AQS详解

在同步组件中，AQS是最核心部分，同步组件的实现依赖AQS提供的模板方法来实现同步组件语义。

AQS实现了对同步状态的管理，以及对阻塞线程进行排队、等待通知等等底层实现。

AQS核心组成:同步队列、独占锁的获取与释放、共享锁的获取与释放、可中断锁、超时锁。
这一系列功能的实现依赖于AQS提供的模板方法。

独占式锁

```
1 void acquire(int arg) : 独占式获取同步状态，如果获取失败插入同步队列进行等待。
2 void acquireInteruptibly(int arg)：在1的基础上，此方法可以在同步队列中响应中断
3 boolean tryAcquireNanos(int arg,long nanosTimeOut)：在2的基础增加了超时等待功能，
4 //到了预计时间还未获得锁直接返回。
5 boolean tryAcquire(int arg)：获取锁成功返回true，否则返回false
6 boolean release(int arg) : 释放同步状态，该方法会唤醒在同步队列的下一个节点。
```

共享式锁

```
1 void acquireShared(int arg) : 共享获取同步状态,同一时刻多个线程获取同步状态
2 void acquireSharedInterruptibly(int arg) : 在1的基础上增加响应中断
3 boolean tryAcquireSharedNanos(int arg,long nanosTimeOut):在2的基础上增加
4 //超时等待
5 boolean releaseShared(int arg) : 共享式释放同步状态
```

同步队列

在AQS内部有一个静态内部类Node，这是同步队列中每个具体的节点。

节点有如下属性:

```
1 int waitStatus:节点状态
2 Node prev:同步队列中前驱节点
3 Node next:同步队列中后继节点
4 Thread thread:当前节点包装的线程对象
5 Node nextWaiter:等待队列中下一个节点
```

节点状态值如下:

```
1
2 /** waitStatus value to indicate thread has cancelled */
3 //当前节点由于超时或者中断在同步队列中取消
4 static final int CANCELLED = 1;
```

```
/** waitStatus value to indicate successor's thread needs unparking */
//当前的节点的前驱节点被阻塞，当前节点在执行release或者cancel时需要执行unpark
//来唤醒后继节点
static final int SIGNAL = -1;
/** waitStatus value to indicate thread is waiting on condition */
 //节点处于等待队列中。当其他线程对Condition调用signal()方法后，该节点会
 //从等待队列移到同步队列中
 static final int CONDITION = -2;
 /**
  * waitStatus value to indicate the next acquireShared should
  * unconditionally propagate
  */
  //共享式同步状态会无条件传播
 static final int PROPAGATE = -3;
```

AQS同步队列采用带有头尾节点的双向链表

独占式锁获取 -- 源码分析：

独占式锁的获取，如下图所示为ReentrantLock源码：

```
/**
 * Performs lock. Try immediate barge, backing up to normal
 * acquire on failure.
 */
final void lock() {
 //使用CAS操作尝试将同步状态从0改为1，如果成功则将同步状态持有
 //线程置为当前线程，否则调用acquire()方法
 if (compareAndSetState(0, 1))
 setExclusiveOwnerThread(Thread.currentThread());
  else
  acquire(1);
 }
```

acquire()方法：

```
//再次尝试获取同步状态，如果成功将当前线程置为持有锁线程，方法退出
//失败则继续向下执行
public final void acquire(int arg) {
 if (!tryAcquire(arg) &&
 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
 selfInterrupt();
}
```

tryAcquire()方法：

```
protected boolean tryAcquire(int arg) {
 throw new UnsupportedOperationException();
}
```

