1 AQS 简介

- AQS 为抽象锁队列，它内部引入了一个锁队列，并实现了获取锁队列**失败后**线程的**入队列**操作和成功释放锁后，唤醒队列下一个节点线程的**出队列操作**
- AQS 也提供了获取和释放锁的**模板**，子类只要实现获取/释放锁(tryAcquire/tryRelease)的逻辑，就可以实现一个完成的锁，如 ReentrantLock、ReentrantReadWriteLock
- AQS、ReentrantLock、ReentrantReadWriteLock 结构与源码分析 一文中 分析了AQS 的结构

2 Semaphore 简介

- 需求分析：一个旅游景区，当人数达到额定值时，就不能在放入进去了，出来一个人才能进去一个人
- 概要设计：AQS 正好可以实现上面的需求
  - 可将 state 变量设为额定值（state 表示剩余的可获取的资源）
  - 当一个线程获取锁成功后，将 state - 1。释放锁成功后，将 state + 1
  - 当 state < = 0 时，表示没有资源了，获取资源的线程将被挂起

2.1 Semaphore 设计

- 以下即是最简洁的 Semaphore 所需要的核心代码，自己实现的仅仅只有 tryAcquireShared()、tryReleaseShared()。其余的部分都被 AQS 实现了，可见 AQS 的强大。

```
public class MySemaphore {
    private Sync sync;
    public MySemaphore(int permits) {
        this.sync = new Sync(permits);
    }
    //我觉得不应该响应中断，因为被中断的锁显然没有获取到锁，这时就会走 finally 中的 release() 方法来释放锁，这样就可能会抛 error 错误了
    public void acquire() {sync.acquireShared(1);}
    public void release() {sync.releaseShared(1);}
    static class Sync extends AbstractQueuedSynchronizer {
        private int max;
        public Sync(int permits) {
            setState(permits);
            max = permits;
        }
        @Override
        protected int tryAcquireShared(int arg) {
            int state;
            for (;;) {
```

```
                //当资源不够的时候就要阻塞当前线程了，否则一直循环CAS，直到成功为止，因为只要有资源就不应该被阻塞
                if ((state = getState()) - arg < 0) return -1;
                if (compareAndSetState(state,state-arg)){
                    return 1;
                }
            }
        }
        @Override
        //因为没有办法判断线程是否持有共享锁，为了防止随意释放，就直接抛 error 了
        //每次释放必定成功，因为每次释放，都应该唤醒等待队列中的线程
        protected boolean tryReleaseShared(int arg) {
            for(;;) {
                int state = getState();
                int nextS = state + arg;
                if (nextS > max) {
                    throw new Error("acquire 和  release 的个数不相同！");
                }
                if (compareAndSetState(state,nextS)) {
                    return true;
                }
            }
        }
    }
}
```

3 CountDownLatch 简介

- 需求分析：某场考试，只有所有考生都来了之后，监考老师才能发卷子（真实情况不可能哦）
- 概要设计：同样可以用 AQS 来实现
  - 可将额定值设为 state 变量（state 的值表示有多少把锁还没有被释放）
  - 某个考试入场，则释放一把锁（state - 1），如果 state == 0 时，才唤醒所有等待锁的线程(监考老师)
  - 并且获取锁成功的条件是 state <= 0

3.1 CountDownLatch 设计

- 同样最简洁的 CountDownLatch 所需要的核心代码，也仅仅实现了 tryAcquireShared() 和 tryReleaseShared() 方法，就将我们需要的功能实现了，可见 AQS 的强大

```
public class MyCountDownLatch {
    private Sync sync;
    public MyCountDownLatch(int count){sync = new Sync(count);}

    public void countDown(){sync.releaseShared(1);}
    //这里的 await() 方法可以设计成响应中断模式，因为 CountDownLatch 释放锁的操作是由其他线程调用的
    public void await() throws InterruptedException {sync.acquireSharedInterruptibly(1);}

    static class Sync extends AbstractQueuedSynchronizer {
        public Sync(int count) {setState(count);}
        //当 state 为 0 的时候才获取到，不为0就被阻塞
```

```
        @Override
        protected int tryAcquireShared(int arg) {return getState() <= 0 ? 1 : -1;}
        @Override
        protected boolean tryReleaseShared(int arg) {
            for(;;) {
                int state = getState();
                //如果 state 已经小于等于0了，说明等待线程已经被唤醒了。就不用这个线程来再次唤醒了
                if (state <= 0) return false;
                int nextS = state - arg;
                if (compareAndSetState(state,nextS)) {
                    return nextS <= 0;//这时如果小于等于0，就应该去唤醒等待线程了
                }
            }
        }
    }
}
```

4 总结

- 我们只需通过重写 AQS 中的 获取锁、释放锁方法，就可以实现各种功能的锁！
- 并且 AQS 已经为我们提供了中断锁、tryLock() 的全部代码。我们仅仅需要实现普通锁，就可以直接得到另外两把锁！