2.深入理解AbstractQueuedSynchronizer(AQS)

2.1 AQS简介

在同步组件的实现中，AQS是核心部分，同步组件的实现者**通过重写AQS的方法实现自己想要表达的同步语义**，AQS则实现对**同步状态的管理，以及对阻塞线程进行排队，等待通知等一些底层的实现处理。AQS的核心也包括了这些方面，同步队列，独占式锁的获取与释放，共享锁的获取与释放以及可中断锁，超时锁获取这些特性的实现，而这些实际上则是AQS提供出来的模板方法，归纳如下：**

独占式锁：

1. void acquire(int arg) : 独占式获取同步状态，如果获取失败则插入同步队列进行等待。
2. void acquireInterruptibly(int arg) : 与acquire方法相同，但在同步队列中等待时可以响应中断。
3. boolean tryAcquireNanos(int arg,long nanosTimeout) : 在2的基础上增加了超时等待功能，在超时时间内没有获得同步状态返回false
4. boolean tryAcquire(int arg) : 获取锁成功返回true，否则返回false
5. boolean release(int arg) : 释放同步状态，该方法会唤醒在同步队列中的下一个节点。

1. void acquire(int arg) : 独占式获取同步状态，如果获取失败则插入同步队列进行等待。
2. void acquireInterruptibly(int arg) : 与acquire方法相同，但在同步队列中等待时可以响应中断。
3. boolean tryAcquireNanos(int arg,long nanosTimeout) : 在2的基础上增加了超时等待功能，在超时时间内没有获得同步状态返回false
4. boolean tryAcquire(int arg) : 获取锁成功返回true，否则返回false
5. boolean release(int arg) : 释放同步状态，该方法会唤醒在同步队列中的下一个节点。

共享式锁：

1. void acquireShared(int arg) : 共享式获取同步状态，与独占锁的区别在于同一时刻有多个线程获取同步状态。
2. void acquireSharedInterruptibly(int arg) : 增加了响应中断的功能
3. boolean tryAcquireSharedNanos(int arg,lone nanosTimeout) : 在2的基础上增加了超时等待功能
4. boolean releaseShared(int arg) : 共享锁释放同步状态。

1. void acquireShared(int arg) : 共享式获取同步状态，与独占锁的区别在于同一时刻有多个线程获取同步状态。
2. void acquireSharedInterruptibly(int arg) : 增加了响应中断的功能
3. boolean tryAcquireSharedNanos(int arg,lone nanosTimeout) : 在2的基础上增加了超时等待功能
4. boolean releaseShared(int arg) : 共享锁释放同步状态。

学习这些模板方法之前，首先要了解**AQS中的同步队列是一种什么样的数据结构，因为同步队列是AQS对同步状态的管理的基石。**

2.2 同步队列

当共享资源被某个线程占有，其他请求该资源的线程将会阻塞，从而进**入同步队列。就数据结构而言**，队列的实现 方式无外乎两者一是通过数组的形式，另外一种则是链表的形式。AQS中的同步队列则是通过链式方式进行实现。
1.节点的数据结构是什么?
2.单向双向
3.带头不带头

AQS有一个静态内部类 Node，这是我们同步队列每个**具体节点，在这个类中有如下几个属性**
节点属性如下：

```
1. volatile int waitStatus; // 节点状态
2. volatile Node prev; // 当前节点的前驱节点
3. volatile Node next; // 当前节点的后继节点
4. volatile Thread thread; // 当前节点所包装的线程对象
5. Node nextWaiter; // 等待队列中的下一个节点
```

节点状态如下：

```
1. int INITIAL = 0; // 初始状态
2. int CANCELLED = 1; // 当前节点从同步队列中取消
3. int SIGNAL = -1; // 后继节点的线程处于等待状态，如果当前节点释放同步状态会通知后继节点，使得后继节点的线程继续运行。

4. int CONDITION = -2; // 节点在等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了signal()方法后，该节点将会从等待队列中转移到同步队列中，加入到对同步状态的获取中。
5. int PROPAGATE = -3; // 表示下一次共享式同步状态获取将会无条件地被传播下去。
```

又发现源代码里面有如下几个属性：通过调试debug种的Debuger也可以看出，其结构。每个线程被封装成一个节点。又有前驱节点，后继节点，头节点。

```
*/
private transient volatile Node head;  //头节点  -同步队列

/**
* Tail of the wait queue, lazily initialized.  Modified only via
* method enq to add new wait node.
*/
private transient volatile Node tail;  //尾接点 -同步队列

 /**
 * The synchronization state.
 */
 private volatile int state;   //锁获取，和释放标记 -节点标记-（锁状态） 引用计数
```

