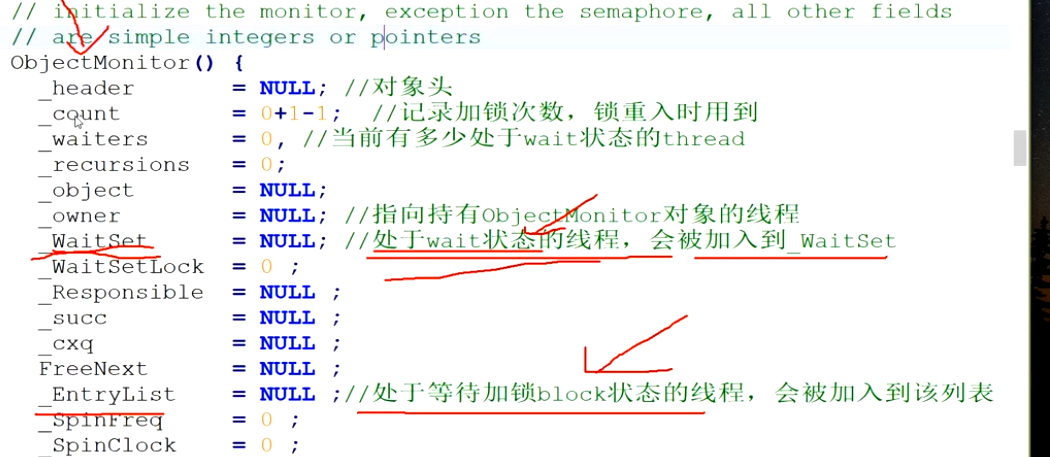
# synchronized



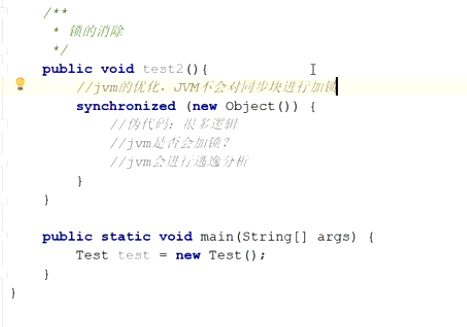
在两个方法实现加锁释放锁,不用lock,只是用隐式锁

## 对象头markword

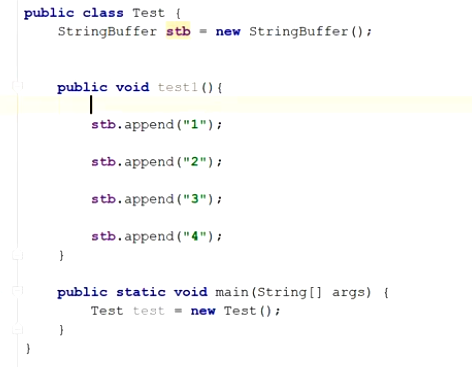


老版本Synchronized,在线程没有获取到moniter对象,竞争失败时会处于wait(阻塞状态),会被先放到WaitSet队列里面

## 锁的消除

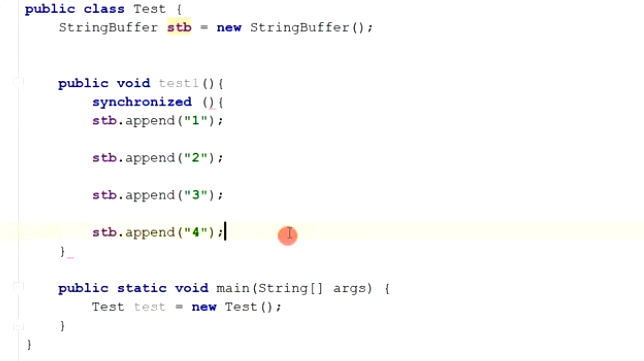


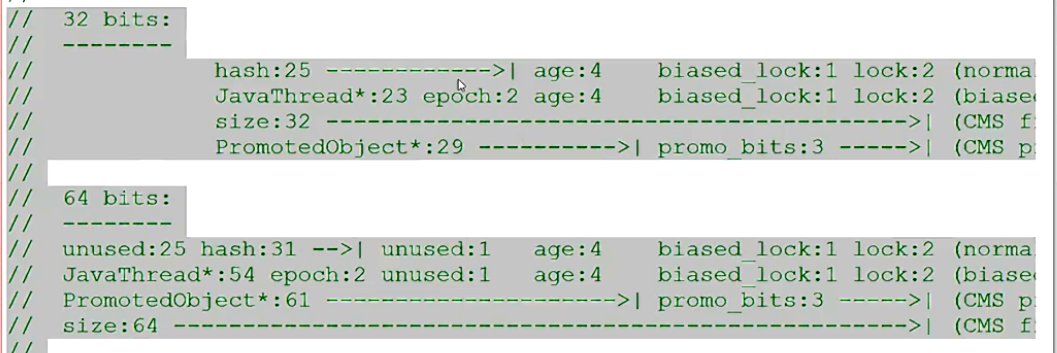