尝试获取锁资源，成功返回true。具体资源获取/释放范式交由自定义同步器实现。ReentrantLock中默认为非公平锁，公平锁后面在讨论，关于非公平锁的具体实现方式如下：

nonfairTryAcquire()方法：尝试去获取锁，若当前锁资源没有被初始化，则直接将当前线程置为持有锁线程，若持有锁线程就是当前线程，重入，修改同步状态，若为以上两种情况返回true，否则返回false

```
/**
 * Performs non-fair tryLock. tryAcquire is implemented in
 * subclasses, but both need nonfair try for trylock method.
 * 执行非公平tryLock，tryAcquire是在子类中实现的，但是都
 * 需要tryLock方法的非公平尝试
 */

final boolean nonfairTryAcquire(int acquires) {
 final Thread current = Thread.currentThread(); //获取当前线程
 int c = getState(); //取得当前同步状态
 //若当前同步状态为0，表明还没有被初始化，则进行CAS操作修改同步状态
 //并当前线程置为持有锁线程，返回true
 if (c == 0) {
 if (compareAndSetState(0, acquires)) {
 setExclusiveOwnerThread(current);
 return true;
 }
 }
 //如果同步状态已经被初始化，则需要判断持有锁线程是否为当前线程
 //锁的可重入性，如果是当前线程，则将当前持有锁线程的状态值加一
 else if (current == getExclusiveOwnerThread()) {
 int nextc = c + acquires;
 //异常处理
 if (nextc < 0) // overflow
 throw new Error("Maximum lock count exceeded");
 //重新设置同步状态
 setState(nextc);
 return true;
 }
```

```
 return false;
}
```

---

addWaiter()方法：将当前线程封装成节点尾插入同步队列当中

```
/**
 * Creates and enqueues node for current thread and given mode.
 * 按照给定模式（独占式）为用节点封装当前线程并置入同步队列
 * @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared
 * @return the new node
 */
private Node addWaiter(Node mode) {
 //将当前线程封装为节点
 Node node = new Node(Thread.currentThread(), mode);
 // Try the fast path of enq; backup to full enq on failure
 Node pred = tail; //获取当前队列的尾节点
 //如果尾节点不为空则，使用CAS操作尝试将当前节点尾插入同步队列
 //如果成功返回当前节点
 if (pred != null) {
 node.prev = pred;
 if (compareAndSetTail(pred, node)) {
 pred.next = node;
 return node;
 }
 }
 //当前尾节点为空，或者CAS尾插失败就会执行该方法
 enq(node);
 return node;
}
```

enq()方法：

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. 如果CAS尾插入节点失败后负责自旋进行尝试；

```
/**
 * Inserts node into queue, initializing if necessary. See picture above.
 * 将节点插入到同步队列中，必要时初始化
 * @param node the node to insert
 * @return node's predecessor
 */
private Node enq(final Node node) {
```

```
for (;;) {
//当前节点为空
 Node t = tail;
 if (t == null) { // Must initialize
 //头节点初始化
 if (compareAndSetHead(new Node()))
 tail = head;
 } else {
 node.prev = t;
 //CAS尾插，失败则不断进行自旋重试直到成功为止
 if (compareAndSetTail(t, node)) {
 t.next = node;
 return t;
 }
 }
 }
}
```

---

acquireQueued()方法：　(排队获取锁) ---- 重要 ----

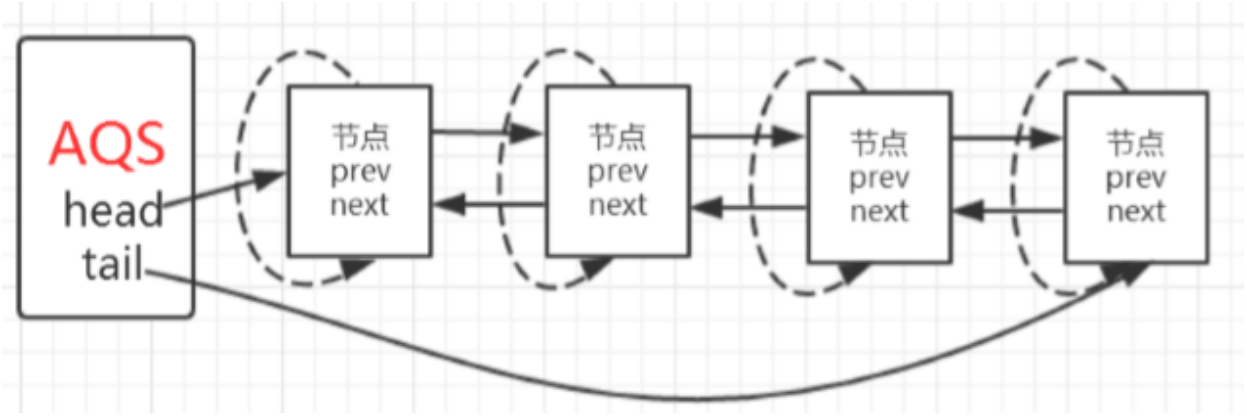
1. 如果当前节点的前驱节点是头结点，并且能够获得同步状态的话，当前线程能够获得锁，该方法执行结束退出。

