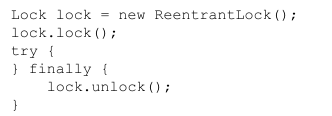
**1.Lock接口**

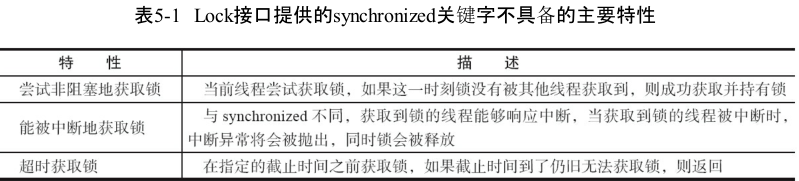
锁是用来控制**多个线程访问共享资源**的方式，一般来说，**一个锁**能够防止**多个线程同时访问共享资源**（但是**有些锁**可以**允许**多个线程并发的访问共享资源，比如**读写锁**）。Java SE 5之后，并发包中新增了**Lock接口**（以及相关实现类）用来实现锁功能，在使用时需要**显式**地**获取和释放锁**。虽然它缺少了（通过synchronized块或者方法所提供的）隐式获取释放锁的便捷性，但是却拥有了**锁获取与释放的可操作性**、**可中断的获取锁以及超时获取锁**等多种synchronized关键字所不具备的同步特性。

使用synchronized关键字将会**隐式地获取锁，但是它将锁的获取和释放固化了**，也就是**先获取再释放**。当然，这种方式简化了同步的管理，可是扩展性没有显示的锁获取和释放来的好。例如，针对一个场景，手把手进行锁获取和释放，先获得锁A，然后再获取锁B，当锁B获得后，释放锁A同时获取锁C，当锁C获得后，再释放B同时获取锁D，以此类推。这种场景下，synchronized关键字就不那么容易实现了，而使用Lock却容易许多。

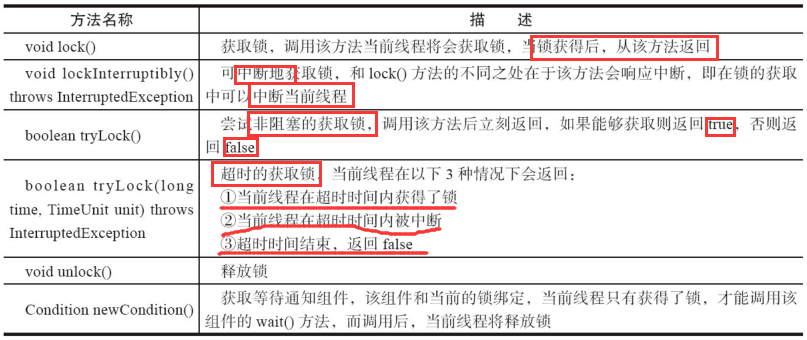


在finally块中释放锁，目的是**保证在获取到锁之后，最终能够被释放。**

不要将**获取锁的过程**写在**try块**中，因为如果在获取锁（自定义锁的实现）时**发生了异常**，异常抛出的同时，也会导致**锁无故释放**。



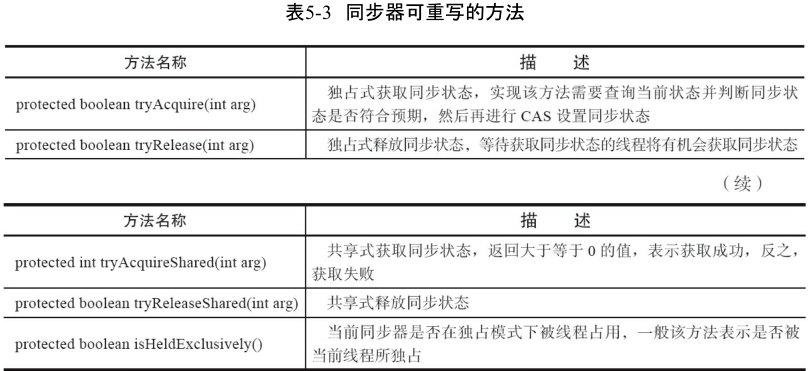
基本操作

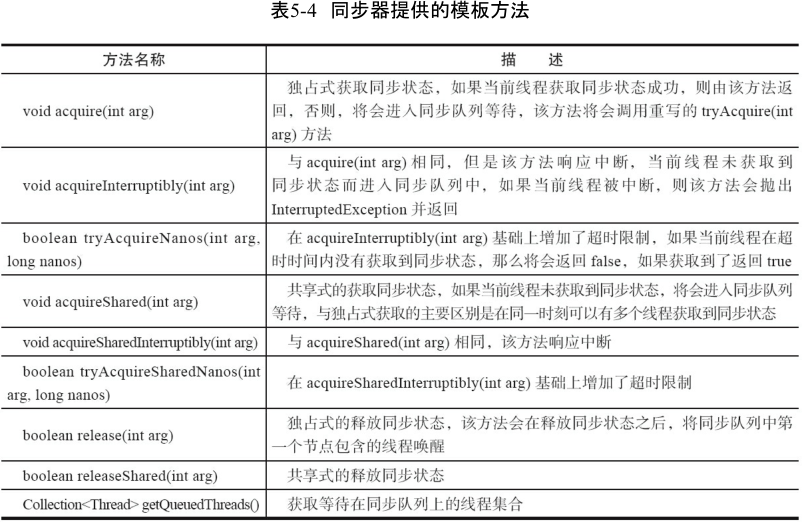


**2.队列同步器AQS**

队列同步器AbstractQueuedSynchronizer，是用来**构建锁**或者**其他同步组件的基础框架**，它使用了一个**int成员变量表示同步状态**，通过内置的**FIFO队列**来完成资源获取线程的**排队工作**。