由此我们知道了节点的数据结构：**带头（尾）节点的双向链表： 每个节点拥有其前驱节点和后继节点**

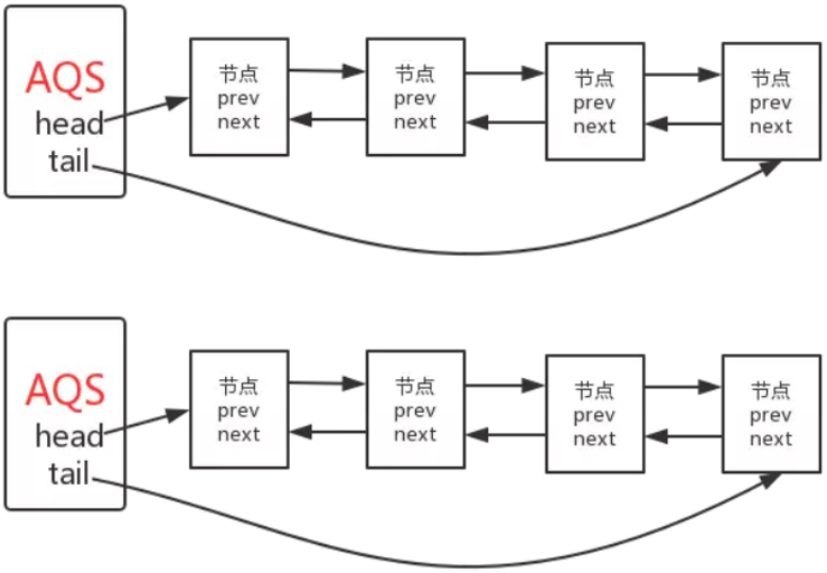
引用 Doug Lea 大神在设计AQS时AQS的数据结构中的注释

```
* <p>To enqueue into a CLH lock, you atomically splice it in as new
* tail. To dequeue, you just set the head field.
* <pre>
*      +------+  prev +-----+       +-----+
* head |      | <---- |     | <---- |     |  tail
*      +------+       +-----+       +-----+
* </pre>
```

源代码在如下：

file:///C:/Program%20Files/Java/jdk1.8.0_181/src.zip!/java/util/concurrent/locks/AbstractQueuedSynchronizer.java

也就是说AQS实际上是通过头尾指针来管理同步队列，同时包括获取锁失败的线程进行入队操作，释放时对同步队列中的线程进行通知等核心方法，其示意图如下：



通过如上分析：我们可以知道以下几点：
1.节点的数据结构，即AQS的静态内部类Node，节点的等待信息；
2.同步队列时一个双向链式队列，AQS通过持有头尾指针来管理同步队列；

2.3 独占锁

2.3.1独占锁的获取

lock()获取独占锁，获取失败就将当前线程加入同步队列，成功则线程执行。

ReentrantLock `NonfairSync中的（非公平锁）` lock()方法源码

```
final void lock() {
    //如果CAS替换成功，则当前线程直接获得锁 （应用场景：只有这一个线程在获取锁）
    if (compareAndSetState(0, 1))
        setExclusiveOwnerThread(Thread.currentThread()); //当前线程直接获得锁
    else
        acquire(1);//获取锁失败的化调用AQS提供的acquire(int arg)模板方法
}
```

获取锁失败的化调用AQS提供的acquire(int arg)模板方法
tryAcquire(arg):再次尝试获取同步状态，成功直接方法退出。

```
public final void acquire(int arg) {
        //1.首先调用子类覆写的方法tryAcquire（arg:1） 尝试获取同步状态 获取则直接退出，
    //失败继续调用addWaiter（Node.ExcLUSIVE）
    //最后才是accquireQueue()
        //再失败调用selfInterrupt() - 将当前线程状态置为 interruped
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg)){
                //如果条件为真，则当前线程调用interruped
        selfInterrupt();
     }
}
```

```
static void selfInterrupt() {
    Thread.currentThread().interrupt();---线程状态置为 interruped 线程正在阻塞中则 线程直接终止。
```

```
}
```

