AQS java并发工具类

1.Tools工具类简单介绍

这些工具类基于AQS**共享**模式

包含5个工具类：Executors,Semaphore,Exchanger,CyclicBarrier,CountDownLatch.

2.Semaphore

> 使用场景： 限流....（降级）

> 艺术 p196

默认创建非公平锁

```
public Semaphore(int permits, boolean fair) {
        sync = fair ? new FairSync(permits) : new NonfairSync(permits);
    }
```

permits是设置state

```
protected final void setState(int newState) {
    state = newState;
}
```

使用:

```
semaphore.acquire();//获取锁
semaphore.release();
```

1.1acquire()源码解析

```
public void acquire() throws InterruptedException {
    sync.acquireSharedInterruptibly(1);
}
```

AQS。

```
public final void acquireSharedInterruptibly(int arg) //arg = 1;
        throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0) //尝试获取资源，失败执行do...阻塞线程
        doAcquireSharedInterruptibly(arg);
}
```

共享方式，可以拿多次。

tryAcquire()

tryAcquire() false --------tryAcquireShared返回的是负数

state可以是负数

```
    public boolean tryAcquire() {
        return sync.nonfairTryAcquireShared(1) >= 0;
    }

非公平获取锁
    final int nonfairTryAcquireShared(int acquires) {
        for (;;) {//自旋
            int available = getState();
            int remaining = available - acquires;// 可用资源-需要的资源
            if (remaining < 0 ||//判断剩余资源是否小于0
                compareAndSetState(available, remaining))
                return remaining;
        }
    }

公平获取锁

    protected int tryAcquireShared(int acquires) {
                for (;;) {
                    if (hasQueuedPredecessors())
                        return -1;
                    int available = getState();
                    int remaining = available - acquires;
                    if (remaining < 0 ||
                        compareAndSetState(available, remaining))
                        return remaining;
                }
            }
```

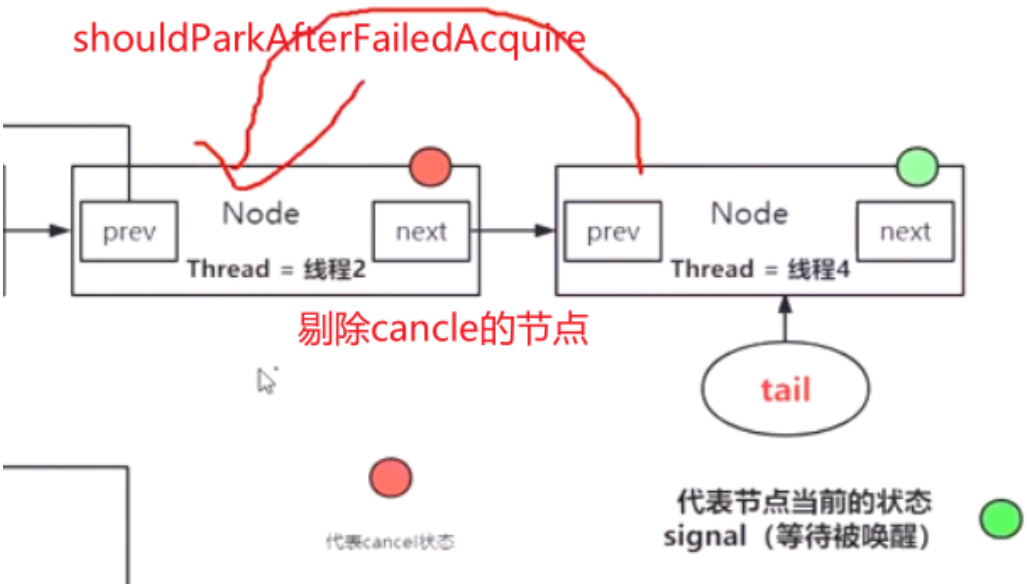
doAcquireSharedInterruptibly

如果获取资源失败：

- 节点入队操作。addWaiter(Node.SHARED);
- 阻塞线程

被唤醒后tryAcquireShared() 继续竞争资源，获取后return执行程序

```
private void doAcquireSharedInterruptibly(int arg)
    throws InterruptedException {
    final Node node = addWaiter(Node.SHARED);//共享节点 --->加入到同步队列
    boolean failed = true;
    try {
        for (;;) {
            final Node p = node.predecessor();//前驱结点
            if (p == head) {
                int r = tryAcquireShared(arg);//尝试获取资源， 失败就负数。
                if (r >= 0) {
                    setHeadAndPropagate(node, r);
                    p.next = null; // help GC
                    failed = false;
                    return;
                }
            }
            //parkAndCheckInterrupt可以阻塞线程
            if (shouldParkAfterFailedAcquire(p, node) &&
                parkAndCheckInterrupt())
                throw new InterruptedException();
        }
    } finally {
        if (failed)
            cancelAcquire(node);
    }
}
```



shouldParkAfterFailedAcquire

> 作用： 整理同步队列。

判断前驱节点waiteStatus （SiGNAL）是否可以唤醒?

cancleed（1）:移除节点

CONDITION = -2 PROPAGATE = -3 改为 sinal(-1)

```
private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        /*
         * 若前驱结点的状态是SIGNAL，意味着当前结点可以被安全地park
         */
        return true;
    if (ws > 0) {
        /*
         * 前驱节点状态如果被取消状态，将被移除出队列
         */
        do {
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        /*
        -2,-3,0 置换为 -1 sinal
         * 当前驱节点waitStatus为 0 or PROPAGATE状态时
         * 将其设置为SIGNAL状态，然后当前结点才可以可以被安全地park
         */
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}
```

一直cas cpu会非常高?