同步器的主要使用方式是**继承**，子类通过继承同步器并实现它的抽象方法来管理同步状态，在抽象方法的实现过程中免不了要对同步状态进行更改，这时就需要使用同步器提供的3个方法（**getState()**、**setState(int newState)**和**compareAndSetState(int expect,int update)**）来进行操作，因为它们能够保证状态的改变是安全的。





同步器提供的模板方法基本上分为3类：**独占式获取与释放同步状态**、**共享式获取与释放同步状态**和**查询同步队列中的等待线程**情况。自定义同步组件将使用同步器提供的模板方法

来实现自己的同步语义。

顾名思义，独占锁就是在同一时刻**只能有一个线程获取到锁**，而**其他获取锁的线程只能处于同步队列中等待**，只有**获取锁的线程释放了**锁，**后继的线程才能够获取锁**，

|  |
| --- |
| **class** Mutex1 **implements** Lock {  // 静态内部类，自定义同步器  **private** **static** **class** Sync **extends** AbstractQueuedSynchronizer {  // 是否处于占用状态  **protected** **boolean** isHeldExclusively() {  **return** getState() == 1;  }  // 当状态为0的时候获取锁  **public** **boolean** tryAcquire(**int** acquires) {  **if** (compareAndSetState(0, 1)) {  setExclusiveOwnerThread(Thread.*currentThread*());  **return** **true**;  }  **return** **false**;  }  // 释放锁，将状态设置为0  **protected** **boolean** tryRelease(**int** releases) {  **if** (getState() == 0)  **throw** **new** IllegalMonitorStateException();  setExclusiveOwnerThread(**null**);  setState(0);  **return** **true**;  }  // 返回一个Condition，每个condition都包含了一个condition队列  Condition newCondition() {  **return** **new** ConditionObject();  }  }  // 仅需要将操作代理到Sync上即可  **private** **final** Sync sync = **new** Sync();  **public** **void** lock() {  sync.acquire(1);  }  **public** **boolean** tryLock() {  **return** sync.tryAcquire(1);  }  **public** **void** unlock() {  sync.release(1);  }  **public** Condition newCondition() {  **return** sync.newCondition();  }  **public** **boolean** isLocked() {  **return** sync.isHeldExclusively();  }  **public** **boolean** hasQueuedThreads() {  **return** sync.hasQueuedThreads();  }  **public** **void** lockInterruptibly() **throws** InterruptedException {  sync.acquireInterruptibly(1);  }  **public** **boolean** tryLock(**long** timeout, TimeUnit unit) **throws** InterruptedException {  **return** sync.tryAcquireNanos(1, unit.toNanos(timeout));  }  } |

**独占锁Mutex**是一个自定义同步组件，它在同一时刻**只允许一个线程占有锁，实现了tryAcquire和tryRelease独占方法**。Mutex中定义了一个静态内部类Sync，该内部类继承了**同步器**并实现了**独占式获取和释放同步状态**。在tryAcquire(int acquires)方法中，如果原来状态是0然后经过CAS设置成功（同步状态设置为1），则代表获取了同步状态，然后设置当前线程为拥有锁线程，而在tryRelease(int releases)方法中只是将同步状态重置为0，并将拥有锁线程设置为空。用户使用Mutex时并**不会直接和内部同步器的实现打交道**，而是调用**Mutex提供的方法**，在Mutex的实现中，以获取锁的lock()方法为例，只需要在方法实现中调用**同步器的模板方法acquire(int args)即可**，当前线程调用该方法获取**同步状态失败后**会被加入到**同步队列中等待**，这样就大大降低了实现一个可靠自定义同步组件的门槛。

**2.1队列同行器的实现分析**

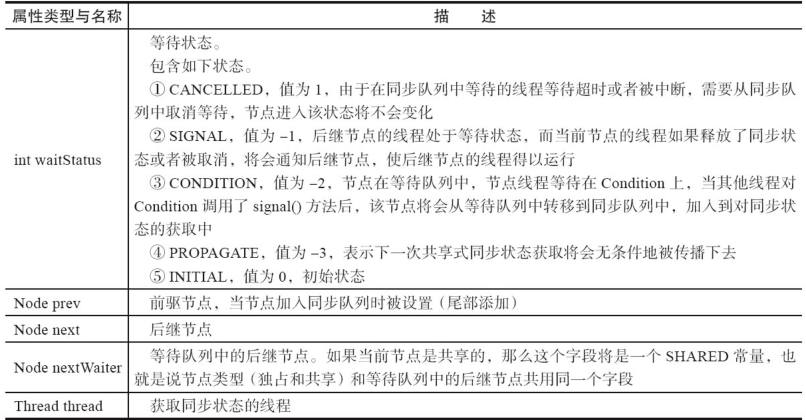
从实现角度分析同步器是如何完成线程同步的，主要包括：同步队列、独占式同步状态获取与释放、共享式同步状态获取与释放以及超时获取同步状态等同步器的核心数据结构与模板方法。