别的多个线程去调用test2这个方法,加锁的都是new object(),每个线程自己的new object()对象,别的线程访问不到,加锁就没有意义,不会有同步问题,jvm会进行逃逸分析,进行优化,不会加锁,这个叫锁的消除.



## 锁的粗化

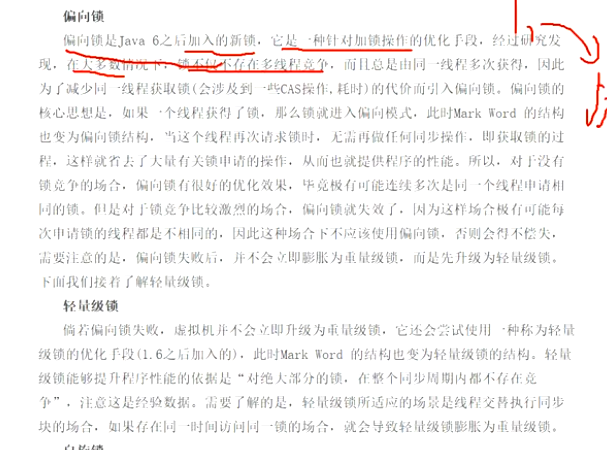
Stringbuffer线程安全的,底层是用synchronized实现的,像上面去调用,jvm会进行优化,只加锁一次把四次字符串更改都加进去,相当于下面这样





32位虚拟机和64位虚拟机的对象头

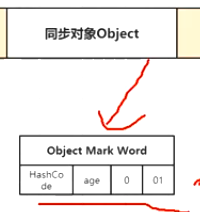
## 什么是偏向锁



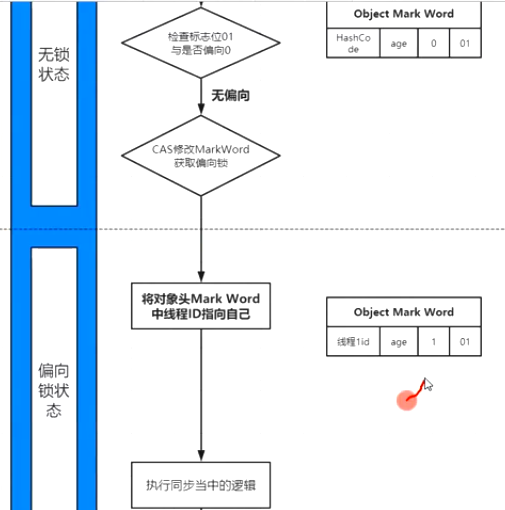


## 锁膨胀升级过程

刚开始,线程1访问同步代码块,检查同步对象object的Markword,中标志位,检测出为无所状态(倒数两位为01,为无锁或偏向锁,倒数第三位为0,为无锁状态)

