AQS系列一ReentrantLock的源码--aqs加锁过程

Java的内置锁一直都是备受争议的，在JDK 1.6之前，synchronized这个重量级锁其性能一直都是较为低下，虽然在1.6后，进行大量的锁优化策略,但是与Lock相比synchronized还是存在一些缺陷的：虽然synchronized提供了便捷性的隐式获取锁释放锁机制（基于JVM机制），但是它却缺少了获取锁与释放锁的可操作性，可中断、超时获取锁，且它为独占式在高并发场景下性能大打折扣。 synchronized不可中断、不能超时获取锁、只能是非公平锁

多线程同步内部如何实现的

wait/notify, synchronized, ReentrantLock。。。。。

模拟一些同步的思路

自旋实现同步

```
volatile int status=0;//标识---是否有线程在同步块-----是否有线程上锁成功
void lock(){                        cas+while自旋

   while(!compareAndSet(0,1)){
   }
   //lock
    //10  t1

}

void unlock(){
   status=0;
}

boolean compareAndSet(int except,int newValue){
   //cas操作,修改status成功则返回true
}
```

缺点：耗费cpu资源。没有竞争到锁的线程会一直占用cpu资源进行cas操作，假如一个线程获得锁后要花费Ns处理业务逻辑，那另外一个线程就会白白的花费Ns的cpu资源

思路：让得不到锁的线程让出CPU

yield+自旋

```
volatile int status=0;
void lock(){
   while(!compareAndSet(0,1)){
     yield();//自己实现
   }
   //lock logic

}
void unlock(){
   status=0;
}
```

要解决自旋锁的性能问题必须让竞争锁失败的线程不空转,而是在获取不到锁的时候能把cpu资源给让出来，yield()方法就能让出cpu资源，当线程竞争锁失败时，会调用yield方法让出cpu。自旋+yield的方式并没有完全解决问题，当系统只有两个线程竞争锁时，yield是有效的。需要注意的是该方法只是当前让出cpu，有可能操作系统下次还是选择运行该线程，比如里面有2000个线程，想想会有什么问题?

sleep+自旋

```
volatile int status=0;
void lock(){
   while(!compareAndSet(0,1)){
          sleep(2);
   }
   //lock---10m

}
void unlock(){
   status=0;
}
```

sleep的时间为什么是10？怎么控制呢？就是你是调用者其实很多时候你也不知道这个时间是多少?

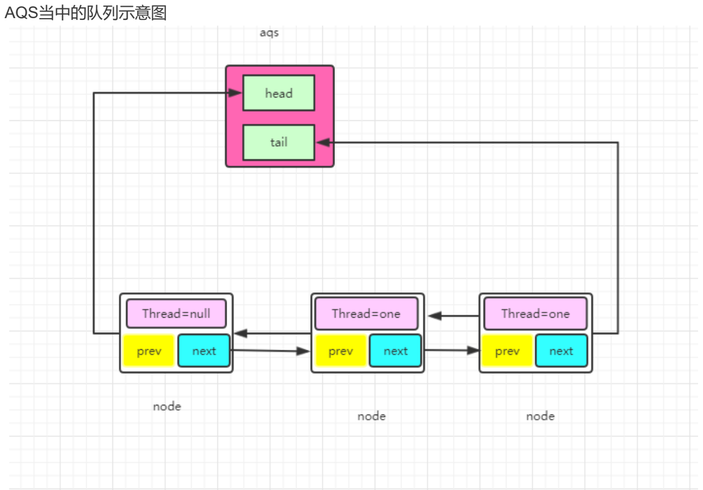
park+自旋

```
volatile int status=0;
Queue parkQueue;//集合 数组  list

void lock(){
   while(!compareAndSet(0,1)){
           //
           park();----
   }
   //lock    10分钟
   。。。。。。
   unlock()
}

void unlock(){
   lock_notify();
}

void park(){
   //将当期线程加入到等待队列
   parkQueue.add(currentThread);
   //将当期线程释放cpu  阻塞   睡眠
   releaseCpu();
}
void lock_notify(){
    //status=0
   //得到要唤醒的线程头部线程
   Thread t=parkQueue.header();
   //唤醒等待线程
   unpark(t);
}
```

