- AQS 知识总结
  - 独占模式
    - 原理
    - 获取资源分析
    - 释放资源分析
  - 共享模式
    - 原理
    - 获取资源分析
    - 释放资源分析
    - Node.PROPAGATE 意义
  - 条件队列
    - 原理
    - 等待
    - 唤醒
  - 实践 —— 代码实现
    - 可重入锁实现
    - 信号量实现
    - 计数器实现
    - 测试代码

AQS 知识总结

AQS，全称 Abstract Queue Synchronizer，中文名称“同步阻塞队列”，在 `java.util.concurrent.locks` 包下面，**是一个抽象的可以实现阻塞线程、排队控制、唤醒线程等操作的同步器基础框架类**，AQS 可以实现排它锁、共享锁、条件锁、计数器等相关功能。

AQS 维护着一个同步队列，队列节点其实就是一个工作线程，每个节点都维护着其前驱节点与后继节点，节点中的 waitStatus 字段标识着节点是否被取消或是正在等待一个条件等。

如果用一句话来总结 AQS 的功能，那就是 **AQS 必须要保证整个队列入队和出队操作的原子性，同时要保证入队的线程在休眠后必须能够被唤醒**。

在 AQS 的实现中，**任何节点获取到资源时都会被设置成头节点，头节点的所有后继节点都是需要陷入休眠的节点，所以头节点必须要唤醒后继节点**。后继**节点是由其前驱节点唤醒的**，所以 AQS 的管理中，**一个节点如果需要陷入休眠，那么它必须将自己链接到一个能够唤醒它的前驱节点**，通俗点讲，就是找个“好爹”，而每个节点在释放锁时，也要忠实的履行职责，唤醒后继节点。

AQS 是一个抽象的可以实现阻塞线程、排队控制、唤醒线程等操作的同步器基础框架类，现在再来理解这句话，**阻塞线程、排队控制、唤醒线程**，AQS 并没有实现获取锁的核心逻辑，获取锁的真正逻辑是用户实现的，AQS 只是帮我们管理线程，没得到锁的线程进行休眠，并且要保证休眠的线程能够被唤醒，可以说这就是整个 AQS 要做的事情。

```
static final class Node {
    // 标志线程已被中断
    static final int CANCELLED = 1;

    // 标志节点在释放时，必须唤醒后继节点
    static final int SIGNAL = -1;

    // 标志节点正在等待一个条件，用以实现条件队列
    static final int CONDITION = -2;

    // 指示下一个 acquireShared 应该无条件地传播，即下一个节点 waitStatus 应该为 SIGNAL
    static final int PROPAGATE = -3;

    // waitStatus 为上述 4 种值，同时可能为 0，代表无意义
    volatile int waitStatus;
    volatile Node prev;
    volatile Node next;
    volatile Thread thread;

    // 为 null 代表独占模式，否则代表共享模式
    Node nextWaiter;
}
```

独占模式

原理

AQS 队列与其他队列并无两样，也是个先进先出的队列，因此新的节点入队时会被放置在队尾，如果节点获取锁（也称资源）失败，**节点将会陷入休眠，在休眠时节点会为自己找一个“好爹”以负责唤醒自己（节点会修改前驱节点的 ws = signal）**。

释放锁时，如果有必要，节点必须唤醒后继节点，让后续节点醒过来继续尝试获取锁。

在一开始时，**AQS 的 head 是一个伪头节点**(假定已经初始化)，直到某个节点成功获取锁才会被设置成头节点。

获取资源分析

acquire 方法是获取锁的入口方法：

```
public final void acquire(int arg) {
    // 如果尝试获取锁不成功，那么进入 acquireQueued 堵塞，不停的尝试获取锁，如果
```

```
acquireQueued 返回 true，则线程自我中断，线程销毁
    // 如果尝试获取锁成功，则不必堵塞或中断，线程继续运行；或者 acquireQueued 返回 false 同样如此
    // 目标: tryAcquire(arg) 返回 true，或 acquireQueued 返回 false，否则线程将中断
    if (!tryAcquire(arg) && acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
}
```

tryAcquire 方法是我们自己实现的核心方法，acquire 方法会至少调用一次 tryAcquire 方法，这也意味着**新加入的节点可能在第一次 tryAcquire 就成功了，那么这个新节点将不会入队而是直接运行，新节点直接抢占运行，而那些在等待队列中的线程却没得到运行，发生不公平的现象**，AQS 提供了 `hasQueuedPredecessors()` 方法以判断当前线程是否是等待时间最久的线程，以实现公平的锁。