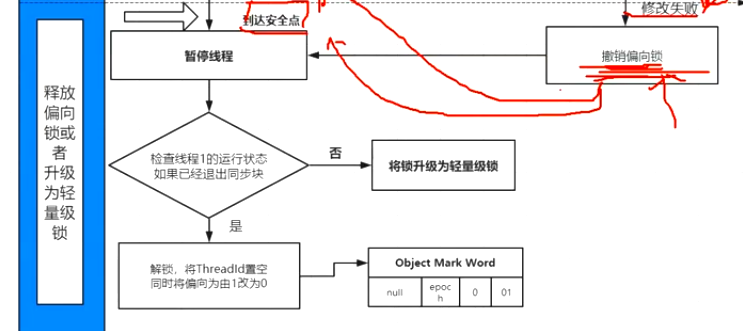


CAS算法修markword,获取偏向锁(把对象头markword中的线程id指向自己,把倒数第三bit位修改为1) 偏向锁不会自动释放

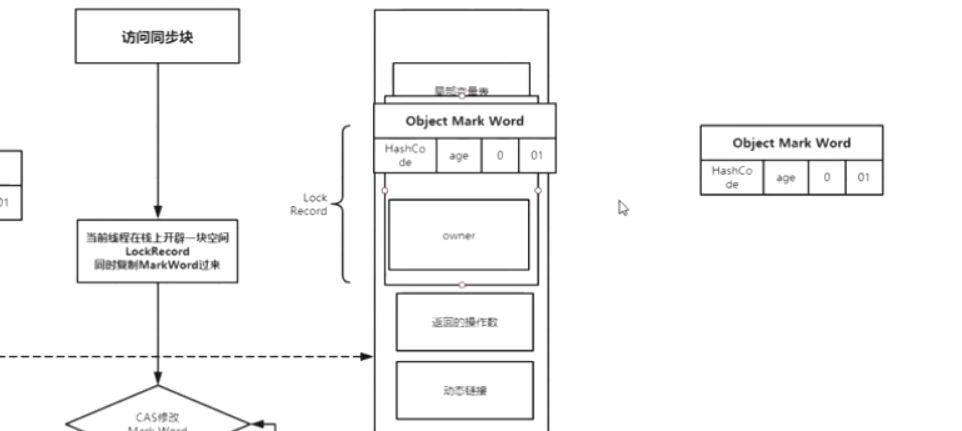


执行同步当中的逻辑

线程2进来,然后产生并发访问,线程2会先检查markword的后三位判断锁状态,当发现倒数第三位为1时,有偏向锁,会先检查,markword中线程id是否为自己,如果不是自己 , CAS也会准备修改markword中线程id为自己 , (正常来说会修改失败,只有当线程1 恰巧执行完才可以修改) 会修改失败,失败后会向jvm提交撤销偏向锁,然后jvm会使线程一达到安全点后暂停线程一 , 暂停之后查看线程一的运行状态,如果线程1 没有运行完(是否已经退出同步块),如果没有退出,就会将锁升级为轻量级锁,如果线程一退出了同步块,则进行解锁,把markword的线程id修改为null,把倒数第三位修改为0.

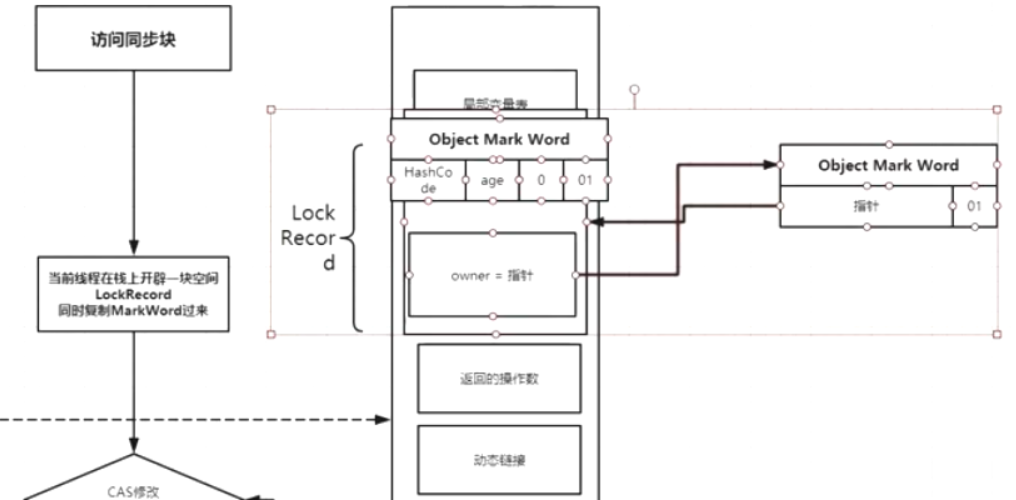


存在竞争后,每个线程都有自己的线程栈,在线程栈上面开辟一个空间LockRecord,(线程栈里面有局部变量表,方法返回值,动态链接),会把锁对象的对象copy一份到LockRecord ,LockRecord里面还有指针变量owner.