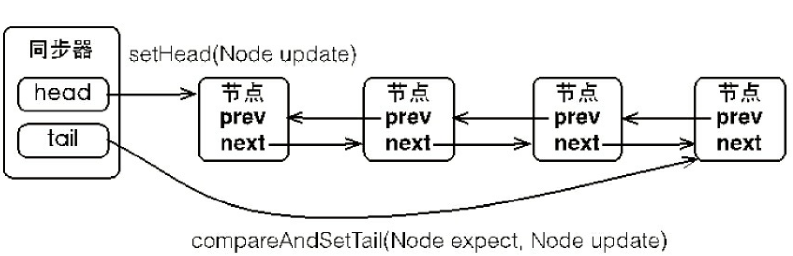
**2.1.1 同步队列**

同步器依赖内部的**同步队列（一个FIFO双向队列）**来完成同步状态的管理，当前线程获取同步状态**失败**时，同步器会将**当前线程**以及**等待状态等信息**构造成为一个节点（Node）并将其加入**同步队列**，同时会**阻塞当前线程**，当**同步状态释放**时，会把**首节点中的线程唤醒**，使其**再次尝试获取同步状态。**

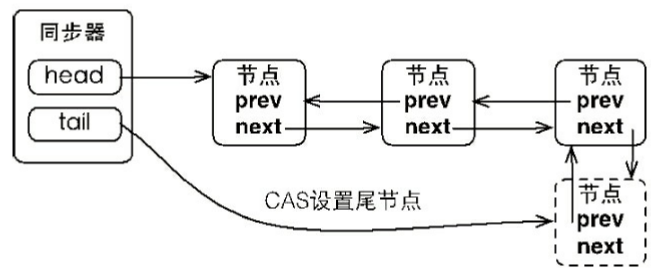
同步队列中的**节点（Node）**用来保存**获取同步状态失败的线程引用**、**等待状态**以及**前驱和**

**后继节点**。

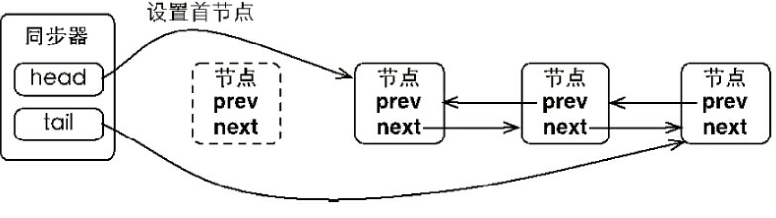




同步器包含了两个节点类型的引用，一个指向头节点，而另一个指向尾节点。试想一下，当一个线程成功地获取了同步状态（或者锁），**其他线程将无法获取到同步状态**，转而被构造成为**节点并加入到同步队列**中，而这个**加入队列的过程必须要保证线程安全**，因此同步器提供了一个**基于CAS的设置尾节点**的方法：**compareAndSetTail(Node expect,Node update)**，它需要传递当前线程**“认为”的尾节点和当前节点**，只有设置成功后，当前节点才正式与之前的尾节点建立关联。



同步队列遵循FIFO，首节点是获取同步状态成功的节点，首节点的线程在释放同步状态时，将会唤醒后继节点，而后继节点将会在获取同步状态成功时将自己设置为首节点，

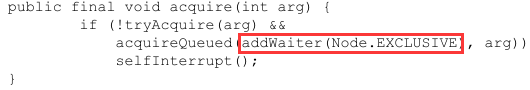


注意：只有在获取同步状态时，才会对首节点进行设置，假设这么一条队列A-B-C，当其他线程释放锁之后，同步队列首节点A先获取同步状态，然后在同步状态过程中将B设为新的首节点。因为在同步状态中实现的，所以不许要CAS保证。

2.1.2.**独占式同步状态获取与释放**

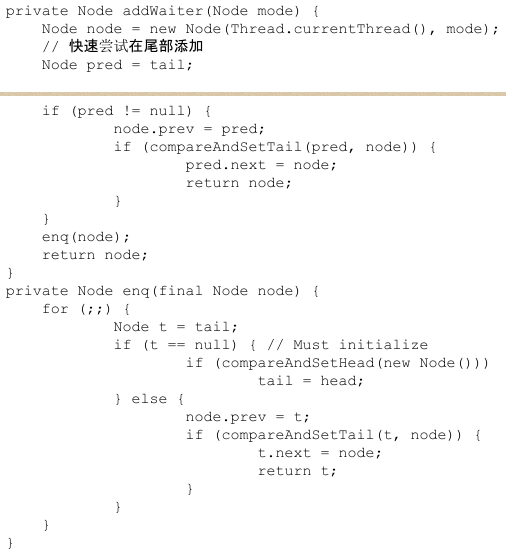
通过调用同步器的acquire(int arg)方法**可以获取同步状态**，该方法**对中断不敏感**，也就是由于**线程获取同步状态失败后进入同步队列中**，**后续对线程进行中断操作时**，线程不会从同