> 如果所有线程一次 tryAcquire 就成功了，那么没有排队的线程，AQS 也就没有什么事情可以做了。

如果第一次 tryAcquire 失败，节点会通过 addWaiter 方法被加入至队列尾部，然后执行 acquireQueued 方法，**该方法是不停堵塞并尝试获取锁的核心方法**：

```
final boolean acquireQueued(final Node node, int arg) {
    boolean interrupted = false;
    try {
        // 无线循环，直到 获取锁 或 休眠成功
        for (; ; ) {
            // 获取上一个节点，如果上一个节点就是队头，并且当前节点成功获取到锁，则设置当前节点为头
            final Node p = node.predecessor();
            if (p == head && tryAcquire(arg)) {
                setHead(node);
                // 由于是独占模式，当前节点获取锁成功，那么上一个节点一定释放了锁，可以光荣退休
                p.next = null;
                // 返回 false，这将导致线程开始工作
                return interrupted;
            }
            // 获取锁失败，休眠等待唤醒
            // 否则查看自己是否能够休眠，如果能够休眠则进行休眠
            if (shouldParkAfterFailedAcquire(p, node))
                interrupted |= parkAndCheckInterrupt();
        }
    } catch (Throwable t) {
        cancelAcquire(node);
        if (interrupted)
            selfInterrupt();
        throw t;
    }
}
```

shouldParkAfterFailedAcquire(p, node) 是节点是否能够陷入休眠的核心方法：

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node)
{
    int ws = pred.waitStatus;
    // 如果前一个节点会通知自己，那么自己可以放心地睡
    if (ws == Node.SIGNAL) // ws == 1
        return true;
    if (ws > 0) {
        // 如果前节点死掉了，则重新链接到一个新的存活节点（找一个“好爹”）
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        // 否则前驱节点必须要承担起唤醒后续节点的重任
        pred.compareAndSetWaitStatus(ws, Node.SIGNAL);
    }
    // 否则，自己不能休眠，没有节点会唤醒自己，必须要在 acquireQueued 方法内不停
尝试 获取锁 以及 休眠
    return false;
}
```

可以发现在 `shouldParkAfterFailedAcquire(p, node)` 方法中，节点只有在前节点保证会唤醒自己的情况下才会陷入休眠，如果前节点死了，节点会遍历直至找到一个正常的节点链接上去；如果前节点正常，但是 ws != Node.SIGNAL，那么 AQS 会将前节点的 ws 设置为 Node.SIGNAL，然后继续重试。

释放资源分析

```
public final boolean release(int arg) {
    if (tryRelease(arg)) {
        Node h = head;
        // 在独占模式下，必须要唤醒后继节点
        // h 是可能为 null 的，这说明队列为空，那么也没有队列可以唤醒了
        // h.waitStatus 也是可能为 0 的，获取锁时，如果线程第二次就成功了，那么
将不会陷入休眠，也不会设置自身状态，而直接成为队头，这种情况下，h.waitStatus ==
0，这说明没有后续节点找到自己当“爹”
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}
```

unparkSuccessor 是唤醒线程的核心方法，当线程被唤醒时，线程从 acquireQueued 醒过来，会继续进入循环尝试获取锁。

这就是 AQS 管理独占模式队列的逻辑：能不管就不管，如果没办法获取锁，AQS 不得不管了，那么就保证你陷入休眠时有人唤醒，保证某些“承担大任”的节点出队时要唤醒后继节点。

共享模式

原理

共享模式允许多个节点共享资源，这也意味着**队列中可能会有多个正在获取资源的节点**，共享模式中节点释放资源并不会把节点移出队列，因为它们可能还会释放部分资源。

共享模式中陷入休眠的逻辑与独占模式并无两样，但**共享模式下在尝试获取资源时，如果发现还有资源可用，共享模式下会唤醒头节点的后继节点，让后继节点继续尝试获取资源。**

当节点获取到资源时，仍然会把节点设置为头节点，头节点必须要唤醒后继节点。

当节点释放资源时，**共享模式中可能会有非头节点释放资源，这种情况下也必须要唤醒头节点的后继节点**。

获取资源分析

