**一．AbstractQueuedSynchronizer是干嘛的？**

用于构建锁或者其他同步组件的基本框架，AQS是一个抽象的基类，它内部通过一个FIFO的同步等待队列 + 一个volatile修饰的state来管理节点等待状态等，整个类采用模板模式实现，提供一些方法供子类实现，支持互斥和共享2种模式。

**二．同步队列**

在AQS里面，**同步队列**叫做CLH(Craig, Landin, and Hagersten)；当线程要入队的时候，只需要构建成一个**Node节点加入队尾即可**，如果要出队，只需要将**队首节点移除，同时唤醒下一个等待的节点**。这个CLH队列可以看成是一个双向链表实现的，刚开始队列里面只有一个空的Head节点，这个时候Head节点和Tail节点都是同一个节点，这个节点是虚拟的，当有线程获不到锁对象的时候，那么就需要构建一个Node节点往这个CLH队列里面加，同时更新节点之间的引用关系(主要就是链表的东西)，然后加入到队列里面后，一直自旋去尝试获取同步等待状态。

|  |
| --- |
| static final class Node {  //表示一个共享节点模式下等待的常量  static final Node SHARED = new Node();  //表示在互斥模式下等待的常量  static final Node EXCLUSIVE = null;  //表示当前节点线程被取消  static final int CANCELLED = 1;  //表示当前节点线程等待被唤醒  static final int SIGNAL = -1;  //表示当前节点线程正在等待某个条件  static final int CONDITION = -2;  //表示下一次共享模式的同步等待状态的获取将会一直循环下去  static final int PROPAGATE = -3;  volatile int waitStatus;  //当前节点的前一个节点  volatile Node prev;  //当前节点的后一个节点  volatile Node next;  //获取同步等待状态的当前线程  volatile Thread thread;  //当前节点的下一个处于等待状态的节点  Node nextWaiter;   final boolean isShared() {  return nextWaiter == SHARED;  }  //找到当前节点的前一个节点   final Node predecessor() throws NullPointerException {  Node p = prev;  if (p == null)  throw new NullPointerException();  else  return p;  }  Node() { // Used to establish initial head or SHARED marker  }  Node(Thread thread, Node mode) { // Used by addWaiter  this.nextWaiter = mode;  this.thread = thread;  }  Node(Thread thread, int waitStatus) { // Used by Condition  this.waitStatus = waitStatus;  this.thread = thread;  }  } |

AQS主要就靠这个Node节点来维护同步等待队列 。

**三、互斥模式**

在互斥模式下，请求去获取一个锁的方法如下（因为还提供了**支持中断和超时的实现**，后面分析，此处分析不支持中断和超时的）

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

Acquire方法，首先会调用tryAcquire，空方法，交给子类实现，如果tryAcquire方法返回true，说明这个线程获取了锁，不进行处理，如果返回false：

1、addWaiter 方法构建**一个互斥模式的节点Node**，将其**加入这个CLH队列**。

2、acquireQueued方法，当节点入队后，会一直自旋，acquireQueued方法主要用来处理这个自旋节点什么时候该被唤醒，是否该被中断等

|  |
| --- |
| private Node addWaiter(Node mode) {  //构造一个节点,第一个参数为当前线程,第二个参数为节点类型  Node node = new Node(Thread.currentThread(), mode);  //首先尝试快速入队尾，如果失败了再说,这里是将队尾节点tail赋值为一个新节点pred  Node pred = tail;  if (pred != null) {  node.prev = pred;  //CAS将当前节点设置为队尾节点  if (compareAndSetTail(pred, node)) {  //原来的队尾节点的下一个节点就是当前这个入队节点，也就是当前入队节点现在是队尾节点  pred.next = node;  return node;  }  }  //如果快速入队失败了，那么实行正常入队  enq(node);  return node;  }  ----------------------------------------------------------------------  private Node enq(final Node node) {  /\*\*一直自旋，相当于一个死循环 \*/  for (;;) {  Node t = tail;  //如果目前队列里面还没有任何节点，那么需要new一个Node节点来充当队列头节点,此时整个队列里面  if (t == null) { // Must initialize  if (compareAndSetHead(new Node()))  tail = head;  } else {  //将当前节点的前驱节点指向同步等待队列的尾节点。  node.prev = t;  //CAS设置队列尾节点为当前节点  if (compareAndSetTail(t, node)) {  //设置成功后，以前的队尾节点的下一个节点就指向了当前入队的节点，这时入队成功，然后返回  t.next = node;  return t;  }  }  }  } |

入队成功，进行自旋等待，看什么时候可以被唤醒去尝试获取锁

|  |
| --- |
| final boolean acquireQueued(final Node node, int arg) {  boolean failed = true;  try {  boolean interrupted = false;  //刚刚**加入到CLH队列的当前线程是进行自旋**，**去获取锁**  for (;;) {  //获取当前节点的上一个节点  final Node p = node.predecessor();  //如果上一个节点是头节点，说明队列里面只有自己在排队等待获取锁，那么就直接尝试去获取锁  if (p == head && tryAcquire(arg)) {  //如果获取锁成功，将当前节点设置为头节点  setHead(node);  //原本的头节点的下一个节点就不存在了，设置为null,便于gc  p.next = null; // help GC  failed = false;  return interrupted;  }  //如果上面返回失败了，那么首先看一下当前节点是否需要被阻塞,并检查中断状态  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

