# ReentranLock

作者：csh
原文链接：https://ld246.com/article/1666923911264
来源网站：链滴

## ReentrantLock#lock

```
public final void acquire(int arg) {
   //尝试获取锁,如果获取锁失败则尝试添加到同步队列中
      if (!tryAcquire(arg) &&
          acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
          selfInterrupt();
   }
```

ReentrantLock.FairSync#tryAcquire

尝试获取锁,如果当前线程是head的后继节点并且head线程已经释放锁的时候,根据队列取头部节点为占线程

```
protected final boolean tryAcquire(int acquires) {
          final Thread current = Thread.currentThread();
      //获取状态
          int c = getState();
          if (c == 0) {
       //如果队列为空或者队列不为空时当前线程是head的后继节点,并且设置status acquies成功
              if (!hasQueuedPredecessors() &&
                  compareAndSetState(0, acquires)) {
         //设置当前线程独占
                  setExclusiveOwnerThread(current);
                  return true;
              }
          }
    //锁的重入,state不为0说明当前线程已经占有锁再次请求锁
          else if (current == getExclusiveOwnerThread()) {
              int nextc = c + acquires;
              if (nextc < 0)
                  throw new Error("Maximum lock count exceeded");
       //设置status
              setState(nextc);
              return true;
          }
     //否则返回false,尝试加锁失败,将会尝试加入队列
          return false;
      }
   }
```

AbstractQueuedSynchronizer#acquireQueued

```
/**
     * 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
     */
    final boolean acquireQueued(final Node node, int arg) {
        boolean failed = true;
        try {
```

```
            boolean interrupted = false;
            for (;;) {
        //从等待队列中获取头节点才会尝试获取锁
                final Node p = node.predecessor();
        //如果pre是头节点,并且尝试获取锁成功,说明加入队列失败
                if (p == head && tryAcquire(arg)) {
         //移除头节点并且将当前node设置为头节点
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
                    return interrupted;
                }
        //不是头节点,或者尝试获取锁失败
                if (shouldParkAfterFailedAcquire(p, node) &&
                    parkAndCheckInterrupt())
                    interrupted = true;
            }
        } finally {
            if (failed)
                cancelAcquire(node);
        }
    }
```

AbstractQueuedSynchronizer#shouldParkAfterFailedAcquire

```
/**
     * Checks and updates status for a node that failed to acquire.
     * Returns true if thread should block. This is the main signal
     * control in all acquire loops.  Requires that pred == node.prev.
     *
     * @param pred node's predecessor holding status
     * @param node the node
     * @return {@code true} if thread should block
     */
    private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
        int ws = pred.waitStatus;
        if (ws == Node.SIGNAL)
            /*
             * This node has already set status asking a release
             * to signal it, so it can safely park.
             */
      //如果pred ws为等待一个释放信号去唤醒,说明node线程也可以安全的park
            return true;
        if (ws > 0) {
            /*
             * Predecessor was cancelled. Skip over predecessors and
             * indicate retry.
             */
      //prenode ws 已经取消,向上获取ws>0的node
            do {
                node.prev = pred = pred.prev;
            } while (pred.waitStatus > 0);
            pred.next = node;
        } else {
```

```
            /*
             * waitStatus must be 0 or PROPAGATE.  Indicate that we
             * need a signal, but don't park yet.  Caller will need to
             * retry to make sure it cannot acquire before parking.
             */
        //将preNode ws设置从ws设置为SIGNAL
            compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
        }
        return false;
    }
```

## ReentrantLock#tryLock(long, java.util.concurrent.TimeUnit)

```
public boolean tryLock(long timeout, TimeUnit unit)
            throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(timeout));
    }
```

AbstractQueuedSynchronizer#tryAcquireNanos

```
public final boolean tryAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (Thread.interrupted())
            throw new InterruptedException();
    //先去强势获取锁,失败后尝试在nanos内重试
        return tryAcquire(arg) ||
            doAcquireNanos(arg, nanosTimeout);
    }
```

AbstractQueuedSynchronizer#doAcquireNanos

```
/**
     * Acquires in exclusive timed mode.
     *
     * @param arg the acquire argument
     * @param nanosTimeout max wait time
     * @return {@code true} if acquired
     */
    private boolean doAcquireNanos(int arg, long nanosTimeout)
            throws InterruptedException {
        if (nanosTimeout <= 0L)
            return false;
        final long deadline = System.nanoTime() + nanosTimeout;
        final Node node = addWaiter(Node.EXCLUSIVE);
        boolean failed = true;
        try {
            for (;;) {
                final Node p = node.predecessor();
                if (p == head && tryAcquire(arg)) {
          //若predecessor是头节点并且当前线程尝试获取锁成功设置当前node为头节点,返回获取锁
功
                    setHead(node);
                    p.next = null; // help GC
                    failed = false;
```

```
                return true;
            }
            nanosTimeout = deadline - System.nanoTime();
            if (nanosTimeout <= 0L)
      //超时返回false
                return false;
    //见AbstractQueuedSynchronizer#shouldParkAfterFailedAcquire
            if (shouldParkAfterFailedAcquire(p, node) &&
                nanosTimeout > spinForTimeoutThreshold)
                LockSupport.parkNanos(this, nanosTimeout);
            if (Thread.interrupted())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```

## ReentrantLock#unlock

```
/**
* Attempts to release this lock.
*
* <p>If the current thread is the holder of this lock then the hold
* count is decremented.  If the hold count is now zero then the lock
* is released.  If the current thread is not the holder of this
* lock then {@link IllegalMonitorStateException} is thrown.
*
* @throws IllegalMonitorStateException if the current thread does not
*         hold this lock
*/
public void unlock() {
sync.release(1);
}
```

AbstractQueuedSynchronizer#release

```
public final boolean release(int arg) {
//尝试释放锁
if (tryRelease(arg)) {
Node h = head;
if (h != null && h.waitStatus != 0)
//唤醒头部节点
unparkSuccessor(h);
return true;
}
return false;
}
```

AbstractQueuedSynchronizer#unparkSuccessor

```
/**
```

```
* Wakes up node's successor, if one exists.
*
* @param node the node
  */
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;
for (Node t = tail; t != null && t != node; t = t.prev)
if (t.waitStatus <= 0)
s = t;
}
if (s != null)
//唤醒线程
LockSupport.unpark(s.thread);
}
```