AQS（AbstractQueuedSynchronizer）类的设计主要代码（具体参考源码）

```
private transient volatile Node head; //队首
private transient volatile Node tail;//尾
private volatile int state;//锁状态，加锁成功则为1，重入+1 解锁则为0
```

AQS当中的队列示意图



Node类的设计

```
public class Node{
    volatile Node prev;
    volatile Node next;
```

```
    volatile Thread thread;
    int ws;
}
```

上锁过程重点

```
final ReentrantLock lock = new ReentrantLock(true);
 Thread t1= new Thread("t1"){
     @Override
     public void run() {
         lock.lock();
         logic();
         lock.unlock();
     }
 };
t1.start();
```

锁对象：其实就是*ReentrantLock*的实例对象，上述代码第一行中的*lock*对象就是所谓的锁

自由状态：自由状态表示锁对象没有被别的线程持有，计数器为*0*

计数器：再*lock*对象中有一个字段*state*用来记录上锁次数，比如*lock*对象是自由状态则*state*为*0*，如果大于零则表示被线程持有了，当然也有重入那么*state*则*>1*

waitStatus：仅仅是一个状态而已；*ws*是一个过渡状态，在不同方法里面判断*ws*的状态做不同的处理，所以*ws=0*有其存在的必要性

tail：队列的队尾

head：队列的对首

ts：第二个给*lock*加锁的线程

tf：第一个给*lock*加锁的线程

tc：当前给线程加锁的线程

tl：最后一个加锁的线程

tn：随便某个线程　　当然这些线程有可能重复，比如第一次加锁的时候*tf==tc==tl==tn*

节点：就是上面的*Node*类的对象，里面封装了线程，所以某种意义上*node*就等于一个线程

lock方法的逻辑

```
final void lock() {
    acquire(1);//1------标识加锁成功之后改变的值
}
```

acquire方法的逻辑

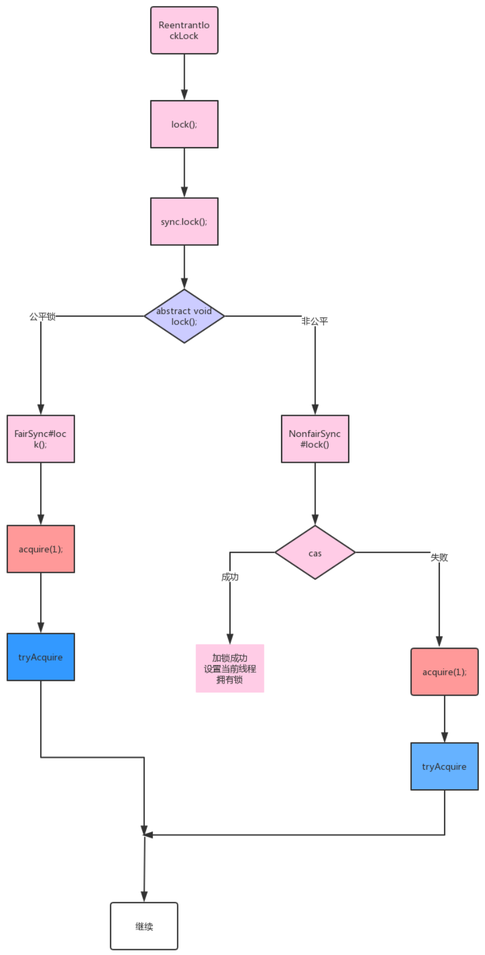
```
public final void acquire(int arg) {
    //tryAcquire(arg)尝试加锁，如果加锁失败则会调用acquireQueued方法加入队列去排队，如果加锁成功则不会调用
    //acquireQueued方法下文会有解释
    //加入队列之后线程会立马park，等到解锁之后会被unpark，醒来之后判断自己是否被打断了，如果被打断了则执行selfInterrupt方法
    //为什么需要执行这个方法？下文解释
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

公平锁首先会调用tryAcquire去尝试加锁，当然这里的尝试加锁并不是直接加锁，事实上tryAcquire当中其实

第一步便是判断锁是不是自由状态，如果是则判断直接是否需要排队（hasQueuedPredecessors方法判断队列是否被初始化（如果没有初始化显然不需要排队），和是否需要排队（队列如果被初始化了，则自己有可能需要排队）**）；如果**hasQueuedPredecessors**返回false，由于取反了故而不需要排队则进行Cas操作去上锁，如果需要排队则不会进入if分支当中，也不会进else if，会直接返回false表示加锁失败（为什么不进else if呢？这是java基础自己想想）**

第二步如果不是自由状态再判断是不是重入，如果不是重入则直接返回false加锁失败，如果是重入则把计数器+1

当然我们这里说的都是公平锁的，那么非公平锁和公平锁的区别再哪里呢？下图说明了公平和非公平的区别，记住一朝排队，永远排队



以上便是对tryAcquire**方法的总结**

公平锁的加锁过程的代码 tryAcquire方法源码分析