步队列中移出



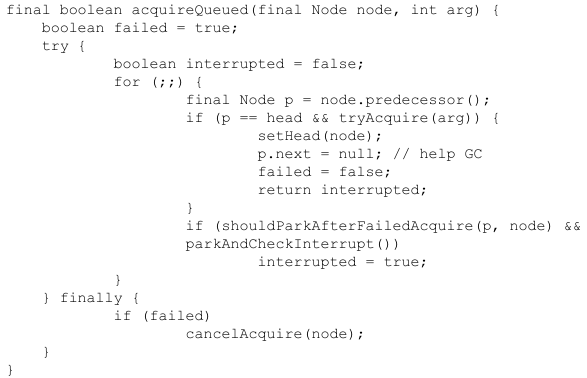
首先调用自定义同步器实现的tryAcquire(int arg)方法，该方法保证线程安全的获取同步状态，如果**同步状态获取失败**，则构造同步节点（独占式Node.EXCLUSIVE，**同一时刻只能有一个线程成功获取同步状态**）并通过addWaiter(Node node)方法将该**节点加入到同步队列的尾部**，最后调用acquireQueued(Node node,int arg)方法，使得该节点**以“死循环”的方式获取同步状态**。如果**获取不到**则**阻塞**节点中的线程，而**被阻塞线程的唤醒**主要依靠**前驱节点的出队**或**阻塞线程被中断**来实现。

节点的构造以及加入同步队列



通过使用compareAndSetTail(Node expect,Node update)方法来确保节点能够被**线程安全**添加。如果不符合上一种情况，就使用**enq(final Node node)**方法，同步器通过**“死循环”**来保证**节点的正确添加**，在**“死循环”**中只有通过**CAS将节点设置成为尾节点**之后，当前线程才能从该方法**返回**，否则，当前线程**不断地尝试设置**。

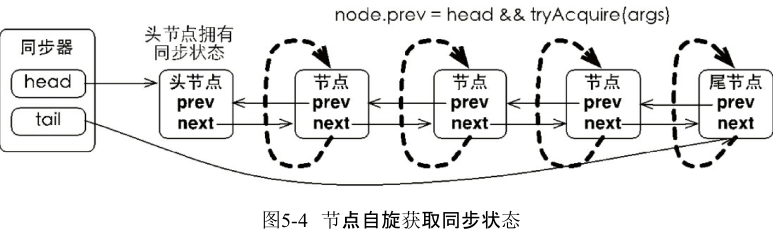
节点进入同步队列之后，就进入了一个**自旋**的过程，当条件满足，**获取到了同步状态**，就可以从这个**自旋过程中退出**，否则依旧留在这个**自旋过程中**（并会**阻塞节点的线程**）



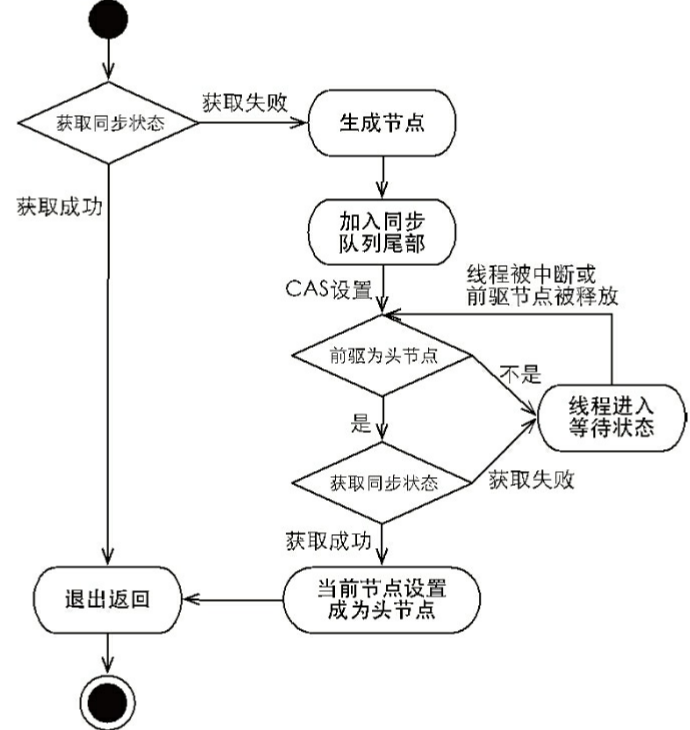
在acquireQueued(final Node node,int arg)方法中，当前线程在“死循环”中尝试获取同步状态，而**只有前驱节点是头节点**才能够尝试**获取同步状态**，这是为什么？