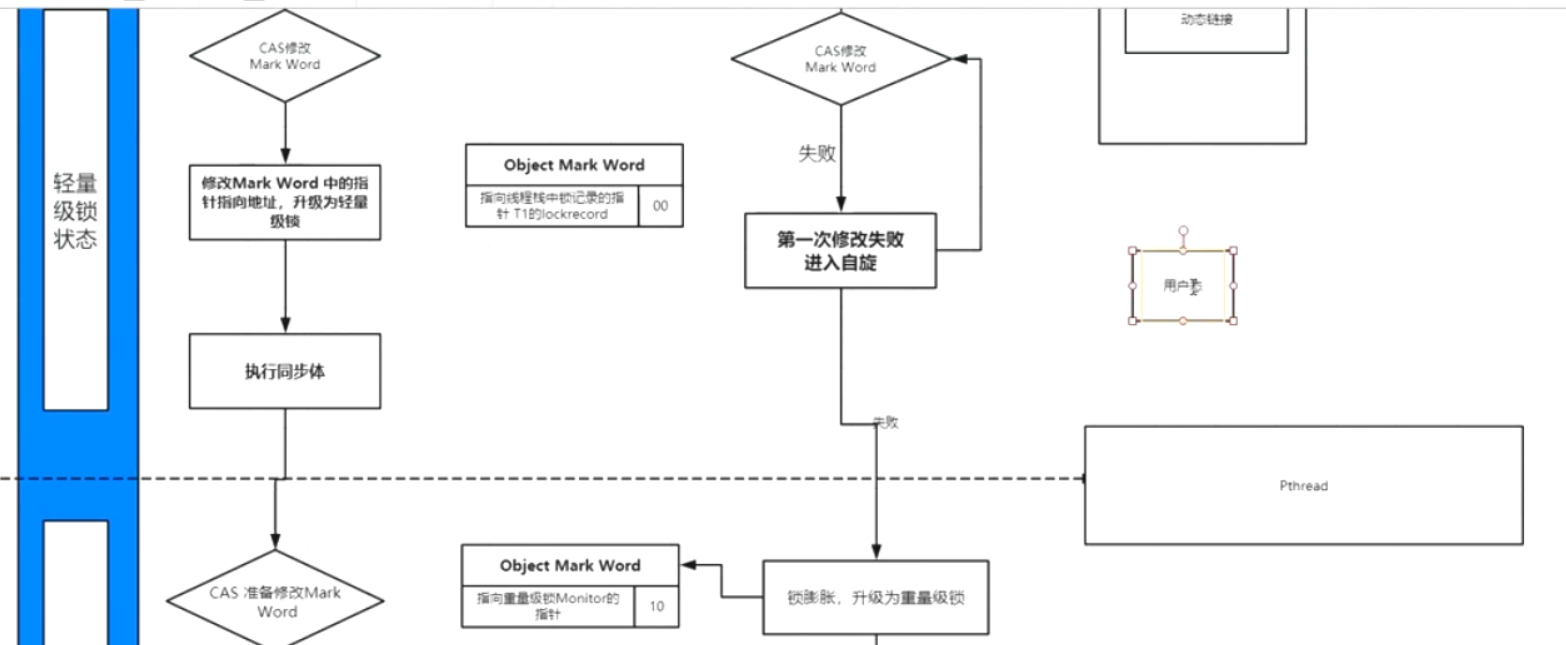


复制完对象头之后,把锁对象的对象头前三十位变成指针指向当前线程(这时是刚开始抢,两个线程都进行这个操作)的线程栈里面LockRecord空间, LockRecord空间里面owner指针指向锁对象的对象头.