循环两次后， LockSupport.park(this);//阻塞

```
/**
 * 阻塞当前节点，返回当前Thread的中断状态
 * LockSupport.park 底层实现逻辑调用系统内核功能 pthread_mutex_lock 阻塞线程
 */
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);//阻塞
    return Thread.interrupted();
}
```

> 为什么可以套入那么多循环?

循环两次阻塞 --- 12 1:48:30

自己写程序的应用/AQS的玩儿法：

for(;;){

//读取redis

}

```
    private final boolean parkAndCheckInterrupt() {
         LockSupport.park(this);//阻塞
         return Thread.interrupted();
     }
```

p126 同步器的addWaiter和enq方法

```
private Node addWaiter(Node mode) {
    // 1. 将当前线程构建成Node类型
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    // 2. 1当前尾节点是否为null?
    if (pred != null) {
        // 2.2 将当前节点尾插入的方式
        node.prev = pred;
        // 2.3 CAS将节点插入同步队列的尾部
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}

/**
 * 节点加入CLH同步队列
 */
private Node enq(final Node node) {
    for (;;) {
        Node t = tail;
        if (t == null) { // Must initialize
            //队列为空需要初始化，创建空的头节点
            if (compareAndSetHead(new Node()))
                tail = head;
        } else {
            node.prev = t;
            //set尾部节点
            if (compareAndSetTail(t, node)) {//当前节点置为尾部
                t.next = node; //前驱节点的next指针指向当前节点
                return t;
            }
        }
    }
}
```

2.2.release()源码解析

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

releaseShared

1.尝试还资源tryReleaseShared

2.成功执行doReleaseShared

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

sync重写了AQS的tryReleaseShared

```
 protected final boolean tryReleaseShared(int releases) {
    for (;;) {
        int current = getState();
        int next = current + releases;
        if (next < current) // overflow
            throw new Error("Maximum permit count exceeded");
        if (compareAndSetState(current, next))
            //更新State：如果state是current则改为next
            return true;
    }
}
```

doReleaseShared()

```
/**
 * 把当前结点设置为SIGNAL或者PROPAGATE
 * 唤醒head.next(B节点)，B节点唤醒后可以竞争锁，成功后head->B，然后又会唤醒B.next，一直重复直到共享节点都唤醒
 * head节点状态为SIGNAL，重置head.waitStatus->0，唤醒head节点线程，唤醒后线程去竞争共享锁
 * head节点状态为0，将head.waitStatus->Node.PROPAGATE传播状态，表示需要将状态向后继节点传播
 */
private void doReleaseShared() {
    for (;;) {
        Node h = head;
        if (h != null && h != tail) {
            int ws = h.waitStatus;
            if (ws == Node.SIGNAL) {//head是SIGNAL状态
                /* head状态是SIGNAL，重置head节点waitStatus为0，这里不直接设为Node.PROPAGATE,
                 * 是因为unparkSuccessor(h)中，如果ws < 0会设置为0，所以ws先设置为0，再设置为PROPAGATE
```

```
                     * 这里需要控制并发，因为入口有setHeadAndPropagate跟release两个，避免两次unpark
                     */
                    if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
                        continue; //设置失败，重新循环
                    /* head状态为SIGNAL，且成功设置为0之后,唤醒head.next节点线程
                     * 此时head、head.next的线程都唤醒了，head.next会去竞争锁，成功后head会指向获取锁的节点，
                     * 也就是head发生了变化。看最底下一行代码可知，head发生变化后会重新循环，继续唤醒head的下一个节点
                     */
                    unparkSuccessor(h);
                    /*
                     * 如果本身头节点的waitStatus是出于重置状态（waitStatus==0）的，将其设置为“传播”状态。
                     * 意味着需要将状态向后一个节点传播
                     */
                }
                else if (ws == 0 &&
                         !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
                    continue;                // loop on failed CAS
            }
            if (h == head) //如果head变了，重新循环
                break;
        }
    }
```

唤醒 unparkSuccessor()

LockSupport.unpark(s.thread);//唤醒线程

唤醒后，等待的对列继续死循环 tryAcquire()方法。

```
    private void unparkSuccessor(Node node) {
        // node == head
        //获取wait状态
        int ws = node.waitStatus;
        if (ws < 0)
            compareAndSetWaitStatus(node, ws, 0);// 将等待状态waitStatus设置为初始值0

        /**
         * 若后继结点为空，或状态为CANCEL（已失效），则从后尾部往前遍历找到最前的一个处于正常阻塞状态的结点
         * 进行唤醒
         */
        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)
            LockSupport.unpark(s.thread);//唤醒线程
    }
```