```
public final void acquireShared(int arg) {
    // 如果为 -1 表示失败了，则进入同步队列堵塞获取锁
    // 如果成功了，不用入队
    if (tryAcquireShared(arg) < 0)
        doAcquireShared(arg);
}
```

tryAcquireShared(arg) 是具体获取资源的方法，这个方法的返回值为：

1. ret = -1：获取资源失败。
2. ret = 0：获取资源成功，但没有其他资源可用。
3. ret = 1：获取资源成功，并且还有资源可用。

来看 doAcquireShared 方法：

```
private void doAcquireShared(int arg) {
    // 标记为共享节点添加到队列
    final Node node = addWaiter(Node.SHARED);
    boolean interrupted = false;
    try {
        for (; ; ) {
            final Node p = node.predecessor();
            if (p == head) {
                int r = tryAcquireShared(arg);
```

```
                /**
                     * r = -1: 失败
                     * r = 0：成功，但没有其他资源了，不允许其他共享者进入了
                     * r = 1：成功，还有资源，允许其他共享者进入
                     */
                if (r >= 0) {
                    // 这是我们第一次见这个函数
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    // 注意，这里返回了，因此不会堵塞
                    return;
                }
            }
            // 堵塞等待唤醒
            if (shouldParkAfterFailedAcquire(p, node))
                interrupted |= parkAndCheckInterrupt();
        }
    } catch (Throwable t) {
        cancelAcquire(node);
        throw t;
    } finally {
        if (interrupted)
            selfInterrupt();
    }
}
```

可以发送与独占模式还是比较相似的，只有 `setHead` 变成了 `setHeadAndPropagate` ，来看看这个函数：

```
private void setHeadAndPropagate(Node node, int r) {
    // 注意！！！这里 h 是旧头部
    Node h = head;
    // 设置 node 为新头部
    setHead(node);
    // r > 0，说明还有资源可用，所以尝试唤醒休眠线程，这个很好理解
    // h.waitStatus 是旧头部的状态，h.waitStatus 不可能为 SIGNAL，因为当为
SIGNAL 时，后继线程应该正在休眠；一旦 h 唤醒后继节点后，h 自身也会通过 CAS 将 ws
设置为别的状态，总之，不可能为 SIGNAL
    // h.waitStatus 只可能为 PROPAGATE，为啥为 PROPAGATE 也要唤醒节点？我们后
面再讲
    if (r > 0 || h.waitStatus < 0) {
        Node s = node.next;
        if (s != null && !s.isShared()) {
            return;
        }
        // 因为还有资源可用，所以尝试唤醒头节点的后继节点
        doReleaseShared();
    }
}
```

当资源可用时或者旧头部的状态为 PROPAGATE 时，此时需要继续唤醒后继节点，这就是这个方法的逻辑。

释放资源分析

```
public final boolean releaseShared(int arg) {
    if (tryReleaseShared(arg)) {
        doReleaseShared();
        return true;
    }
    return false;
}
```

在释放资源时，一旦释放成功，则必会进入 `doReleaseShared()` 方法：

```
private void doReleaseShared() {
    for (; ; ) {
        // 找到当前头部节点，尝试唤醒后继节点
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            // 如果需要唤醒后继节点的话，那么尝试将自身状态设置为 0，并唤醒后继节
点
            if (ws == Node.SIGNAL) {
                if (!h.compareAndSetWaitStatus(Node.SIGNAL, 0))
continue;
                unparkSuccessor(h);
            }
            // 否则就将自己状态设置为 Node.PROPAGATE
            else if (ws == 0 && !h.compareAndSetWaitStatus(0,
Node.PROPAGATE))
                continue;                // loop on failed CAS
        }
        if (h == head)                   // loop if head changed
            break;
    }
}
```

`doReleaseShared()` 方法会尝试释放头节点的后继节点，而只要 tryReleaseShared 成功，就一定会执行 `doReleaseShared()` 方法，也就是说任**何一个节点释放资源时，都会保证去尝试唤醒一个正在休眠的节点**。

到这里，共享模式已经可以实现了，那 Node.PROPAGATE 字段有何意义呢？

Node.PROPAGATE 意义

Node.PROPAGATE 字段的出现是为了解决一个 Bug JDK-6801020

> 本节摘抄至 AQS源码深入分析之共享模式-你知道为什么AQS中要有PROPAGATE这个状态吗？_雕爷的架构之路-CSDN博客_aqs propagate

来看看离现在非常久远的Java 5u22中的该处代码是如何实现的：