```
tryAcquire(arg)



protected final boolean tryAcquire(int acquires) {//1
    //获取当前线程
    final Thread current = Thread.currentThread();
    //获取lock对象的上锁状态，如果锁是自由状态则=0，如果被上锁则为1，大于1表示重入
    int c = getState();
    if (c == 0) {//没人占用锁--->我要去上锁----1、锁是自由状态
        //hasQueuedPredecessors，判断自己是否需要排队这个方法比较复杂，
        //下面我会单独介绍，如果不需要排队则进行cas尝试枷锁，如果加锁成功则把当前线程设置为拥有锁的线程
        //继而返回true
        if (!hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            //设置当前线程为拥有锁的线程，方面后面判断是不是重入（只需把这个线程拿出来判断是否当前线程即可判断重入）
            setExclusiveOwnerThread(current);
            return true;
```

```
        }
    }
    //如果C不等于0，而且当前线程不等于拥有锁的线程则不会进else if 直接返回false，加锁失败
    //如果C不等于0，但是当前线程等于拥有锁的线程则表示这是一次重入，那么直接把状态+1表示重入次数+1
    //那么这里也侧面说明了reentrantlock是可以重入的，因为如果是重入也返回true，也能lock成功
    else if (current == getExclusiveOwnerThread()) {
        int nextc = c + acquires;
        if (nextc < 0)
            throw new Error("Maximum lock count exceeded");
        setState(nextc);
        return true;
    }
    return false;
}
```

重点：

hasQueuedPredecessors判断是否需要排队的源码分析，这里需要记住一点，整个方法如果最后返回false，则去加锁，如果返回true则不加锁，因为这个方法被取反了