1.NonfairSync中的覆写的AQS的 tryAcquire

无锁或有锁但是：锁的重入，即可获得同步

```
protected final boolean tryAcquire(int acquires) {
    return nonfairTryAcquire(acquires); //调用 nonfairTryAcquire 此时 acquires==1 (想要获得锁)
}
nonfairTryAcquire（acquires:1）
final boolean nonfairTryAcquire(int acquires) {
    //当前线程
    final Thread current = Thread.currentThread();
    int c = getState();//获得 当前锁的状态（1：有锁 0：无锁）
    if (c == 0) { //当前锁状态 为无锁
        if (compareAndSetState(0, acquires)) {  //尝试CAS获取锁 acquire为1
            setExclusiveOwnerThread(current);   //将当前线程设置为当前独占线程
            return true;        //返回true - tryAcquire() 返回true —— acquire获得同步状态退出
        }
    }
    else if (current == getExclusiveOwnerThread()) {//当前锁状态为有锁 判断当前获得锁是否是当前线程（锁的重入）
        int nextc = c + acquires;         //锁标记设置为 c+1 ： 引用计数+1 锁的重入
        if (nextc < 0) // overflow        //如果锁标记小于0  则抛出异常 最大锁计数超出
            throw new Error("Maximum lock count exceeded");
        setState(nextc);                      //锁标记大于0，则更新锁标记+1（引用计数）
        return true;//返回true - tryAcquire() 返回true —— acquire获得同步状态退出
                    //（当前锁就是本线程持有，只是进行了锁的重入）
    }
    return false; //当前或的锁的线程不是本线程，返回false：tryAcquire() 返回false ，acquire进行下一个判断
                  //&&acquireQueued(addWaiter(Node.EXCLUSIVE), arg)) 中的addwaiter（）
}
```

2.获得同步状态失败

失败调用addwaiter()：将当前指针以指定模式（独占式，共享式）封装为Node节点后置入同步队列（尾插）

分析可以看**下面的注释。**程序的逻辑主要分为两个部分：**1. 当前同步队列的尾节点为null，调用方法enq()插入;**
2. 当 前队列的尾节点不为null，则采用尾插入（compareAndSetTail（）方法）的方式入队。
另外还会有另外一个问题： 如果 if (compareAndSetTail(pred, node))为false怎么办？会继续执行到enq()方法，同时很明显**compareAndSetTail 是一个CAS操作，通常来说如果CAS操作失败会继续自旋（死循环）进行重试。**

```
&&acquireQueued(addWaiter(Node.EXCLUSIVE), arg:1))  /传参是 模式：独占式 Node节点
private Node addWaiter(Node mode) {
    //将当前节点包装成节点，以独占式模式
    Node node = new Node(Thread.currentThread(), mode);  //线程作为节点数据
    // Try the fast path of enq; backup to full enq on failure
    //队列的尾接点
    Node pred = tail;
    if (pred != null) { //尾接点不为空，证明当前队列不为空
        node.prev = pred;     //当前节点的前驱节点为尾节点，准备尾插
        if (compareAndSetTail(pred, node)) {   //查看能否 将当前节点设置尾节点
            pred.next = node;     //尾插当前节点，到同步队列队尾
            return node;   //返回当前节点（同步队列尾接点的下一个节点）作为acquireQueued（node,arg）  node参数
        }
    }
                //当前队列为空（尾节点为空） 或CAS失败
    enq(node);          //调用enq方法
    return node;  //返回当前节点（同步队列尾接点的下一个节点）作为
```