①头结点是成功获得同步状态的节点，而头结点的线程释放了同步状态后，将会唤醒后继节点，后继节点的线程被唤醒后，需要检查自己的前驱节点是否为头节点。

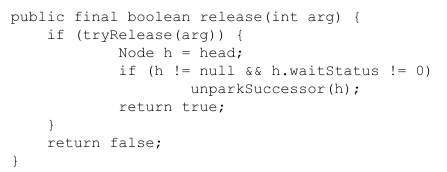
②维护同步队列的FIFO原则。



可以看到节点和节点之间在循环检查的过程中基本**不相互通信**，而是简单地判断**自己的前驱是否为头节点**，这样就使得节点的释放规则符合FIFO



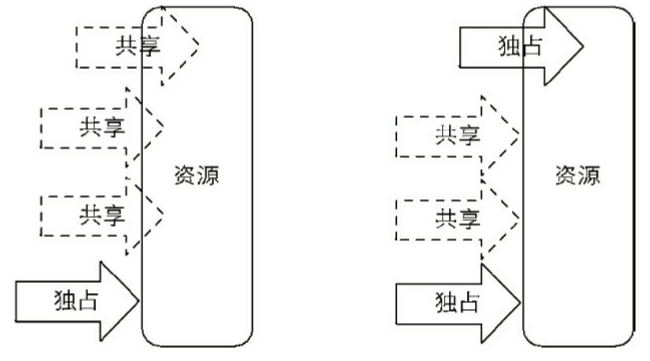
当前线程获取同步状态并执行了相应逻辑之后，就需要**释放同步状态**，使得后续节点能够继续获取同步状态。通过调用**同步器的release(int arg)**方法可以**释放同步状态**，该方法在释放了同步状态之后，会唤醒其后继节点（进而使**后继节点重新尝试获取同步状态，从等待状态到继续自旋**）。该方法代码如代码清单5-6所示。



该方法执行时，如果成功释放了同步状态，就会唤醒头结点的后继节点线程，unparkSuccessor（Node node）方法使用LockSupport来唤醒处于等待的线程。

在获取同步状态时，**同步器**维护**一个同步队列**，**获取状态失败的线程**都会被加入到队列中并在队列中进行**自旋**；**移出队列**（或停止自旋）的条件是**前驱节点为头节点且成功获取了同步状态**。在**释放同步状态**时，同步器调用tryRelease(int arg)方法**释放同步状态**，然后唤醒头节点的**后继节点**。

**2.1.3 共享式同步状态获取与释放**



左：共享式访问资源时，其他共享式的访问军备允许，而独占式访问被阻塞

右：独占式访问资源时，同一时刻其他访问均被阻塞

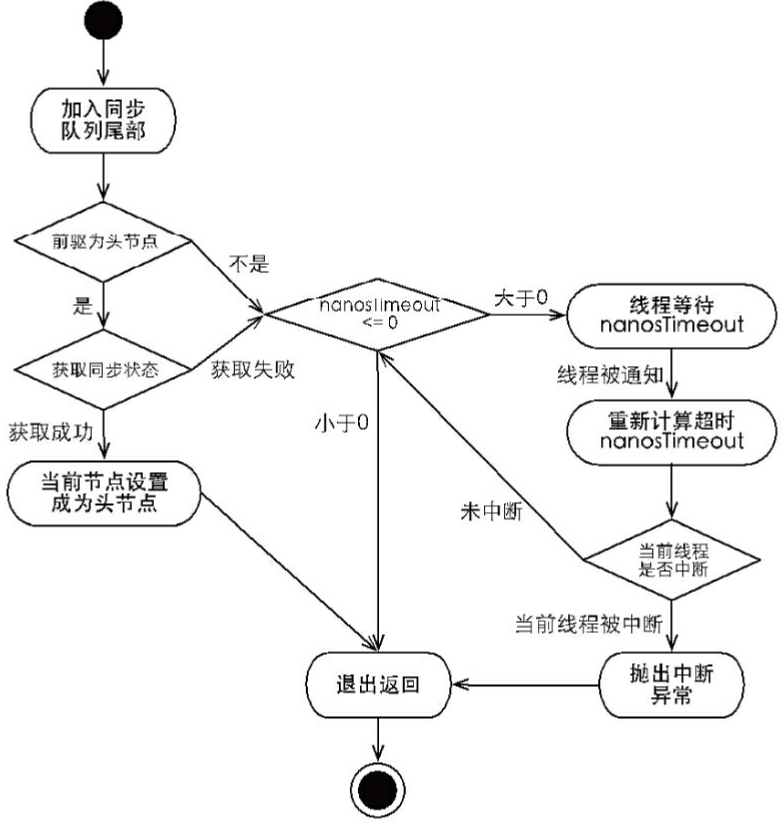
通过调用acquireShared(int arg)方法可以共享式的获取同步状态