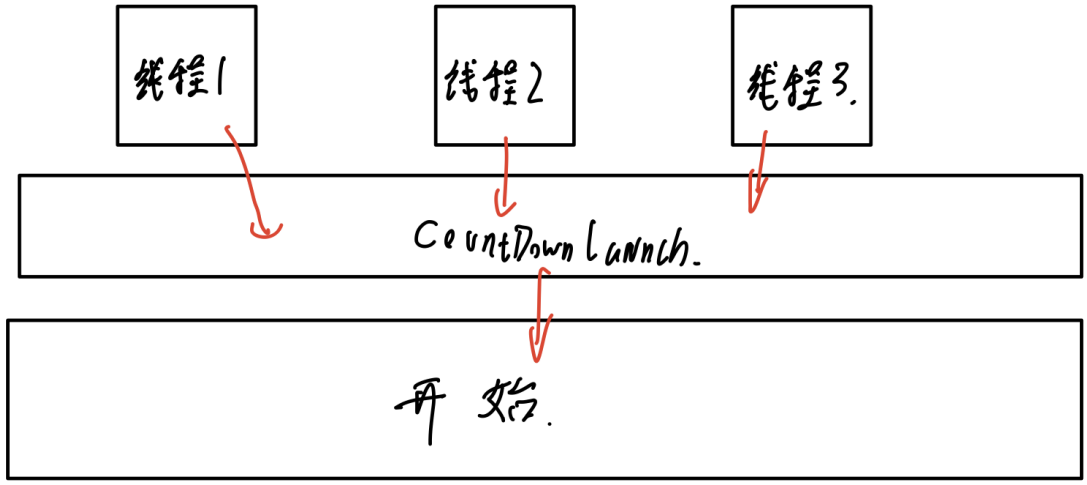
3.CountDownLatch

3.1CountDownLatch是什么?

CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行。例如，应用程序的主线程希望在负责启动框架服务的线程已经启动所有的框架服务之后再执行。

3.2CountDownLatch如何工作?

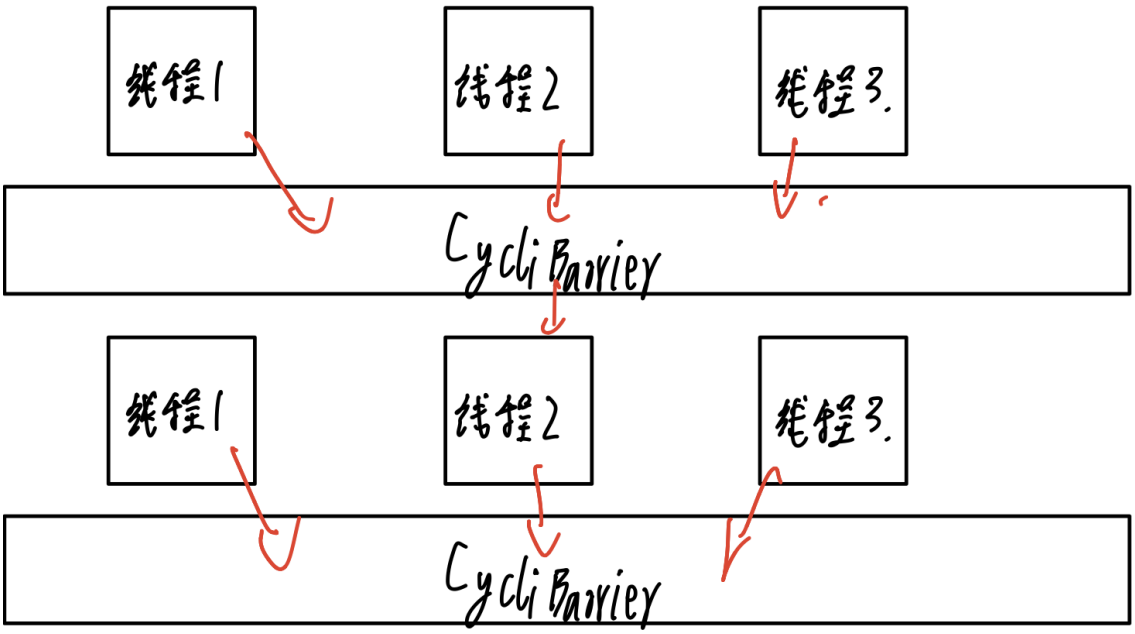
CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。



4.CycliBarrier

栅栏屏障，让一组线程到达一个屏障（也可以叫同步点）时被阻塞，直到最后一个线程到达屏障时，屏障才会开门，所有被屏障拦截的线程才会继续运行。 CyclicBarrier默认的构造方法是CyclicBarrier（int parties），其参数表示屏障拦截的线程数量，每个线程调用await方法告CyclicBarrier我已经到达了屏障，然后当前线程被阻塞。

> 栅栏可以重复使用



CountDownLatch和CycliBarrier区别：

CountDownLatch	CycliBarrier
不可以重复使用	可以重复使用
当计数器值到达0时，开始	等**最后一个线程执行完成** 可以开始

Exchanger

线程之间交换数据

Executors

线程池工具类