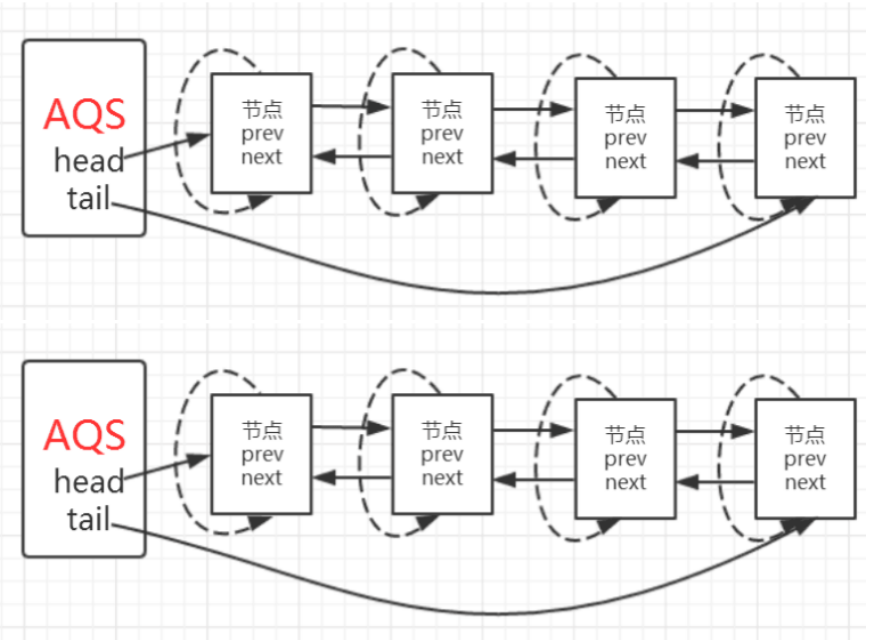
```
                    //   acquireQueued（node,arg）  node参数  传入的参数都是尾插成功的节点。
}
```

```
//当前队列为空（尾节点为空）  //死循环：自旋  ??  ：再次进入害怕同步队列前进行了一个出队列操作？
// 直到将当前节点插入同步队列成功为止
private Node enq(final Node node) {  //传参当前节点-独占式
    for (;;) {        //死循环：自旋
        Node t = tail;   //t为尾节点
        if (t == null) { // Must initialize  //如果尾接点为空（空队列）初始化同步队列
            if (compareAndSetHead(new Node())) //创建一个节点，并CAS查看头节点能否被改变
                tail = head;  //头尾节点置为一个点（初始化）
        } else {            //队列不为空  --尝试尾插
            node.prev = t;  //当前节点的前驱节点为尾节点，准备尾插
            if (compareAndSetTail(t, node)) { //查看能否 将当前节点设置尾节点
                                              // CAS尾插，失败进行自旋重试直到成功为止。
                t.next = node;       //尾插当前节点，到同步队列队尾
                return t;      //返回当前节点前驱节点（未插入之前的尾接点）
            }
        }
    }
}
```

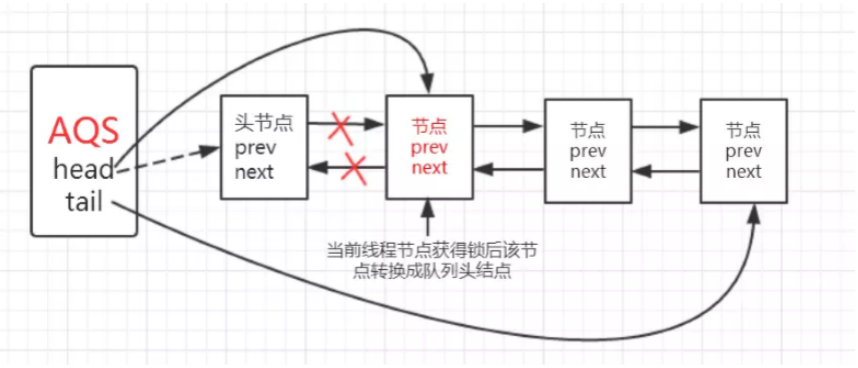
`enq(Node node)`:当前队列为空或者CAS尾插失败调
用此方法来初始化队列或不断自旋尝试尾插。

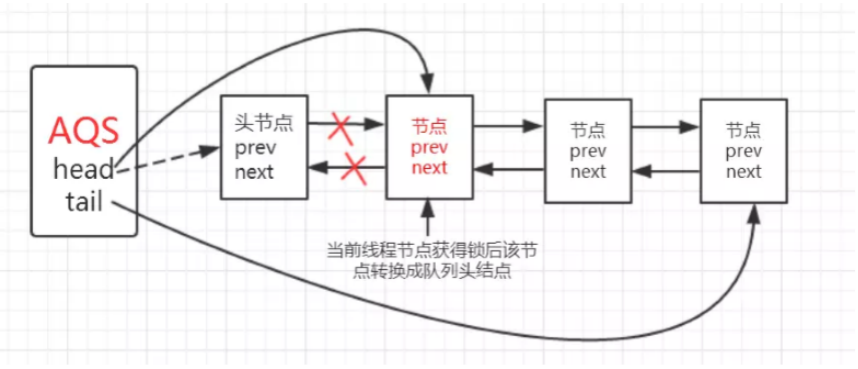
`enq()`方法可能承担两个任务：
1．在当前线程是第一个加入同步队列时，调用`compareAndSetHead(new Node())`方法，完成链式队列的头结点 的初始化；
2．自旋不断尝试CAS尾插入节点直至成功为止。