```
private void setHeadAndPropagate(Node node, int propagate) {
  setHead(node);
  if (propagate > 0 && node.waitStatus != 0) {
      Node s = node.next;
      if (s == null || s.isShared())
          unparkSuccessor(node);
  }
}

 public final boolean releaseShared(int arg) {
  if (tryReleaseShared(arg)) {
      Node h = head;
      if (h != null && h.waitStatus != 0)
          unparkSuccessor(h);
      return true;
  }
  return false;
}
```

可以看到，早期版本的实现相比于现在的实现来说简单了很多，总结起来最主要的区别有以下几个：

在setHeadAndPropagate方法中，早期版本对节点waitStatus状态的判断只是!=0，而现在改为了<0； 早期版本的releaseShared方法中的执行逻辑和独占锁下的release方法是一样的，而现在将具体的唤醒逻辑写在了doReleaseShared方法里面，和setHeadAndPropagate方法共同调用。 而可能出现bug的测试代码如下：

```
import java.util.concurrent.Semaphore;

public class TestSemaphore {

  private static Semaphore sem = new Semaphore(0);

  private static class Thread1 extends Thread {
      @Override
      public void run() {
          sem.acquireUninterruptibly();
      }
  }

  private static class Thread2 extends Thread {
      @Override
      public void run() {
          sem.release();
      }
  }

  public static void main(String[] args) throws InterruptedException {
      for (int i = 0; i < 10000000; i++) {
          Thread t1 = new Thread1();
          Thread t2 = new Thread1();
```

```
            Thread t3 = new Thread2();
            Thread t4 = new Thread2();
            t1.start();
            t2.start();
            t3.start();
            t4.start();
            t1.join();
            t2.join();
            t3.join();
            t4.join();
            System.out.println(i);
        }
    }
}
```

其实上面所做的操作无非就是创建了四个线程：t1和t2用于获取信号量，而t3和t4用于释放信号量，其中的10000000次for循环是为了放大出现bug的几率，join操作是为了阻塞主线程。现在就可以说出出现bug的现象了：也就是这里可能会出现线程被hang住的情况发生。

可以想象这样一种场景：假如说当前CLH队列中有一个空节点和两个被阻塞的节点（t1和t2想要获取信号量但获取不到被阻塞在CLH队列中（state初始为0））：head->t1->t2（tail）。

- 时刻1：t3调用release->releaseShared->tryReleaseShared，将state+1变为1，同时发现此时的head节点不为null并且waitStatus为-1，于是继续调用unparkSuccessor方法，在该方法中会将head的waitStatus改为0；
- 时刻2：t1被上面t3调用的unparkSuccessor方法所唤醒，调用了tryAcquireShared，将state-1又变为了0。注意，此时还没有调用接下来的setHeadAndPropagate方法；
- 时刻3：t4调用release->releaseShared->tryReleaseShared，将state+1变为1，同时发现此时的head节点虽然不为null，但是waitStatus为0，所以就不会执行unparkSuccessor方法；
- 时刻4：t1执行setHeadAndPropagate->setHead，将头节点置为自己。但在此时propagate也就是剩余的state已经为0了（propagate是在时刻2时通过传参的方式传进来的，那个时候-1后剩余的state是0），所以也不会执行unparkSuccessor方法。

至此可以发现一轮循环走完后，CLH队列中的t2线程永远不会被唤醒，主线程也就永远处在阻塞中，这里也就出现了bug。那么来看一下现在的AQS代码在引入了PROPAGATE状态后，在面对同样的场景下是如何解决这个bug的：

- 时刻1：t3调用release->releaseShared->tryReleaseShared，将state+1变为1，继续调用doReleaseShared方法，将head的waitStatus改为0，同时调用unparkSuccessor方法；

- 时刻2：t1被上面t3调用的unparkSuccessor方法所唤醒，调用了tryAcquireShared，将state-1又变为了0。注意，此时还没有调用接下来的setHeadAndPropagate方法；
- 时刻3：t4调用release->releaseShared->tryReleaseShared，将state+1变为1，同时继续调用doReleaseShared方法，此时会将head的waitStatus改为PROPAGATE；
- 时刻4：t1执行setHeadAndPropagate->setHead，将新的head节点置为自己。虽然此时propagate依旧是0，但是“h.waitStatus < 0”这个条件是满足的（h现在是PROPAGATE状态），同时下一个节点也就是t2也是共享节点，所以会执行doReleaseShared方法，将新的head节点（t1）的waitStatus改为0，同时调用unparkSuccessor方法，此时也就会唤醒t2了。

至此就可以看出，在引入了PROPAGATE状态后，可以有效避免在高并发场景下可能出现的、线程没有被成功唤醒的情况出现。

条件队列

原理

条件队列由 AQS 内部类 `ConditionObject` 维护，关于条件变量介绍与使用可参考我的其他博客：**线程间同步方式**。

条件变量是一组等待唤醒的逻辑，在 AQS的实现中，等待的逻辑其实就是将线程加入由 ConditionObject 维护的条件队列中，由于 ConditionObject 内部并没有保证同步，所以等待时必须保证持有锁，AQS 保证线程在进入休眠时会释放锁。

唤醒的逻辑也很清晰，在 AQS 中，唤醒线程其实就是将由 ConditionObject 维护的条件队列中的头节点移至 AQS 中的同步等待队列中，唤醒该节点，然后节点就会进入 `acquireQueued` 方法，与独占模式中争抢资源的逻辑一致。

等待