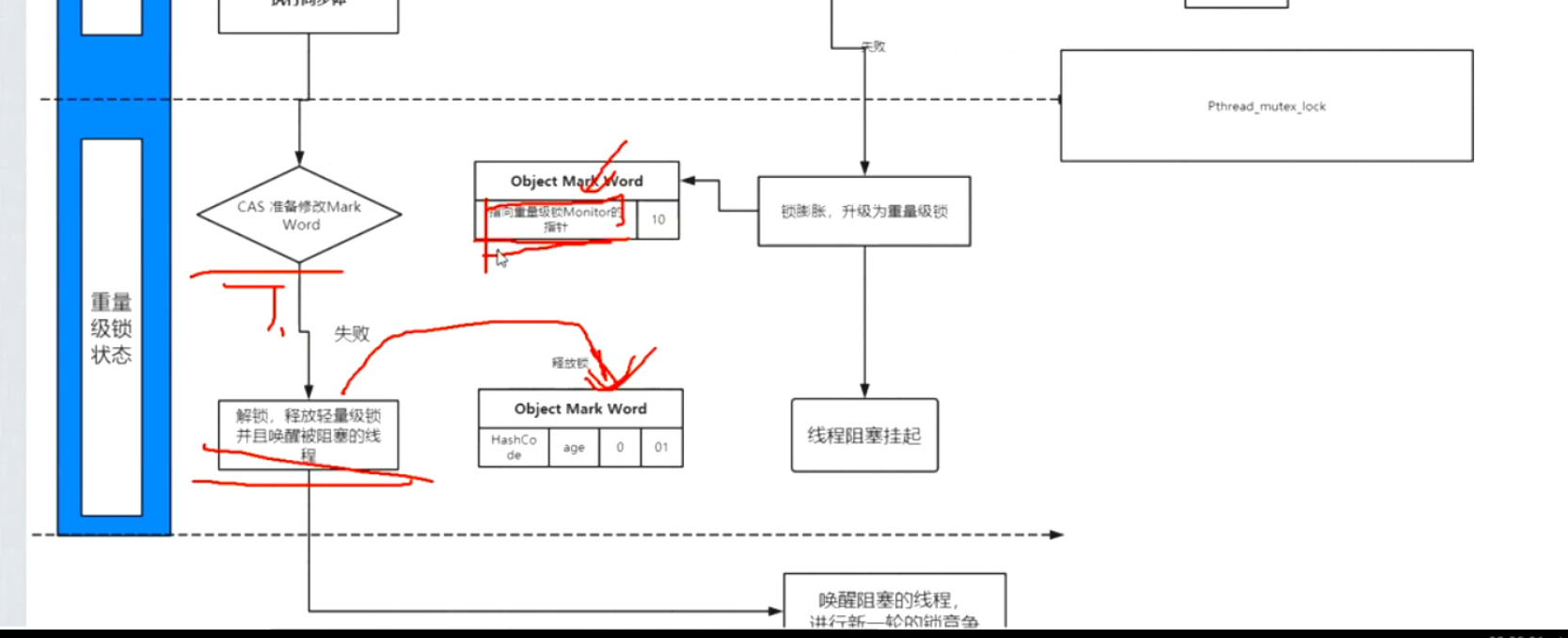
轻量级锁记录的方式如下

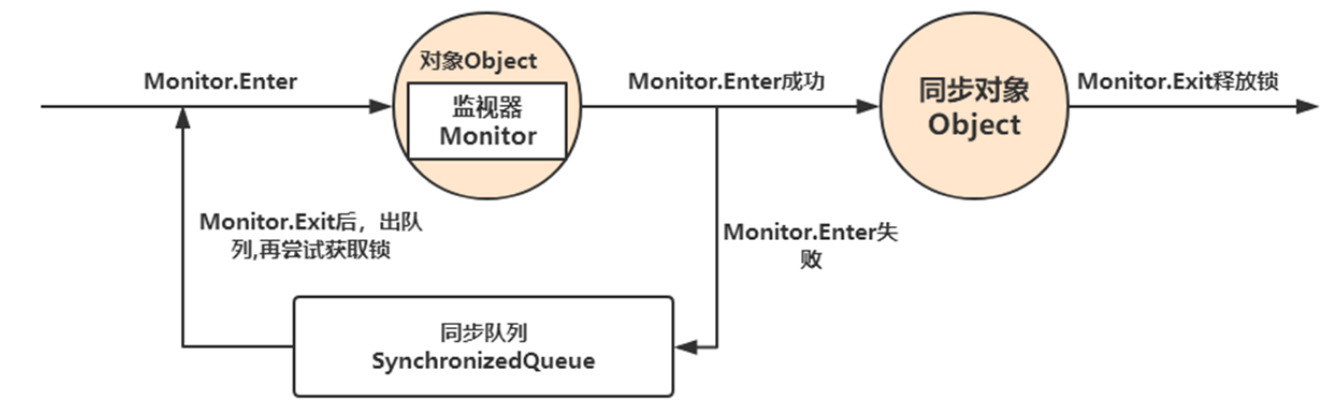


如果T1线程抢夺成功,那么T1就持有轻量级锁,就可以执行同步块内容,T2线程抢夺失败会进行自旋,这是T1如果执行完成,那么T1就会释放锁,然会T2在自旋成功的情况下,就会再次进行修改锁对象的对象头前三十位指针指向自己线程栈的LockRecord空间,自己的owner指针指向锁对象的对象头,如果T2自旋次数超过规定次数或者自旋失败,那个T2线程就会向jvm请求锁升级,升级为重量级锁