当前节点已经安模式（独占，共享式）封装成节点并尾插到同步队列的队尾。并返回这个节点给 acquireQueued 做参数
`acquireQueued()`获取锁成功条件:先驱节点是头结点的并且成功获得同步状态
程序逻辑通过注释已经标出，整体来看这是一个这又是一个自旋的过程（for（;;)），代码首先获取当前节点的先驱节点，如果先驱节点是头结点的并且成功获得同步状态的时候**if (p == head && tryAcquire(arg))** 当前节点所 指向的线程能够获取锁。
反之，获取锁失败进入等待状态。整体示意图为下图：



将当前节点通过setHead()方法设置为队列的头结点，然后将之前的头结点的next域设置为null并且pre域也为null，即与队列断开，无任何引用方便GC时能够将内存进行回收。






```
final boolean acquireQueued(final Node node, int arg) {//node:尾插到同步队列的队尾节点（当前线程）
    boolean failed = true;
    try {
        boolean interrupted = false;
        for (;;) {
            //获得当前节点的前驱节点
            final Node p = node.predecessor();
            //当前节点能否获得锁的前提：当前节点的前驱节点是头节点
            if (p == head && tryAcquire(arg)) {
                 //队列头节点指向当前节点
                 setHead(node);
                 //释放前驱节点
                 p.next = null; // help GC
                 failed = false;  //将取消节点 判断的标记设置为false 不会取消该节点;
                 return interrupted; //返回flase; ---acquire 当前线程已经获取到同步状态，直接退出
             }
                        // 获取同步状态失败，线程进入等待状态等待获取独占锁
                        //当线程已将在同步队列等待，则判断 parkAndCheckInterrupt()
             if (shouldParkAfterFailedAcquire(p, node) &&
                 parkAndCheckInterrupt())
                 interrupted = true; //    自旋到：当前节点的前驱节点是头节点    返回flase; ---acquire 调用selfInterrupt()设置中断标记位 ，试图中断当前线程（等待线程状态改为interruped则会中断）
         }
    } finally {  //最终取消当前节点
        if (failed)
            cancelAcquire(node);
    }
}

private void setHead(Node node) {
    head = node;
    node.thread = null;
```

```
32      node.prev = null;
33  }
```

那么当获取锁失败的时候会调用shouldParkAfterFailedAcquire()方法和parkAndCheckInterrupt()方法。

shouldParkAfterFailedAcquire()方法源码为：

```
1  private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) { //前驱，当前节点
2      int ws = pred.waitStatus; //前驱节点状态
3      if (ws == Node.SIGNAL)    //当前驱节点状态是SIGNAL=-1，表示后继节点的线程处于等待状态，如果当前节点释放同步状态会通知后继节点，使得后继节点的线程继续运行。
4          /*
5           * This node has already set status asking a release
6           * to signal it, so it can safely park.
7           */
8          return true; //当前节点已经在同步队列里等待了。继续进行parkAndCheckInterrupt()判断
9      if (ws > 0) { // CANCELLED = 1; // 前驱节点节点已经从同步队列中取消
10         /*
11          * Predecessor was cancelled. Skip over predecessors and
12          * indicate retry.
13          */
14         do {
15          node.prev = pred = pred.prev;//跳过它重新查找下一个不是被取消的前驱节点，并删除被取消的前驱节点
16         } while (pred.waitStatus > 0); // 不断重试直到找到一个前驱节点状态不为取消状态（为SIGNAL状态的节点）
17         pred.next = node; 删除被取消的前驱节点
18     } else { //前驱节点不是取消状态，且不是SIGNAL状态 0，  -2 -3
19         /*
20          * waitStatus must be 0 or PROPAGATE.  Indicate that we
21          * need a signal, but don't park yet.  Caller will need to
22          * retry to make sure it cannot acquire before parking.
23          */
24             //将前驱节点状态置为-1，
25            // 表示后继节点应该处于等待状态
26         compareAndSetWaitStatus(pred, ws, Node.SIGNAL); //将前驱节点状态置为-1，
27     }
28     return false; //  该线程已经被设置到等待状态 ，（前驱节点均是被取消）或者已将到队头，不用执行后续判断，进行自旋操作再次尝试获取锁。
29 }
```

1.节点在同步队列中获取锁

2.失败后调用shouldParkAferFaildAcquire(Node prev,Node node)