```
public final void await() throws InterruptedException {
    // 将节点添加至 条件队列
    Node node = addConditionWaiter();

    // 尝试释放自己持有的锁，如果未持有锁会抛出异常
    int savedState = fullyRelease(node);
    int interruptMode = 0;

    // 如果不在同步等待队列的话（即还没被唤醒，如果被唤醒了会被移到等待队列中）
    while (!isOnSyncQueue(node)) {
        // 没有信号唤醒，那就休眠等待
        LockSupport.park(this);
        if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
            break;
```

```
    }

    // 被唤醒，以被添加到等待队列
    // 进入 acquireQueued，与独占模式逻辑相同，不停 争抢锁 或 休眠
    if (acquireQueued(node, savedState) && interruptMode != THROW_IE)
        interruptMode = REINTERRUPT;

    if (interruptMode != 0)
        reportInterruptAfterWait(interruptMode);
}
```

唤醒

```
private void doSignal(Node first) {
    do {
        if ( (firstWaiter = first.nextWaiter) == null)
            lastWaiter = null;
        first.nextWaiter = null;
        // 将队列中的头节点(等待时间最长的)作为参数，调用 transferForSignal
    } while (!transferForSignal(first) && (first = firstWaiter) != null);
}
```

```
final boolean transferForSignal(Node node) {
    // 如果无法设置，那么线程肯定被取消了，因为等待中的队列 ws 只可能是 CONDITION
    if (!node.compareAndSetWaitStatus(Node.CONDITION, 0))
        return false;

    // 将节点加入同步等待队列，这个返回的是前一个节点
    Node p = enq(node);
    int ws = p.waitStatus;
    // 将前一个节点设置为 SIGNAL，如果设置成功，自己可以放心休眠
    if (ws > 0 || !p.compareAndSetWaitStatus(ws, Node.SIGNAL))
        LockSupport.unpark(node.thread);
    return true;
}
```

实践 —— 代码实现

可重入锁实现

```
public class MyReentrantLock {
    private AbstractQueuedSynchronizer sync;

    /**
     * 实现不公平、可重入锁的 AQS
     */
    private class UnFairSync extends AbstractQueuedSynchronizer {
        @Override
```

```
        protected boolean tryAcquire(int arg) {
            int state = getState();
            // 如果资源可用，或持有锁的是当前线程，则尝试封锁资源
            if (state == 0 || getExclusiveOwnerThread() ==
Thread.currentThread()) {
                return compareAndSetState(0, arg);
            }
            return false;
        }

        @Override
        protected boolean tryRelease(int arg) {
            int state = getState();
            return compareAndSetState(state, state - arg);
        }
    }


    /**
     * 实现公平、可重入锁的 AQS
     */
    private class FairSync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int arg) {
            // 如果当前线程不是等待队列中的第一个线程，则不让其站有锁
            // 这是公平的锁，只有等待队列中等待时间最长的能够拥有锁
            if (getFirstQueuedThread() != Thread.currentThread()) {
                return false;
            }
            int state = getState();
            // 如果资源可用，或持有锁的是当前线程，则尝试封锁资源
            if (state == 0 || getExclusiveOwnerThread() ==
Thread.currentThread()) {
                return compareAndSetState(0, arg);
            }
            return false;
        }