```
public final boolean hasQueuedPredecessors() {
    Node t = tail;
    Node h = head;
    Node s;
    /**
     * 下面提到的所有不需要排队，并不是字面意义的不需要排队，我实在想不出什么词语来描述这个“不需要排队”；不需要排队有两种情况
     * 一：队列没有初始化，不需要排队，不需要排队，不需要排队；直接去加锁，但是可能会失败；为什么会失败呢？
     * 假设两个线程同时来lock，都看到队列没有初始化，都认为不需要排队，都去进行CAS修改计数器；但是某个线程t1先拿到锁，那么另外一个t2则会CAS失败，这个时候他就会初始化队列，并排队
     *
    * 二：队列被初始化了，但是tc过来加锁，发觉队列当中第一个排队的就是自己（比如重入；那么什么是第一个排队的呢？下面解释了，往下看）这个时候他也-不需要排队，不需要排队，不需要排队；为什么不需要排对？
    * 因为队列当中第一个排队的线程他回去尝试获取一下锁，因为有可能这个时候持有锁锁的那个线程可能释放了锁，如果释放了就直接获取锁执行。但是如果没有释放他就会去排队，所以这里的不需要排队，不是真的不需要排队，好好理解一下
     *
     * h != t 判断首不等于尾这里要分三种情况
     * 1、队列没有初始化，也就是第一个线程tf来加锁的时候那么这个时候队列没有初始化，h和t都是null，那么这个时候不等于不成立（false）那么由于是&&运算后面的就不会走了，
     * 直接返回false表示不需要排队，而前面又是取反（if (!hasQueuedPredecessors())），所以会直接去cas加锁。
     * ----------第一种情况总结：队列没有初始化没人排队，那么我直接不排队，直接上锁；合情合理、有理有据令人信服；好比你去买票，服务员都闲的蛋疼，没人排队，你直接过去价钱拿票
     *
    * 2、队列被初始化了，后面我们会分析队列初始化的流程，如果队列被初始化那么h!=t则成立；h != t 返回true；但是是&&运算，故而还需要进行后续的判断
    * （有人可能会疑问，比如队列里面只有一个数据，那么头和尾都是同一个怎么会成立呢？其实这是第三种情况--队列里面只有一个数据；这里先不考虑，假设现在队列里面有大于1个数据）
    * 大于1个数据则成立;继续判断把h.next赋值给s；s有是头的下一个，则表示他是队列当中参与排队的线程而且是排在最前面的；为什么是s最前面不是h嘛？诚然h是队列里面的第一个，但是不是排队的第一个；
    * 因为h是持有锁的，但是不参与排队；这个也很好理解，比如你去买火车票，你如果是第一个这个时候售票员已经在给你服务了，你不算排队，你后面的才算排队；
    * 队列里面的h是不参与排队的这点一定要明白参考下面关于队列初始化的解释---因为h要么是虚拟出来的节点，要么是持有锁的节点；什么时候是虚拟的呢？什么时候是持有锁的节点呢？下文分析
    * 然后判断s是否等于空，其实就是判断队列里面是否只有一个数据；假设队列大于1个，那么肯定不成立（s==null---->false），因为大于一个h.next肯定不为空；
    * 由于是||运算如果返回false，还要判断s.thread != Thread.currentThread()；这里又分为两种情况
    *       2.1 s.thread != Thread.currentThread() 返回true，就是当前线程不等于在排队的第一个线程s；
    *             那么这个时候整体结果就是h!=t：true；（s==null false || s.thread != Thread.currentThread() true------> 最后true）结果： true && true 方法最终放回true，那么去则需要去排队
    *             其实这样符合情理，队列不为空，有人在排队，而且第一个排队的人和现在来参与竞争的人不是同一个，那么你就乖乖去排队
    *       2.2 s.thread != Thread.currentThread() 返回false 表示当前来参与竞争锁的线程和第一个排队的线程是同一个线程
    *           那么这个时候整体结果就是h!=t：true；（s==null false || s.thread != Thread.currentThread() false------> 最后false）结果 true && false 方法最终放回false，那么去则不需要去排队
    *           不需要排队则调用 compareAndSetState(0, acquires) 去改变计数器尝试上锁；这里又分为两种情况（日了狗了这一行代码；有同学课后反应说子路老师老师老是说这个AQS难，你现在仔细看看这一行代码的意义，真的不简单的）
    *            2.2.1 第一种情况加锁成功？有人会问为什么会成功啊，很简单假如这个时候h也就是持有锁的那个线程执行完了，释放锁了，那么肯定成功啊；成功则执行 setExclusiveOwnerThread(current)；然后返回true 自己看代码
    *            2.2.2 第二种情况加锁失败？有人会问为什么会失败啊。很简单假如这个时候h也就是持有锁的那个线程没执行完，没释放锁，那么肯定失败啊；失败则直接返回false，不会进else if（java基础，else if是相对于 if（c == 0)的）
    *                  那么如果失败怎么办呢？后面分析；
    *
    *----------第二种情况总结，如果队列被初始化了，而且至少有一个人在排队那么自己也去排队；但是有个插曲；他会去看看那个第一个排队的人是不是自己，如果是自己那么他就去尝试假设；尝试看看锁有没有释放
    *----------也合情合理，好比你去买票，如果有人排队，那么你乖乖排队，但是你会去看看第一个排队的人是不是你女朋友；或者男朋友
    *----------如果是你女朋友就相当于是你自己（这里实在想不出现实世界关于重入的例子，只能用男女朋友来替代），你就叫你女朋友看看售票员有没有搞完，有没有轮到你女朋友，因为你女朋友是第一个排队的
    *----------疑问：比如如果在在排队，那么他是park状态，如果是park状态，自己怎么还可能重入啊。希望有同学可以想出来为什么和我讨论一下，作为一个菜逼，希望有人教教我
```