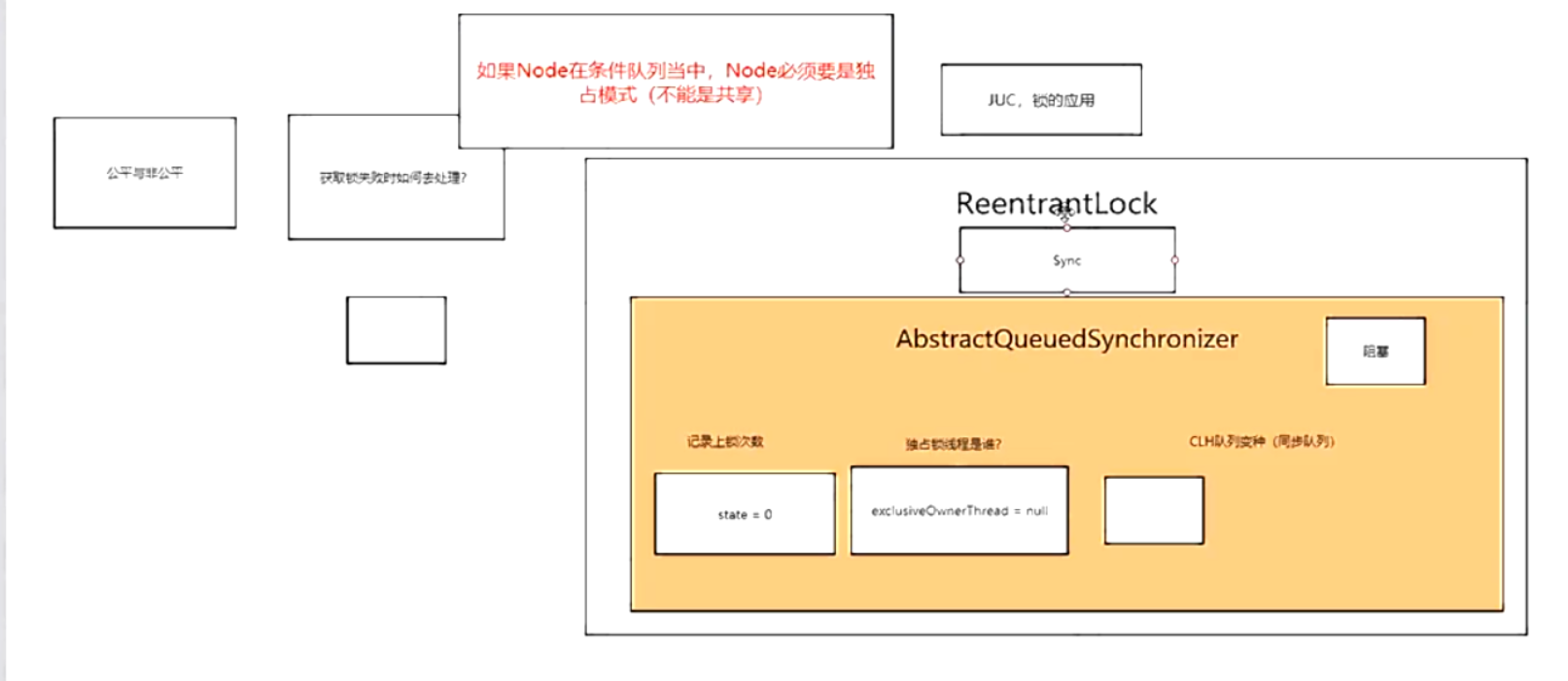
升级重量锁时,虚拟机会调用操作系统底层的线程库,各个系统都不一样,linux是pThread,会去申请互斥量,这时涉及到了从用户态到内核态的转变,这个转变过程特别消耗资源,然后就会把锁对象的前32位转换为互斥锁,然后会效用Pthread\_mutex\_lock锁使自己T2线程阻塞挂起,这时T1如果刚执行完同步块在解锁时修改锁对象的对象头前32位修改失败时,会先进行解锁,然后唤醒阻塞的线程,进行新一轮的锁竞争.