        @Override
        protected boolean tryRelease(int arg) {
            int state = getState();
            return compareAndSetState(state, state - arg);
        }
    }

    public MyReentrantLock(boolean fair) {
        this.sync = fair ? new FairSync() : new UnFairSync();
    }

    public MyReentrantLock() {
        this(false); //默认不公平锁
    }

    public void lock() {
        sync.acquire(1);
```

```
    }

    public void unLock() {
        sync.release(1);
    }
}
```

信号量实现

```
public class MySemaphore {
    AbstractQueuedSynchronizer sync;

    public MySemaphore(int totalResource) {
        sync = new Sync(totalResource);
    }

    private class Sync extends AbstractQueuedSynchronizer {
        public Sync(int s) {
            setState(s);
        }

        @Override
        protected int tryAcquireShared(int arg) {
            int state = getState();
            System.out.println("state = " + state);
            // 如果资源够的话，则获取资源
            if (state >= arg && compareAndSetState(state, state - arg)) {
                // 如果获取资源之后还有资源的话，返回 1，否则返回 0
                return state - arg > 0 ? 1 : 0;
            }
            // 获取资源失败
            return -1;
        }

        @Override
        protected boolean tryReleaseShared(int arg) {
            int state = getState();
            return compareAndSetState(state, state + arg);
        }
    }

    public void acquire(int resource) {
        sync.acquireShared(resource);
    }

    public void release(int resource) {
        sync.releaseShared(resource);
    }
}
```

计数器实现

```
public class MyCountDownLatch {

    private AbstractQueuedSynchronizer sync;

    private class Sync extends AbstractQueuedSynchronizer {
        public Sync(int n) {
            // 初始化任务数
            setState(n);
        }

        @Override
        protected int tryAcquireShared(int arg) {
            int state = getState();
            // 等于 0 时代表任务已全部完成，返回 1，允许等待的线程 await
            // 大于 0 时，说明还有任务，允许其他线程领取任务，await 的需要继续等
待
            // 不可能小于 0
            return state == 0 ? 1 : -1;
        }

        @Override
        protected boolean tryReleaseShared(int arg) {
            int state = getState();
            if (state > 0) {
                // 尝试将任务数减少 1
                return compareAndSetState(state, state - 1);
            }
            return false;
        }
    }

    public MyCountDownLatch(int n) {
        sync = new Sync(n);
    }

    public void countDown() {
        sync.releaseShared(1);
    }

    public void await() {
        sync.acquireShared(1);
    }
}
```

测试代码

```
public class Main {
    public static void main(String[] args) throws InterruptedException {
        test1();// 测试可重入锁
        test2();// 测试信号量
        test3();// 计数器
    }
```

```
    synchronized static void countDown(MyCountDownLatch cdl, int id) {
        cdl.countDown();
        System.out.println("线程" + id + " 完成一个任务");
    }

    static void test3() {
        int N = 10;
        MyCountDownLatch cdl = new MyCountDownLatch(N);
        System.out.println("总任务数：" + N);
        for (int i = 0; i < N; i++) {
            final int id = i;
            new Thread(()-> {
                countDown(cdl, id);
            }).start();
        }
        cdl.await();
        System.out.println("任务全部完成，继续运行！");
    }

    static void test2() throws InterruptedException {
        MySemaphore sem = new MySemaphore(5);
        for (int i = 1; i <= 10; i++) {
            int id = i;
            new Thread(() -> {
                sem.acquire(1);
                System.out.println("线程" + id + "获取 1 个资源");
            }).start();
        }

        // 等待资源全部被获取
        Thread.sleep(100);

        // 释放 3 个资源
        System.out.println("\n主线程释放 3 个资源\n");
        sem.release(3);

        // 等待资源全部被获取
        Thread.sleep(100);

        // 释放 3 个资源
        System.out.println("\n主线程释放 2 个资源\n");
        sem.release(2);
    }

    static void test1() throws InterruptedException {
        MyReentrantLock lock = new MyReentrantLock();
        for (int i = 1; i <= 100; i++) {
            int id = i;
            new Thread(() -> {
                lock.lock();
                System.out.println("线程" + id + " 持有锁");
                System.out.println("线程" + id + " 释放锁");
                System.out.println();
                lock.unLock();
```

```
            }).start();
        }
    }
}
```