```
    *
    * 3、队列被初始化了，但是里面只有一个数据；什么情况下才会出现这种情况呢？可能有人会说ts加锁的时候里面就只有一个数据；其实不是，因为队列初始化的时候会虚拟一个h作为头结点，当前线程作为第一个排队的节点
    * 为什么这么做呢？因为aqs认为h永远是不排队的，假设你不虚拟节点出来那么ts就是h，而ts其实需要排队的，因为这个时候tf可能没有执行完，ts得不到锁，故而他需要排队；
    * 那么为什么要虚拟为什么ts不直接排在tf之后呢，上面已经时说明白了，tf来上锁的时候队列都没有，他不进队列，故而ts无法排在tf之后，只能虚拟一个null节点出来；
    * 那么问题来了；究竟上面时候才会出现队列当中只有一个数据呢？假设原先队列里面有5个人在排队，当前面4个都执行完了，轮到第五个线程得到锁的时候；他会把自己设置成为头部，而尾部又没有，故而队列当中只有一个h就是第五个
    * 至于为什么需要把自己设置成头部；其实已经解释了，因为这个时候五个线程已经不排队了，他拿到锁了，所以他不参与排队，故而需要设置成为h；即头部；所以这个时间内，队列当中只有一个节点
    * 关于加锁成功后把自己设置成为头部的源码，后面会解析到；继续第三种情况的代码分析，记得这个时候队列已经初始化了，但是只有一个数据，并且这个数据所代表的线程是持有锁
    * h != t false 由于后面是&&运算，故而返回false可以不参与运算，整个方法返回false；不需要排队
    *
    *
    *-------------第三种情况总结：如果队列当中只有一个节点，而这种情况我们分析了，这个节点就是当前持有锁的那个节点，故而我不需要排队，进行cas；
    *-------------如果持有锁的线程释放了锁，那么我能成功上锁
    *-------------
    *
    **/
    return h != t &&
        ((s = h.next) == null || s.thread != Thread.currentThread());
}
```

加锁过程总结：公平锁和非公平锁

如果是第一个线程tf，那么和队列无关，线程直接持有锁。并且也不会初始化队列，如果接下来的线程都是交替执行，那么永远和AQS队列无关，都是直接线程持有锁，如果发生了竞争，比如tf持有锁的过程中T2来lock，那么这个时候就会初始化AQS，初始化AQS的时候会在队列的头部虚拟一个Thread为NULL的Node，因为队列当中的head永远是持有锁的那个node（除了第一次会虚拟一个，其他时候都是持有锁的那个线程锁封装的node），现在第一次的时候持有锁的是tf而tf不在队列当中所以虚拟了一个node节点，队列当中的除了head之外的所有的node都在park，当tf释放锁之后unpark某个（基本是队列当中的第二个，为什么是第二个呢？前面说过head永远是持有锁的那个node，当有时候也不会是第二个，比如第二个被cancel之后，至于为什么会被cancel，不在我们讨论范围之内，cancel的条件很苛刻，基本不会发生）node之后，node被唤醒，假设node是t2，那么这个时候会首先把t2变成head（sethead），在sethead方法里面会把t2代表的node设置为head，并且把node的Thread设置为null，为什么需要设置null？其实原因很简单，现在t2已经拿到锁了，node就不要排队了，那么node对Thread的引用就没有意义了。所以队列的head里面的Thread永远为null。

acquire方法的逻辑---到此我们已经解释完了!tryAcquire(arg)方法，为了方便我再次贴一下代码

```
public final void acquire(int arg) {
    //tryAcquire(arg)尝试加锁，如果加锁失败则会调用acquireQueued方法加入队列去排队，如果加锁成功则不会调用
    //acquireQueued方法下文会有解释
    //加入队列之后线程会立马park，等到解锁之后会被unpark，醒来之后判断自己是否被打断了，如果被打断了则执行selfInterrupt方法
    //为什么需要执行这个方法？下文解释
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

acquireQueued(addWaiter(Node.exclusive),arg))方法解析-----如果代码能执行到这里说tc需要排队

需要排队有两种情况---换言之代码能够执行到这里有两种情况：

1、tf持有了锁，并没有释放，所以tc来加锁的时候需要排队，但这个时候**队列并没有初始化**

2、tn(无所谓哪个线程，反正就是一个线程)持有了锁，那么由于加锁tn!=tf(tf是属于第一种情况，我们现在不考虑tf了)，所以**队列是一定被初始化了的**，tc来加锁，那么队列当中有人在排队，故而他也去排队