2. 获取锁失败的话，先将节点状态设置为SIGNAL，然后调用LookSupport.park()方法使得当前线程阻塞。

```
/**
 * Acquires in exclusive uninterruptible mode for thread already in
 * queue. Used by condition wait methods as well as acquire.
 *
 * @param node the node
 * @param arg the acquire argument
 * @return {@code true} if interrupted while waiting
 */
// 自旋等待获取资源
final boolean acquireQueued(final Node node, int arg) {
 boolean failed = true;
 try {
 boolean interrupted = false;
 for (;;) {
 //当前节点的前驱节点
 final Node p = node.predecessor();
```

```
//如果前驱节点为头节点，则尝试获取同步状态
if (p == head && tryAcquire(arg)) {
//将队头指针指向当前节点
setHead(node);
//释放前驱节点
p.next = null; // help GC
failed = false;
return interrupted;
}
//获取同步状态失败，线程进入等待状态等待获取独占式
if (shouldParkAfterFailedAcquire(p, node) &&
parkAndCheckInterrupt())
interrupted = true;
}
} finally {
//如果获取资源失败，将当前节点置为取消状态
if (failed)
cancelAcquire(node);
}
}
```

如果前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。示意图如下：



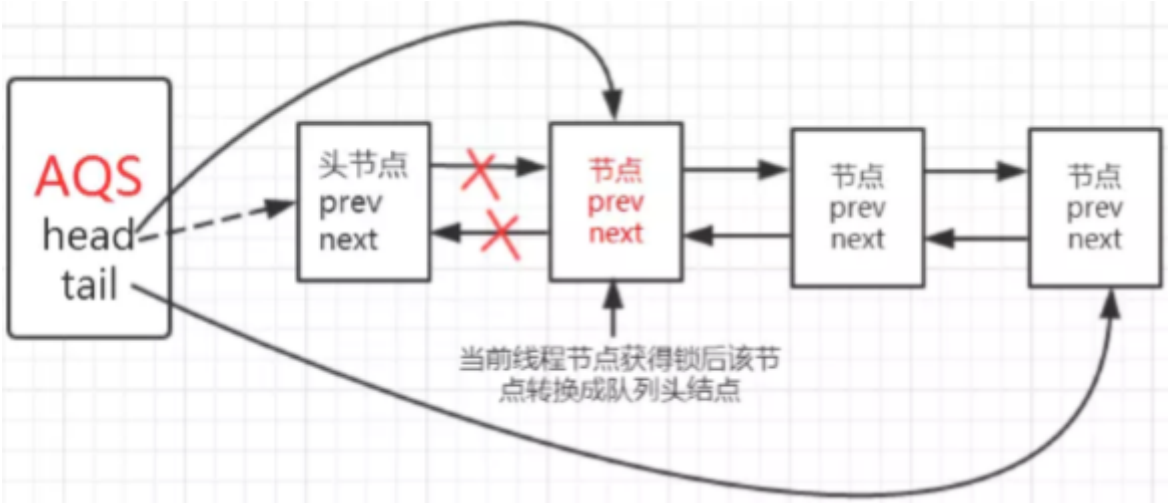
锁获取成功，进行出队操作：

```
//将队头指针指向当前节点
setHead(node);
//释放前驱节点
p.next = null; // help GC
failed = false;
return interrupted;
```

setHead() 方法：重新设置头结点，然后将当前节点prev指针置为null，将节点的thread置为null

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

通过上面流程，将原来的头结点next域置为null，并且它的prev域原本就是null，而新的队头指针已经指向了当前节点，这时无任何引用的原有头结点被GC进行回收。示意图如下：