此方法主要逻辑

是使用CAS将前驱节点状态置为SIGNAL，表示需要将当前节点阻塞。

如果CAS失败，返回acquireQueued():不断自旋直到前驱节点状态置为SIGANL为止。

如果前驱节点状态已经为 -1，则调用parkAndCheckInterrupt()进行判断

```
if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                interrupted = true; //返回flase; ---acquire 调用selfInterrupt()设置中断标记位 ，试图中断当前线程（等待线程状态改为interruped则会中断）
        }
```

parkAndCheckInterrupt()进行判断

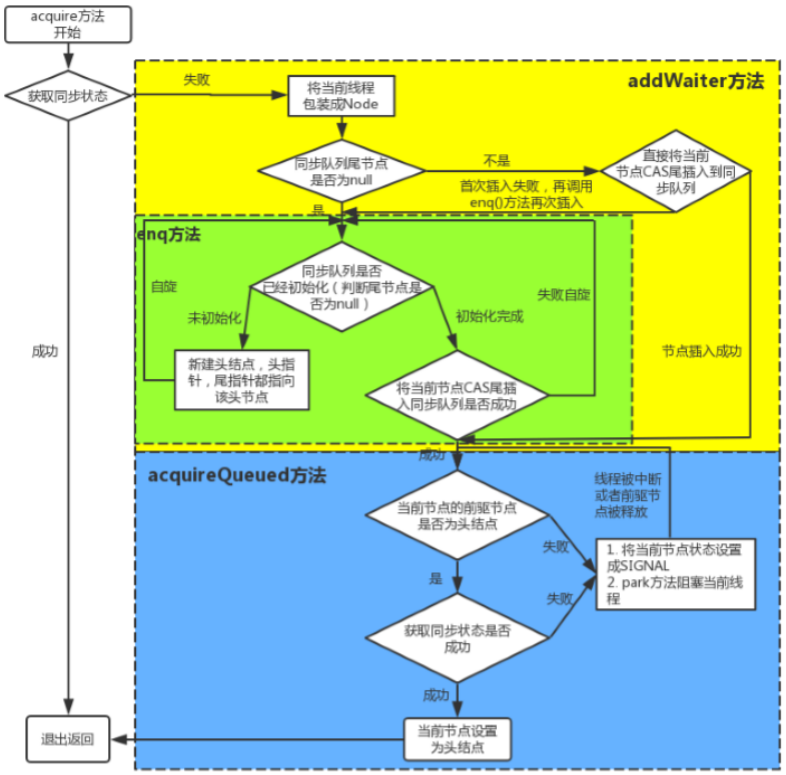
```
1  private final boolean parkAndCheckInterrupt() {
```

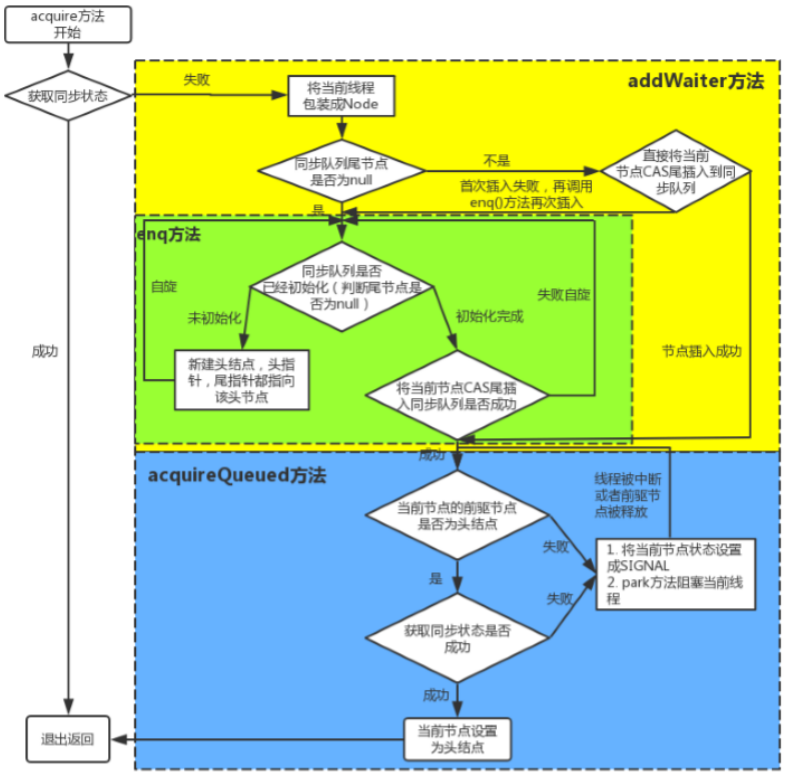
```
    LockSupport.park(this); //将当前节点阻塞
    return Thread.interrupted(); //返回当前线程是否已经被中断 中断返回true;
}


public static boolean interrupted() {
    return currentThread().isInterrupted(true);
}
```