|  |
| --- |
| **public** **final** **void** acquireShared(**int** arg) {  **if** (tryAcquireShared(arg) < 0)  doAcquireShared(arg);  }  同步器调用tryAcquireShared方法尝试获取同步状态，该方法返回值为int类型，如果返回大于等于0，表示能够获取同步状态。因此在共享式获取的自旋过程中，成功获取到同步状态并退出自旋的条件是tryacquireshared方法返回值大于0.  ---------------------------------------------------------------  **private** **void** doAcquireShared(**int** arg) {  **final** Node node = addWaiter(Node.***SHARED***);  **boolean** failed = **true**;  **try** {  **boolean** interrupted = **false**;  **for** (;;) {  **final** Node p = node.predecessor();  **if** (p == head) {  **int** r = tryAcquireShared(arg);  **if** (r >= 0) {  setHeadAndPropagate(node, r);  p.next = **null**; // help GC  **if** (interrupted)  *selfInterrupt*();  failed = **false**;  **return**;  }  }  **if** (*shouldParkAfterFailedAcquire*(p, node) &&  parkAndCheckInterrupt())  interrupted = **true**;  }  } **finally** {  **if** (failed)  cancelAcquire(node);  }  }  如果当前节点的前驱为头结点，尝试获取同步状态，调用tryacquireshared方法，继续自旋，直到返回值>=0,表示该次获取同步状态成功并从自旋状态退出。  共享式释放同步状态，通过releaseShared(int args)方法可以释放同步状态。  **public** **final** **boolean** releaseShared(**int** arg) {  **if** (tryReleaseShared(arg)) {  doReleaseShared();  **return** **true**;  }  **return** **false**;  }  使用tryreleaseshared尝试将状态设置为共享式同步释放的状态，然后使用doReleaseShared（）确保同步状态线程  -----------------------------------------------------------  线程同步状态安全释放后，进行**循环操作**，得到当前头结点的waitStatus，如果是-1， 表明后继线程需要唤醒了，使用**cas**将h的waitStatus置为0，然后将唤醒后面的线程。  **private** **void** doReleaseShared() {  **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**;  }  } |

**2.1.4 独占式超时获取同步状态**

通过调用同步器的**doAcquireNanos(int arg,long nanosTimeout)**方法可以超时获取同步状态，即在指定的时间段内获取同步状态，如果获取到同步状态则返回true，否则，返回false。

|  |
| --- |
| 设定最后期限为当前时间+nanosTimeout，一直自旋，如果前一个线程p是头结点，就尝试获取同步状态，获取成功，将当前设置为独占状态的线程的节点node设置为头结点，，将线程p的next置空，有助于gc，返回true表示获取同步状态，否则计算当前的nanosTimeout = deadline – 系统当前时间，如果小于等于0，说明超时了，如果线程发生中断你，直接抛出异常  **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)) {  setHead(node);  p.next = **null**; // help GC  failed = **false**;  **return** **true**;  }  nanosTimeout = deadline - System.*nanoTime*();  如果p！=head或者tryacquire失败了，*shouldParkAfterFailedAcquire*(p, node)检查前驱p和node的状态，如果前驱的waitStatus==-1，表明p已经准备好去唤醒下一节点即node，如果大于0，说明前驱p已经不是头结点了，需要找到新的前驱，waitStatus为小于等于0的，如果等于0，采用cas将其设定为signal表示可以唤醒下一个线程；如果nanosTimeout>快速自旋的阈值，就使用LockSupport.*parkNanos进行等待nanosTimeout秒；然后从该方法返回。如果小于等于快速自旋那么就进入快速自旋、*  **if** (nanosTimeout <= 0L)  **return** **false**;  **if** (*shouldParkAfterFailedAcquire*(p, node) &&  nanosTimeout > ***spinForTimeoutThreshold***)  LockSupport.*parkNanos*(**this**, nanosTimeout);  **if** (Thread.*interrupted*())  **throw** **new** InterruptedException();  }  } **finally** {  **if** (failed)  cancelAcquire(node);  }  } |



独占式超时获取同步状态doAcquireNanos(int arg,long nanosTimeout)和独占式获取同步状态acquire(int args)在流程上非常相似，其主要区别在于未获取到同步状态时的处理逻辑。acquire(int args)在未获取到同步状态时，**将会使当前线程一直处于等待状态**，而doAcquireNanos(int arg,long nanosTimeout)会使当前线程**等待nanosTimeout纳秒**，如果当前线程在nanosTimeout纳秒内没有获取到同步状态，将会从**等待逻辑中自动返回**。

**2.1.5 自定义同步组件**

设计同步工具：该工具在同一时刻，只允许至多两个线程同时访问，超过两个线程的访问将被阻塞，我们将这个同步工具命名为TwinsLock。

**首先，确定访问模式**。TwinsLock能够在同一时刻支持多个线程的访问，这显然是**共享式访问**，**因此，需要使用同步器提供的acquireShared(int args)方法等和Shared相关的方法**，这就要求**TwinsLock必须重写tryAcquireShared(int args)方法和tryReleaseShared(int args)**方法，这样才能保证同步器的共享式同步状态的获取与释放方法得以执行。