addWaiter(Node.EXCLUSIVE)源码分析：

```
private Node addWaiter(Node mode) {
    //由于AQS队列当中的元素类型为Node，故而需要把当前线程tc封装成为一个Node对象,下文我们叫做nc
    Node node = new Node(Thread.currentThread(), mode);
    //tail为对尾，赋值给pred
    Node pred = tail;
    //判断pred是否为空，其实就是判断对尾是否有节点，其实只要队列被初始化了对尾肯定不为空，假设队列里面只有一个元素，那么对尾和对首都是这个元素
    //换言之就是判断队列有没有初始化
    //上面我们说过代码执行到这里有两种情况，1、队列没有初始化和2、队列已经初始化了
    //pred不等于空表示第二种情况，队列被初始化了，如果是第二种情况那比较简单
   //直接把当前线程封装的nc的上一个节点设置成为pred即原来的对尾
   //继而把pred的下一个节点设置为当nc，这个nc自己成为对尾了
    if (pred != null) {
        //直接把当前线程封装的nc的上一个节点设置成为pred即原来的对尾，对应 10行的注释
        node.prev = pred;
        //这里需要cas，因为防止多个线程加锁，确保nc入队的时候是原子操作
        if (compareAndSetTail(pred, node)) {
            //继而把pred的下一个节点设置为当nc，这个nc自己成为对尾了 对应第11行注释
```

```
            pred.next = node;
            //然后把nc返回出去
            return node;
        }
    }
    //如果上面的if不成了就会执行到这里，表示第一种情况队列并没有初始化---下面32行解析这个方法
    enq(node);
    //返回nc
    return node;
}



private Node enq(final Node node) {//这里的node就是当前线程封装的node也就是nc
    //死循环
    for (;;) {
        //对尾复制给t，上面已经说过队列没有初始化，故而第一次循环t==null（因为是死循环，因此强调第一次，后面可能还有第二次、第三次，每次t的情况肯定不同）
        Node t = tail;
        //第一次循环成了成立
        if (t == null) { // Must initialize
            //new Node就是实例化一个Node对象下文我们成为nn，看下面87行Node类的结构，调用无参构造方法实例化出来的Node里面三个属性都为null
            //入队操作--compareAndSetHead继而把这个nn设置成为队列当中的头部，cas防止多线程、确保原子操作；记住这个时候队列当中只有一个，即nn
            if (compareAndSetHead(new Node()))
                //这个时候AQS队列当中只有一个元素，即头部=nn，所以为了确保队列的完整，设置头部等于尾部，即nn即是头也是尾
                //然后第一次循环结束
                tail = head;
        } else {
            node.prev = t;
            if (compareAndSetTail(t, node)) {
                t.next = node;
                return t;
            }
        }
    }
}
//为了方便 第二次循环我再贴一次代码来对第二遍循环解释
private Node enq(final Node node) {//这里的node就是当前线程封装的node也就是nc
    //死循环
    for (;;) {
        //对尾复制给t，由于第二次循环，故而tail==nn，即new出来的那个node
        Node t = tail;
        //第二次循环不成立
        if (t == null) { // Must initialize
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            //不成立故而进入else
            //首先把nc，当前线程所代表的的node的上一个节点改变为nn，因为这个时候nc需要入队，入队的时候需要把关系维护好
            //所谓的维护关系就是形成链表，nc的上一个节点只能为nn，这个很好理解
            node.prev = t;
            //入队操作--把nc设置为对尾，对首是nn，
            if (compareAndSetTail(t, node)) {
                //上面我们说了为了维护关系把nc的上一个节点设置为nn
                //这里同样为了维护关系，把nn的下一个节点设置为nc
                t.next = node;
                //然后返回t，即nn，死循环结束，也就是24行放回了，这个返回其实就是为了终止循环，返回出去的t，没有意义，你可以查看24行代码。
                return t;
            }
        }
    }
```

```
}

  //这个方法已经解释完成了
  enq(node);
  //返回nc，不管哪种情况都会返回nc：到此addWaiter方法解释完成
  return node;


//再次贴出node的结构方便大家查看
public class Node{
    volatile Node prev;
    volatile Node next;
    volatile Thread thread;
}



------------------总结：addWaiter方法就是让nc入队-并且维护队列的链表关系，但是由于情况复杂做了不同处理
------------------主要针对队列是否有初始化，没有初始化则new一个新的Node nn作为对首，nn里面的线程为null
------------------接下来分析acquireQueued方法
```