总结：，acquireQueued()在自旋过程中主要完成了两件事情：

1. 如果当前节点的前驱节点是头节点，并且能够获得同步状态的话，当前线程能够获得锁该方法执行结束退出； 2. 获取锁失败的话，先将前驱节点状态设置成SIGNAL，再一次尝试获取锁失败后然后调用parkAndCheckInterrupt()中的LookSupport.park方法使得当前线程阻塞。





独占式锁的释放 -release()

独占锁的释放调用unlock方法，而该方法实际调用AQS的release方法。下面来看这两个方法的源码

```
public final boolean release(int arg) {
 //尝试释放锁
 if (tryRelease(arg)) {
 //获取到当前队列的头结点
 Node h = head;
 if (h != null && h.waitStatus != 0)
 //头节点不为空，且状态值不为0
 unparkSuccessor(h);
 //释放锁成功
  return true;
  }
  //释放失败，还没到释放的释放（引用计数不为0）
  return false;
}
```

ReentrantLock 覆写Lock的tryRelease：java\util\concurrent\locks\ReentrantLock.java

```

protected final boolean tryRelease(int releases) {
 int c = getState() - releases; //引用计数减一 （锁重入相关）
 if (Thread.currentThread() != getExclusiveOwnerThread()) //当前线程没有持有锁
 throw new IllegalMonitorStateException(); //抛出异常 锁状态异常，假如当前线程在同步队列中阻塞，则抛出异常使当前线程跳出自旋，进入fianlly取消该节点
 boolean free = false; //返回值
 if (c == 0) { //如果引用计数为0,则表示当前线程应该被释放了
 free = true; //改变返回值
```

```
 setExclusiveOwnerThread(null); //将当前持有独占式锁的线程置为null 即释放锁成功
  }
  setState(c); 将引用计数更新 为什么不用CAS?
  return free; // 释放锁成功，才返回true，反之更新重入锁的引用计数返回false;，release返回false;
 }
```

```
/**
 * The current owner of exclusive mode synchronization.
 */
private transient Thread exclusiveOwnerThread;
/**
 * Sets the thread that currently owns exclusive access.
 * A {@code null} argument indicates that no thread owns access.
 * This method does not otherwise impose any synchronization or
 * {@code volatile} field accesses.
  * @param thread the owner thread
  */
 //将当前持有独占式锁的线程置为null ，释放锁
 protected final void setExclusiveOwnerThread(Thread thread) {
  exclusiveOwnerThread = thread;
 }
```

```
protected final void setState(int newState) { //将状态值更新
 state = newState;
}
```

首先获取头节点的后继节点，当后继节点的时候会调用LookSupport.unpark()方法，该方法会唤醒该节点的后继节点所包装的线程。因此，每一次锁释放后就会唤醒队列中该节点的后继节点所引用的线 程，从而进一步可以佐证获得锁的过程是一个FIFO（先进先出）的过程。 锁释放成功，要唤醒下一个节点。

```
private void unparkSuccessor(Node node) { //唤醒头节点的下一个节点
 /*
 * If status is negative (i.e., possibly needing signal) try
 * to clear in anticipation of signalling. It is OK if this
 * fails or if status is changed by waiting thread.
 */
 int ws = node.waitStatus; //获取当前线程状态
 if (ws < 0) //线程状态小于0 -1，-2 -3
 compareAndSetWaitStatus(node, ws, 0); //将其CAS状态改为0-初始状态 唤醒下一个接节点

  /*
  * Thread to unpark is held in successor, which is normally
  * just the next node. But if cancelled or apparently null,
  * traverse backwards from tail to find the actual
  * non-cancelled successor.
  */


  Node s = node.next; //取得下一个节点的状态（即头节点的下一个节点状态）

  if (s == null || s.waitStatus > 0) { //如果下一节点为空即，有可能是尾节点，且节点状态大于0，即1-该节点从同步队列里面取消
  s = null; //将该节点置为空，从队列尾部向前找，找距离头节点最近的一个。
 //当头节点的下一个节点为空时
 //从同步队列尾部开始一直向前找到距离头节点最近的非空节点。
 //-------------------------------------------------------
```

```
 for (Node t = tail; t != null && t != node; t = t.prev) //从尾部一直向前找，直到找到头节点，或者找到一个为空的节点停止
  if (t.waitStatus <= 0)//如果该节点状态为状态不为取消状态，且不为空，
  s = t;//则一直更新该节点，即一直找到距离头节点的最近的一个非且状态不为取消的节点
 //-----------------------------------------------------------
 }
  if (s != null) //反之 头节点的下一个节点不为空且目前处于阻塞状态，或者找到距离头节点最进得到一个非空且状态不为取消状态的节点，唤醒该节点。
  LockSupport.unpark(s.thread); //唤醒该节点
 }
```