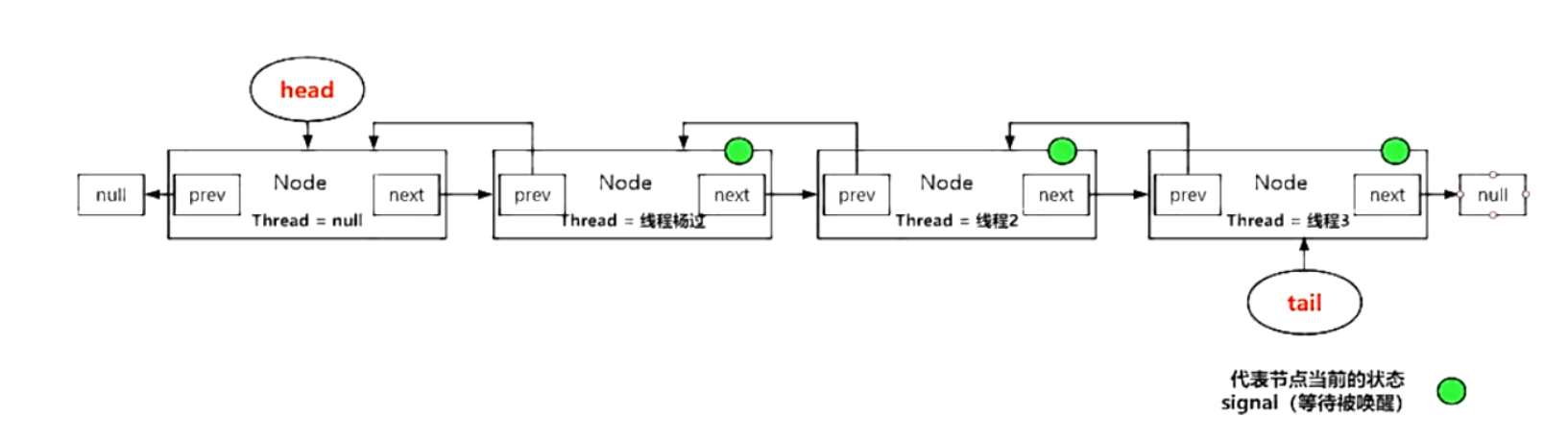
# AQS

AQS定义了一套多线程访问共享变量的同步器框架,是一个依赖状态(state)的同步器

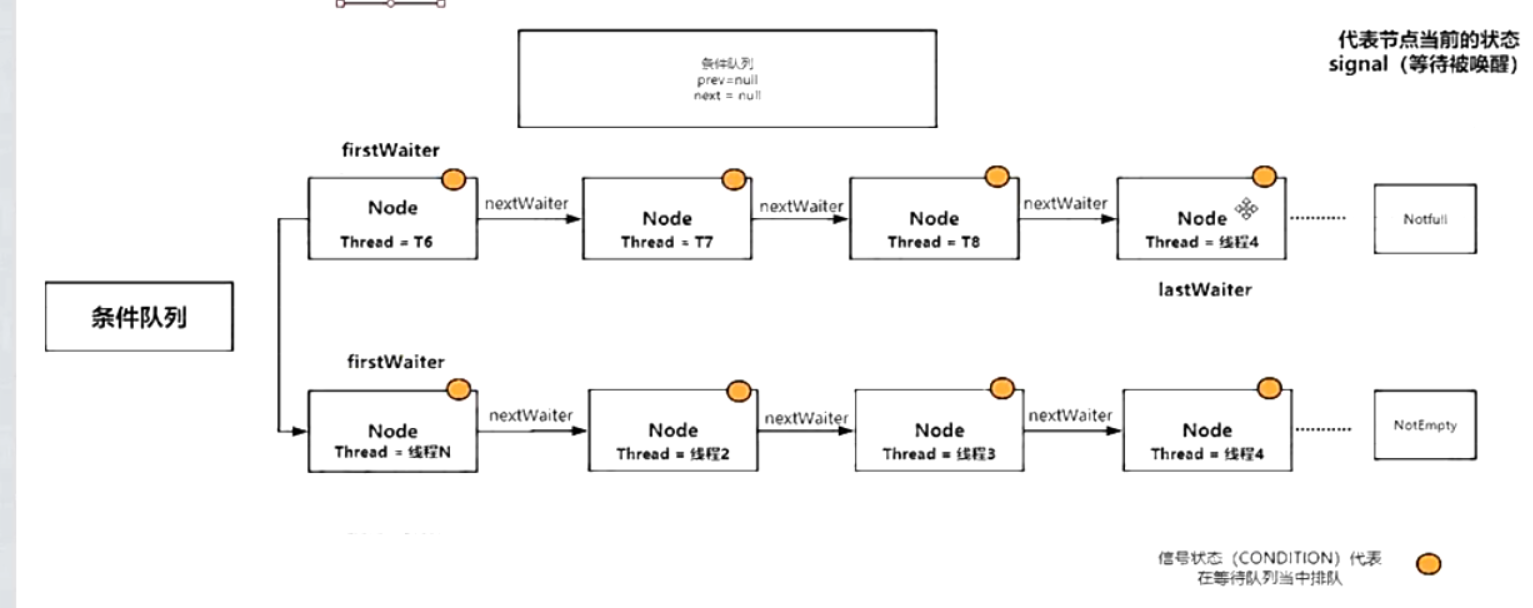


## Node

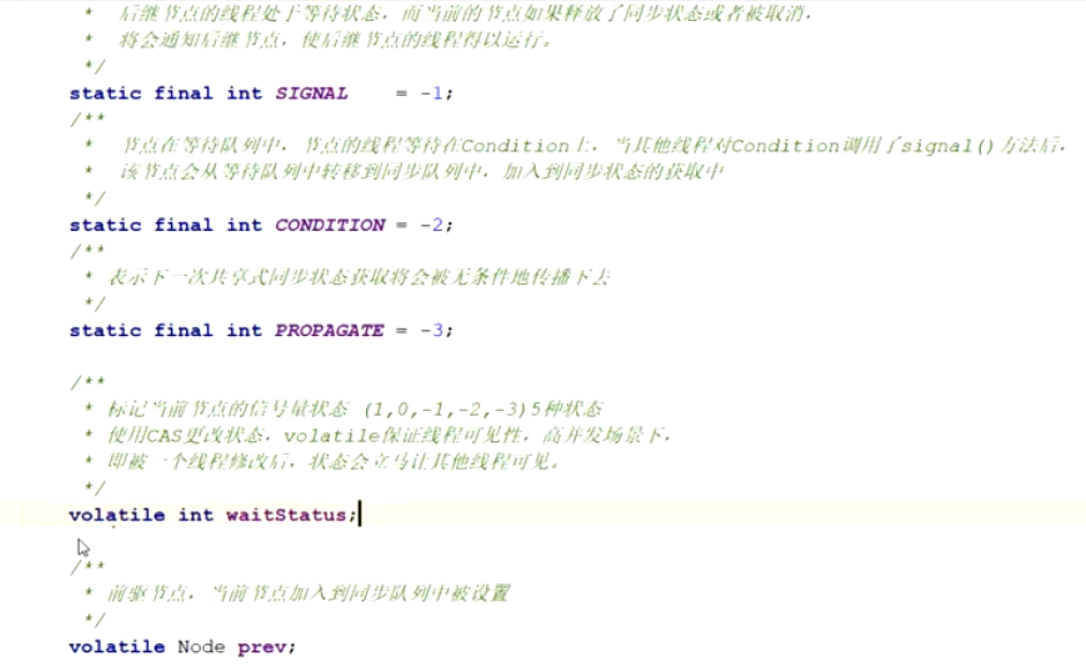
Node结构,node存储里面是可以唤醒的,处于等待状态



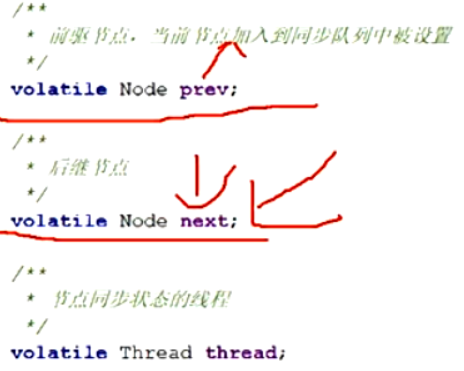
条件队列----条件队列基于node,条件队列存储的都是阻塞的线程



信号量状态,,状态会记录在waitStatus里面



前驱指针和后驱指针



# Atomic和Unsafe类