**其次，定义资源数。**TwinsLock在同一时刻允许至多两个线程的同时访问，表明**同步资源数为2**，这样可以设置**初始状态status为2**，当一个线程进行获取，**status减1**，该线程释放，则**status加1**，状态的合法范围为0、1和2，其中0表示当前已经有两个线程获取了同步资源，此时再有其他线程对同步状态进行获取，该线程只能被阻塞。在同步状态变更时，需要使用**compareAndSet(int expect,int update)方法做原子性保障**。

**最后，组合自定义同步器**。前面的章节提到，**自定义同步组件通过组合自定义同步器**来完成同步功能，**一般情况下自定义同步器会被定义为自定义同步组件的内部类**。

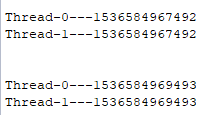
|  |
| --- |
| public class TwinsLock implements Lock {  private final Sync sync = new Sync(2);  private static final class Sync extends AbstractQueuedSynchronizer {  Sync(int count) {  if (count <= 0) {  throw new IllegalArgumentException("count must largethan zero.");  }  setState(count);  }  public int tryAcquireShared(int reduceCount) {  for (;;) {  int current = getState();  int newCount = current - reduceCount;  if (newCount < 0 || compareAndSetState(current, newCount)) {  return newCount;  }  }  }  public boolean tryReleaseShared(int returnCount) {  for (;;) {  int current = getState();  int newCount = current + returnCount;  if (compareAndSetState(current, newCount)) {  return true;  }  }  }  }  public void lock() {  sync.acquireShared(1);  }  public void unlock() {  sync.releaseShared(1);  }  // 其他接口方法略  } |

实现了lock接口，提供了面向使用者的接口，lock和unlock，使用者调用lock方法获取锁，随后用unlock方法释放锁，同一时刻只能有两个线程同时获取到锁。TwinsLock包含一个自定义同步器Sync，以共享式获取同步状态为例，通过**获取当前state值，减去reducecount得到同步后的状态值，然后使用cas对状态值进行更新，**当tryAcquireShared方法返回值大于等于0时，当前**线程才获取同步状态**，即获得了锁。释放锁也是同理。

**设计一个测试方法**：

|  |
| --- |
| **public** **class** TwinsLockTest {  @Test  **public** **void** test() {  **final** Lock lock = **new** TwinsLock();  **class** Worker **extends** Thread {  **public** **void** run() {  **while** (**true**) {  lock.lock();  **try** {  SleepUtils.*second*(1);  System.***out***.println(Thread.*currentThread*().getName() + "---" +System.*currentTimeMillis*());  SleepUtils.*second*(1);  } **finally** {  lock.unlock();  }  }  }  }  // 启动10个线程  **for** (**int** i = 0; i < 10; i++) {  Worker w = **new** Worker();  w.setDaemon(**true**);  w.start();  }  // 每隔1秒换行  **for** (**int** i = 0; i < 10; i++) {  SleepUtils.*second*(1);  System.***out***.println();  }  }  } |

定义一个work线程，然后设定为守护线程，启动10个work线程，让主函数执行10秒钟，为了防止该test方法过早结束；然后在try块前进行加锁finally进行解锁，保证线程成对输出。



**3.重入锁**

它表示该锁能够支持**同一个线程**对资源的重复加锁。除此之外，该锁支持获取锁时的公平和非公平性选择。

ReentrantLock在调用lock方法时，已经获取到锁的线程，能够再次调用lock方法获取锁而不阻塞。

公平锁：先对锁进行获取的请求一定先被满足，那么这个锁是公平的。

非公平锁：获取锁的线程是随机的。

**3.1实现可重入**

重进入：任意线程在获取到锁之后，能够再次获取该锁而不被锁阻塞。

需要解决两个问题：

①线程再次获取锁。需要识别获取锁的线程是否为当前占据锁的线程，如果是，则再次成功获取。

②锁的最终释放。重复获得n次锁，随后在n次释放该锁后，其他线程能够获取该锁。锁的最终释放要求锁对于获取次数进行记录，计数表示当前锁被重复获取的次数，而锁被释放时，计数自减，当计数为0时，表示成功释放。

ReentrantLock的nonfairTryAcquire方法

|  |
| --- |
| **final** **boolean** nonfairTryAcquire(**int** acquires) {  **final** Thread current = Thread.*currentThread*();  **int** c = getState();  **if** (c == 0) {  **if** (compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  **return** **true**;  }  }  **else** **if** (current == getExclusiveOwnerThread()) {  **int** nextc = c + acquires;  **if** (nextc < 0) // overflow  **throw** **new** Error("Maximum lock count exceeded");  setState(nextc);  **return** **true**;  }  **return** **false**;  } |

首先获取state的值，如果为0，说明当前锁未被线程占用，使用cas修改state的值，然后让当前线程作为占有锁的线程。如果state值不为0，那么判断当前线程是否是占有锁的线程，如果是，设置state的值。