节点的状态如下:

1. int INITIAL = 0; // 初始状态
2. int CANCELLED = 1; // 当前节点从同步队列中取消
3. int SIGNAL = -1; // 后继节点的线程处于等待状态，如果当前节点释放同步状态会通知后继节点，使得后继节点的线程继续运行。
4. int CONDITION = -2; // 节点在等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了signal()方法后，该节点将会从等待队列中转移到同步队列中，加入到对同步状态的获取中。
5. int PROPAGATE = -3; // 表示下一次共享式同步状态获取将会无条件地被传播下去。

节点的状态如下:

1. int INITIAL = 0; // 初始状态
2. int CANCELLED = 1; // 当前节点从同步队列中取消
3. int SIGNAL = -1; // 后继节点的线程处于等待状态，如果当前节点释放同步状态会通知后继节点，使得后继节点的线程继续运行。
4. int CONDITION = -2; // 节点在等待队列中，节点线程等待在Condition上，当其他线程对Condition调用了signal()方法后，该节点将会从等待队列中转移到同步队列中，加入到对同步状态的获取中。
5. int PROPAGATE = -3; // 表示下一次共享式同步状态获取将会无条件地被传播下去。

`unparkSuccessor()`：唤醒头节点的下一个节点。（唤醒距离头节点最近的一个非空节点）

```
private void unparkSuccessor(Node node) {
/*
* If status is negative (i.e., possibly needing signal) try
* to clear in anticipation of signalling. It is OK if this
* fails or if status is changed by waiting thread.
*/
int ws = node.waitStatus;
if (ws < 0)
compareAndSetWaitStatus(node, ws, 0);

/*
* Thread to unpark is held in successor, which is normally
* just the next node. But if cancelled or apparently null,
* traverse backwards from tail to find the actual
* non-cancelled successor.
*/
Node s = node.next;
if (s == null || s.waitStatus > 0) {
s = null;
//当头节点的下一个节点为空时
//从同步队列尾部开始一直向前找到距离头节点最近的非空节点。
//-----------------------------------------------------------
for (Node t = tail; t != null && t != node; t = t.prev)
if (t.waitStatus <= 0)
s = t;
//-----------------------------------------------------------
}
if (s != null)
LockSupport.unpark(s.thread); 唤醒当前节点。          （距离头节点最近的非空节点）
}
```

首先获取头节点的后继节点，**当后继节点的时候会调用LookSupport.unpark()方法，该方 法会唤醒该节点的后继节点所包装的线程。因此，每一次锁释放后就会唤醒队列中该节点的后继节点所引用的线 程，从而进一步可以佐证获得锁的过程是一个FIFO（先进先出）的过程。**

1. 线程获取锁失败，线程被封装成Node进行入队操作，核心方法在于addWaiter()和enq()，同时enq()完成对同步队 列的头结点初始化工作以及CAS操作失败的重试;
2. 线程获取锁是一个自旋的过程，当且仅当 当前节点的前驱节点是头结点并且成功获得同步状态时，节点出队即该节点引用的线程获得锁，否则，当不满足条件时就会调用LookSupport.park()方法使得线程阻塞;
3. 释放锁的时候会唤醒后继节点;

总体来说：在获取同步状态时，AQS维护一个同步队列，获取同步状态失败的线程会加入到队列中进行自旋；移除队列（或停止自旋）的条件是前驱节点是头结点并且成功获得了同步状态。在释放同步状态时，同步器会调unparkSuccessor()方法唤醒后继节点。