java.util.concurrent.Semaphore

**Semaphore的官方解释：什么玩意儿。。。。。。。**

**一个计数信号量，从概念上讲，信号量维护了一个许可集。如有必要，在许可可用前会阻塞每一个 acquire()，然后再获取该许可。 每个 release() 添加一个许可，从而可能释放一个正在阻塞的获取者。 但是，不使用实际的许可对象，Semaphore 只对可用许可的号码进行计数，并采取相应的行动。 拿到信号量的线程可以进入代码，否则就等待。通过acquire()和release()获取和释放访问许可。**

1.入门案例

**首先举一个例子：**

|  |
| --- |
| public static void main(String[] args) {   ExecutorService executorService1 = Executors.*newCachedThreadPool*();    final Semaphore semaphore = new Semaphore(5); // 只能5个线程同时访问。   for(int i = 0; i < 20; i++) {  final int NO = i;  Runnable run = new Runnable() {  @Override  public void run() {  try{  semaphore.acquire();  System.*out*.println("Accessing: "+NO);  Thread.*sleep*((long) (Math.*random*()\*10000));  } catch(InterruptedException e) {  e.printStackTrace();  }finally {  semaphore.release();  System.*out*.println("------------");  }  }  };  executorService1.execute(run);  }  executorService1.shutdown(); } |

**上面的例子演示了：同时最多有5个线程可以访问acquire()和semaphore之间的资源。**

2.Semaphore的内部构造：

**主要的就是Sync。private final Sync sync; Semaphore主要依靠这个成员变量来实现功能的。**

**Semaphore内有三个内部类，分别是Sync,FairSync,NonFairSync。FairSync和NonFairSync是Sync的子类，前者是公平锁，后者是非公平锁。**

3.主要方法：

**3.1 构造方法：**

|  |
| --- |
| **public Semaphore(int permits) { // permits：许可。**  **sync = new NonfairSync(permits); // Semaphore默认是使用不公平锁实现的。**  **}** |
| **public Semaphore(int permits, boolean fair) {**  **sync = fair ? new FairSync(permits) : new NonfairSync(permits);**  **}// true是公平的，false是不公平的，默认是false。** |

**3.2 普通方法**

|  |
| --- |
| **public void acquire() throws InterruptedException {**  **sync.acquireSharedInterruptibly(1);**  **}** |

**到这里，acquireSharedInterruptibly是调用的是AbstractQueuedSynchronizer的方法。**

|  |
| --- |
| **public final void acquireSharedInterruptibly(int arg)**  **throws InterruptedException {**  **if (Thread.interrupted())**  **throw new InterruptedException();**  **if (tryAcquireShared(arg) < 0)**  **doAcquireSharedInterruptibly(arg);**  **}** |

**这个方法的作用判断一下，线程是否interrupted，如果是，则中止。否，接着往下看。**

|  |
| --- |
| **private void doAcquireSharedInterruptibly(int arg)**  **throws InterruptedException {**  **final Node node = addWaiter(Node.SHARED);**  **boolean failed = true;**  **try {**  **for (;;) {**  **final Node p = node.predecessor();**  **if (p == head) {**  **int r = tryAcquireShared(arg);**  **if (r >= 0) {**  **setHeadAndPropagate(node, r);**  **p.next = null; // help GC**  **failed = false;**  **return;**  **}**  **}**  **if (shouldParkAfterFailedAcquire(p, node) &&**  **parkAndCheckInterrupt())**  **throw new InterruptedException();**  **}**  **} finally {**  **if (failed)**  **cancelAcquire(node);**  **}**  **}** |

**注意看 标红出的代码。注意了，这个地方，在子类中(FairSync和NonFairSync)中做了实现。先看不公平的吧。**

|  |
| --- |
| **protected int tryAcquireShared(int acquires) {**  **return nonfairTryAcquireShared(acquires);**  **}** |
| **final int nonfairTryAcquireShared(int acquires) {**  **for (;;) {**  **int available = getState();**  **int remaining = available - acquires;**  **if (remaining < 0 ||**  **compareAndSetState(available, remaining))**  **return remaining;**  **}**  **}** |

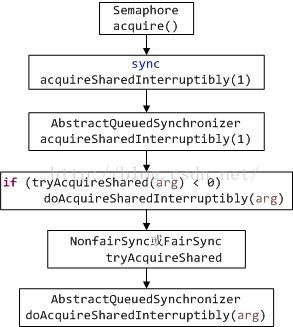
**下面的是公平的。**

|  |
| --- |
| **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;**  **}**  **}** |

**两者相比，公平的代码就多了一个if。也就是说这句话，是实现了公平，看看它是如何实现的。**

|  |
| --- |
| **public final boolean hasQueuedPredecessors() {**  **// The correctness of this depends on head being initialized**  **// before tail and on head.next being accurate if the current**  **// thread is first in queue.**  **Node t = tail; // Read fields in reverse initialization order**  **Node h = head;**  **Node s;**  **return h != t &&**  **((s = h.next) == null || s.thread != Thread.currentThread());**  **}** |

**从方法名上看 这个方法的功能是队列中是否含有等待的线程。先判断一下。如果有的话，为保证公平就将改线程加入到等待队列中去。**



**Release方法**

|  |
| --- |
| **public void release() {**  **sync.releaseShared(1);**  **}** |
| **public final boolean releaseShared(int arg) {**  **if (tryReleaseShared(arg)) {**  **doReleaseShared();**  **return true;**  **}**  **return false;**  **}** |
| **private void doReleaseShared() {**  **/\***  **\* Ensure that a release propagates, even if there are other**  **\* in-progress acquires/releases. This proceeds in the usual**  **\* way of trying to unparkSuccessor of head if it needs**  **\* signal. But if it does not, status is set to PROPAGATE to**  **\* ensure that upon release, propagation continues.**  **\* Additionally, we must loop in case a new node is added**  **\* while we are doing this. Also, unlike other uses of**  **\* unparkSuccessor, we need to know if CAS to reset status**  **\* fails, if so rechecking.**  **\*/**  **for (;;) {**  **Node h = head;**  **if (h != null && h != tail) {**  **int ws = h.waitStatus;**  **if (ws == Node.SIGNAL) {**  **if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))**  **continue; // loop to recheck cases**  **unparkSuccessor(h);**  **}**  **else if (ws == 0 &&**  **!compareAndSetWaitStatus(h, 0, Node.PROPAGATE))**  **continue; // loop on failed CAS**  **}**  **if (h == head) // loop if head changed**  **break;**  **}**  **}** |