当获取锁失败时调用如下方法：

shouldParkAfterFailedAcquire()方法：

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node)
{
 int ws = pred.waitStatus; //获取前驱节点状态
 //如果当前节点被阻塞，则直接点返回
 if (ws == Node.SIGNAL)
 /*
 * This node has already set status asking a release
 * to signal it, so it can safely park.
 */
 return true;
  //若当前节点被取消，则不断重试直到找到下一个不为取消状态的节点
  if (ws > 0) {
  /*
  * Predecessor was cancelled. Skip over predecessors and
  * indicate retry.
  */
  do {
  node.prev = pred = pred.prev;
```

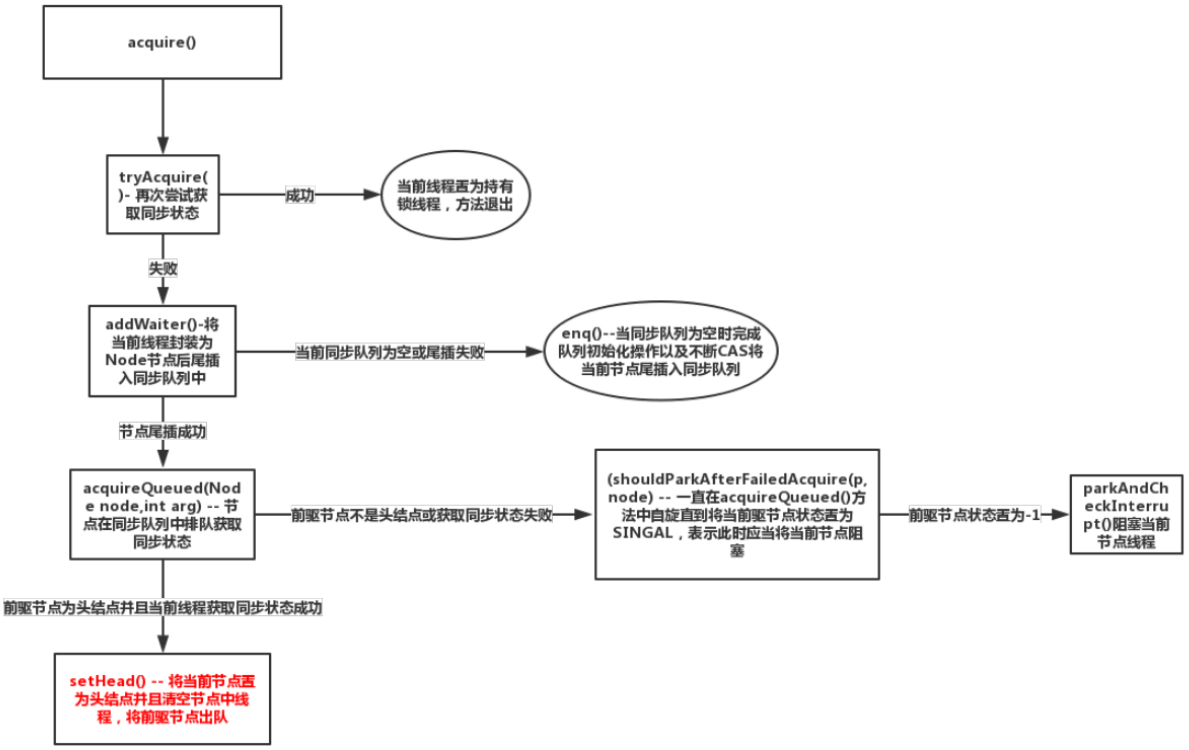
```
18  } while (pred.waitStatus > 0);
19  pred.next = node;
20  } else {
21  /*
22  * waitStatus must be 0 or PROPAGATE. Indicate that we
23  * need a signal, but don't park yet. Caller will need to
24  * retry to make sure it cannot acquire before parking.
25  */
26  //将节点状态修设置阻塞态，设置失败则返回false
27  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
28  }
29  return false;
```

当compareAndSetWaitStatus设置失败则说明 shouldParkAfterFailedAcquire方法返回false，然后会在acquireQueued()方法中for (;;)死循环中会继续重试，直至compareAndSetWaitStatus设置节点状态位为SIGNAL时shouldParkAfterFailedAcquire返回true时才会执行方法 parkAndCheckInterrupt()方法；

parkAndCheckInterrupt()方法：调用LockSupport.park()方法（后面讨论），阻塞当前线程