  如果这个入队的节点，刚开始不被允许取获取这个等待状态，那么就需要看一下这个节点是否应该被阻塞。什么意思呢，就是加入这个节点进入队列的时候，他前面已经有其他节点排着队了，根据FIFO的公平性原则，你这个节点必须得排队等待，在这里的话也就是说会被阻塞住，就没有必要一直去自旋尝试，只需要等待它的前一个节点来唤醒你就好了。

shouldParkAfterFailedAcquire，该方法主要是判断当前节点线程是否应该被阻塞

|  |
| --- |
| private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {  //获取前驱节点的等待状态  int ws = pred.waitStatus;  //如果前驱节点的等待状态是SIGNAL，表明前驱节点也在等待被唤醒，根据队列公平性原则，当前节点必须排队，所以当前节点需要被阻塞，这里就返回了true  if (ws == Node.SIGNAL)  return true;  //如果当前节点的前驱节点的等待状态为 取消状态(可能前驱节点被中断或者超时了)  if (ws > 0) {  // 因为前驱节点已经被取消了等于1，那么需要重新规划当前节点的前驱节点的引用，也就是当前节点的前驱节点是已经被取消的这个前驱节点的前驱节点 Node <---- preNod <-----Node  do {  node.prev = pred = pred.prev;  } while (pred.waitStatus > 0);  pred.next = node;  } else {  //其他情况CAS设置当前节点的前驱节点等待状态为SIGNAL，返回false  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);  }  return false;  } |

回到acquire方法，最后如果acquireQueued(addWaiter(Node.EXCLUSIVE), arg)返回true，说明当前线程被中断，会继续调用selfInterrupt方法

|  |
| --- |
| static void selfInterrupt() {  //中断当前线程  Thread.currentThread().interrupt();  } |

-------------------------------------------------------------------------------

上述过程是，当前未获取到锁的情况，当获取锁后，业务处理完后，就需要释放同步等待状态，会调用下面的方法来释放锁

|  |
| --- |
| public final boolean release(int arg) {  **//首先尝试去释放等待状态，tryRelease同样交给子类自己去个性化实现**  if (tryRelease(arg)) {  Node h = head;  //**如果head不为null且，该节点的等待状态不为0，那么就需要唤醒它的后续节点，让后续节点去获取锁**  if (h != null && h.waitStatus != 0)  unparkSuccessor(h);  return true;  }  return false;  } |

某个节点业务处理完后释放锁之后，需要唤醒它的后继节点，如果后继节点有节点已经被中断或者超时取消了，从tail开始往前遍历，知道找到距离node最近的waitStatus小于等于0的节点；然后进行唤醒

|  |
| --- |
| private void unparkSuccessor(Node node) {  //获取当前节点的等待状态  int ws = node.waitStatus;  if (ws < 0)  //如果节点等待状态小于0，那么就设置为0  compareAndSetWaitStatus(node, ws, 0);  Node s = node.next;  //如果当前节点的下一个节点为null或者说下一个节点的等待状态>0（也就是下一个节点被取消或者中断了），那么需要反向往前找到一个节点不为null且状态不大于0的节点，将它进行唤醒  if (s == null || s.waitStatus > 0) {  s = null;  for (Node t = tail; t != null && t != node; t = t.prev)  if (t.waitStatus <= 0)  s = t;  }  if (s != null)  //唤醒节点线程  LockSupport.unpark(s.thread);  } |

**四、共享模式**

共享模式获取锁方法

|  |
| --- |
| public final void acquireShared(int arg) {  if (tryAcquireShared(arg) < 0)  doAcquireShared(arg);  } |

 如果(tryAcquireShared返回小于0表示请求成功(该方法同样交给了子类去实现)，如果没有成功则继续调用doAcquireShared，看一下调用失败后的处理方法doAcquireShared

|  |
| --- |
| private void doAcquireShared(int arg) {  // **将当前线程以共享模式加入同步等待队列**  final Node node = addWaiter(Node.SHARED);  boolean failed = true;  try {  boolean interrupted = false;  //主循环  for (;;) {  //获取当前线程的前一个节点  final Node p = node.predecessor();  //如果p是头节点，那么继续调用tryAcquireShared尝试去获取锁  if (p == head) {  int r = tryAcquireShared(arg);  if (r >= 0) {  //p节点被移除，置空next引用，帮助GC  setHeadAndPropagate(node, r);  p.next = null; // help GC  if (interrupted)  selfInterrupt();  failed = false;  return;  }  }  //如果当前节点的前驱节点不是头节点，判断当前节点请求失败后是否要被阻塞，如果是，阻塞并保存当前线程中断状态。  if (shouldParkAfterFailedAcquire(p, node) &&  parkAndCheckInterrupt())  interrupted = true;  }  } finally {  if (failed)  cancelAcquire(node);  }  } |

共享模式下的释放操作

|  |
| --- |
| public final boolean releaseShared(int arg) {  if (tryReleaseShared(arg)) {  doReleaseShared();  return true;  }  return false;  }  ------------------------------------------------------------------------  private void doReleaseShared() {  for (;;) {  Node h = head;  //判断同步等待队列是否为空  if (h != null && h != tail) {  //如果不为空，获取头节点的等待状态。  int ws = h.waitStatus;  if (ws == Node.SIGNAL) {  //如果等待状态是SIGNAL，说明其后继节点需要唤醒  //尝试修改等待状态  if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))  continue; //如果修改失败，重新循环检测。  unparkSuccessor(h);//如果修改成功，唤醒头节点的后继节点。  }  else if (ws == 0 &&  !compareAndSetWaitStatus(h, 0, Node.PROPAGATE)) //如果等待状态是0，尝试将其(头节点)设置为PROPAGATE  continue; // 如果设置失败，继续循环检测。  }  if (h == head) // 如果过程中头节点没有发生变化，循环退出；否则需要继续检测。  break;  }  } |