acquireQueued(addWaiter(Node.exclusive),arg))经过上面的解析之后可以理解成为acquireQueued(nc,arg))

```
final boolean acquireQueued(final Node node, int arg) {//这里的node 就是当前线程封装的那个node 下文叫做nc
    //记住标志很重要
    boolean failed = true;
    try {
        //同样是一个标志
        boolean interrupted = false;
        //死循环
        for (;;) {
            //获取nc的上一个节点，有两种情况：1、上一个节点为头部；2上一个节点不为头部
            final Node p = node.predecessor();
            //如果nc的上一个节点为头部，则表示nc为队列当中的第二个元素，为队列当中的第一个排队人；这里的第一和第二不冲突；我上文有解释；
            //如果nc为队列当中的第二个元素，第一个排队的则调用tryAcquire去尝试假设---关于tryAcquire看上面的分析
            //只有nc为第二个元素：第一个排队的情况下才会尝试加锁，其他情况直接去park了，因为第一个排队的执行到这里的时候需要看看持有有锁的线程有没有释放锁，释放了就轮到我了，就不park了
            //有人会疑惑说开始调用tryAcquire加锁失败了（需要排队），这里为什么还要进行tryAcquire不是重复了吗？
            //其实不然，因为第一次tryAcquire判断是否需要排队，如果需要排队，那么我就入队；当我入队之后我发觉前面那个人就是第一个，那么我不死心，再次问问前面那个人搞完没有
            //如果搞完了，我就不park，接着他搞；如果他没有搞完，那么我则在队列当中去park，等待别人叫我
            //但是如果我去排队，发觉前面那个人在睡觉，前面那个人都在睡觉，那么我也睡觉把---------------好好理解一下
            if (p == head && tryAcquire(arg)) {
                //能够执行到这里表示我来加锁的时候，锁被持有了，我去排队，进到队列当中的时候发觉我前面那个人没有park，前面那个人就是当前持有锁的那个人，那么我问问他搞完没有
                //能够进到这个里面就表示前面那个人搞完了；所以这里能执行到的几率比较小；但是在高并发的世界中这种情况真的需要考虑
                //如果我前面那个人搞完了，我nc得到锁了，那么前面那个人直接出队列，我自己则是对首；这行代码就是设置自己为对首
                setHead(node);
                //这里的P代表的就是刚刚搞完事的那个人，由于他的事情搞完了，要出队；怎么出队？把链表关系删除
                p.next = null; // help GC
                //设置表示---记住记加锁成功的时候为false
                failed = false;
                //返回false；为什么返回false 为了不调用50行---acquire方法当中的selfInterrupt方法；为什么不调用？下次解释比较复杂
                return interrupted;
            }
            //进到这里分为两种情况
            //1、nc的上一个节点不是头部，说白了，就是我去排队了，但是我上一个人不是队列第一个
            //2、第二种情况，我去排队了，发觉上一个节点是第一个，但是他还在搞事没有释放锁
            //不管哪种情况这个时候我都需要park，park之前我需要把上一个节点的状态改成park状态
            //这里比较难以理解为什么我需要去改变上一个节点的park状态呢？每个node都有一个状态，默认为0，表示无状态
            //-1表示在park；当时不能自己把自己改成-1状态？为什么呢？因为你得确定你自己park了才是能改为-1；不然你自己改成自己为-1；但是改完之后你没有park那不就骗人？
            //你对外宣布自己是单身状态，但是实际和刘宏斌私下约会；这有点坑人
            //所以只能先park；在改状态；但是问题你自己都park了；完全释放CPU资源了，故而没有办法执行任何代码了，所以只能别人来改；故而可以看到每次都是自己的后一个节点把自己改成-1状态
            if (shouldParkAfterFailedAcquire(p, node) &&
```

```
                //改上一个节点的状态成功之后；自己park；到此加锁过程说完了
                parkAndCheckInterrupt())
                interrupted = true;
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}



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

今天先写到这里---解释了加锁过程，好好看看--TMD这个文档很难写下次我慢慢更新；如果你看到这里觉得写得还不错可以去腾讯课堂给鲁班学院一个好评鼓励我接着写；哎。。。。太难打字了；

关于解锁和读写锁以及其他，我过两天更新，当中有错别字或者有歧义的地方记得和我沟通