```
1 private final boolean parkAndCheckInterrupt() {
2  LockSupport.park(this);
3  return Thread.interrupted();
4 }
```

独占式锁的获取流程图如下：



独占式锁释放 -- 源码分析:

unlock()方法：调用AQS中的release方法

```
1 public void unlock() {
2  sync.release(1);
3 }
```

release()方法：当同步状态释放成功（返回true），执行if语句，当head指向的头结点不为null，并且该节点状态不为0时，执行unparkSuccessor()方法；

```
1 public final boolean release(int arg) {
2  if (tryRelease(arg)) { //尝试去释放资源
3  Node h = head; //头结点
4  if (h != null && h.waitStatus != 0)
5  unparkSuccessor(h); //唤醒head的下一个不为null的节点
6  return true;
7  }
8  return false;
9 }
```

tryRelease()方法：与tryAcquire一样，该方法需要通过同步器自定义实现。一般来说，释放资源直接用state减去给定参数releases，释放后state==0，说明释放成功，

```
protected final boolean tryRelease(int releases) {
 int c = getState() - releases;
 //当要释放的资源不是当前线程，就会抛出一个锁资源持有不合法异常
 if (Thread.currentThread() != getExclusiveOwnerThread())
 throw new IllegalMonitorStateException();
 boolean free = false;
 //如果c==0说明资源释放成功，将返回值改为true
 if (c == 0) {
 free = true;
  //将独占锁的持有线程置为null
  setExclusiveOwnerThread(null);
  }
  //重新设置节点状态
  setState(c);
  return free;
 }
```

unparkSuccessor()方法：唤醒head的下一个不为null的节点

```
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.
 */
 //如果状态为负(即(可能需要信号)试着清除预期信号。如果失败，或者等
 //待线程更改了状态，也没有关系。
 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.
  */
  //使用unpark唤醒head的和后继节点，但若head的后继节点被取消或者为null，
  //则从tail从后往前寻找head下一个不为空且没被取消的节点
  //头结点的后继节点
  Node s = node.next;
```

```
24  if (s == null || s.waitStatus > 0) {
25  s = null;
26  for (Node t = tail; t != null && t != node; t = t.prev)
27  if (t.waitStatus <= 0)
28  s = t;
29  }
30  //唤醒找到的节点
31  if (s != null)
32  LockSupport.unpark(s.thread);
33 }
```

release()方法是unlock()方法的具体实现。首先获取头结点的后继节点，当后继节点不为null，会调用LockSupport.unpark()方法唤醒后继节点包装的线程。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包装的线程。

总结：独占锁获取与释放总结:
1.线程获取锁失败，将线程调用addWaiter()封装成Node进行入队操作。
addWaiter()中方法enq()完成对同步队列的
头结点初始化以及CAS尾插失败后的重试处理。

2.入队之后排队获取锁的核心方法acquireQueued(),节点排队获取锁是一个自旋过程。
当且仅当当前节点的前驱节点为头结点并且成功获取同步状态时，
节点出队并且该节点引用的线程获取到锁。
否则，不满足条件时会不断自旋将前驱节点的状态
置为SIGNAL而后调用LockSupport.park()将当前线程阻塞。

3.释放锁时会唤醒后继节点(后继节点不为null)