ReentrantLock的tryRelease方法

|  |
| --- |
| **protected** **final** **boolean** tryRelease(**int** releases) {  **int** c = getState() - releases;  **if** (Thread.*currentThread*() != getExclusiveOwnerThread())  **throw** **new** IllegalMonitorStateException();  **boolean** free = **false**;  **if** (c == 0) {  free = **true**;  setExclusiveOwnerThread(**null**);  }  setState(c);  **return** free;  } |

获取state值-release，判断当前线程是否是占有锁的线程，如果不是抛出异常，否则判断c==0，如果是释放，设置当前拥有锁的线程为空。

**3.2 公平锁与非公平锁的区别**

非公平锁，只要CAS设置同步状态成功，就可以获取锁，公平锁则不同

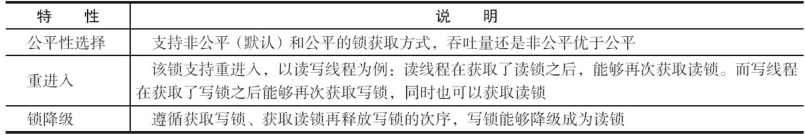
|  |
| --- |
| **protected** **final** **boolean** tryAcquire(**int** acquires) {  **final** Thread current = Thread.*currentThread*();  **int** c = getState();  **if** (c == 0) {  **if** (!hasQueuedPredecessors() &&  compareAndSetState(0, acquires)) {  setExclusiveOwnerThread(current);  **return** **true**;  }  }  **else** **if** (current == getExclusiveOwnerThread()) {  **int** nextc = c + acquires;  **if** (nextc < 0)  **throw** **new** Error("Maximum lock count exceeded");  setState(nextc);  **return** **true**;  }  **return** **false**;  }  } |

获取state的值，如果state为0，表明当前锁未被占用，判断该线程是否有前驱节点，如果有则需要等前驱节点获取并释放锁才能继续获取锁，如果没有，并且state重新设置成功，就将当前线程设置为拥有锁的线程，返回true。如果state不为0，如果当前线程就是占有锁的线程，设置state，返回true。

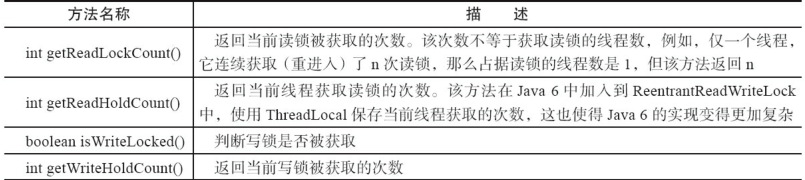
公平性锁保证了锁的获取按照FIFO原则，而代价是进行大量的线程切换。非公平性锁虽然可以造成线程饥饿，但是极少的线程切换，保证了其更大的吞吐量。

**4.读写锁**

同一时刻可以允许多个读线程访问，但是在写线程访问时，所有的读线程和其他写线程均被阻塞。读写锁维护了一对锁，一个读锁一个写锁，通过分离读锁和写锁，使得并发性相比一般排它锁有了较大提升。



**4.1读写锁的接口与示例（ReentrantReadWriteLock）**

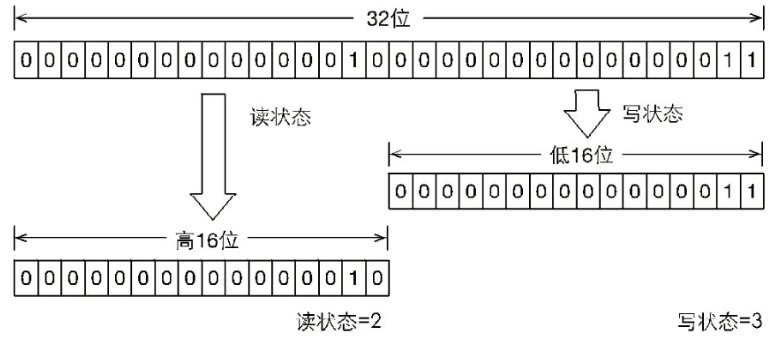


**4.2 读写锁的实现分析**

**4.2.1 读写状态的设计**

依赖自定义的同步器来实现同步，读写状态就是同步器的同步状态。同步器需要在同步状态（一个整形变量）上维护多个读线程和一个写线程的状态，使得该状态的设计成为读写实现的关键。

如果一个整型变量上维护多种状态，就一定需要“按位切割使用”这个变量，读写锁将变量切分成了两个部分，高16位表示读，低16位表示写，划分方式如图5-8所示。



答案是通过位运算。假设当前同步状态值为S，写状态等于S&0x0000FFFF（将高16位全部抹去），读状态等于S>>>16（无符号补0右移16位）。当写状态增加1时，等于S+1，当读状态增加1时，等于S+(1<<16)，也就是S+0x00010000。

**S不等于0时，当写状态（S&0x0000FFFF）等于0时，则读状态（S>>>16）大于0，即读锁已被获取。**

**4.2.2 写锁的获取与释放**

写锁是一个支持**重进入**的排它锁。如果当前线程已经获取了**写锁**，则**增加写状态**。如果当前线程在获取写锁时，读锁已经被获取（读状态不为0）或者该线程不是已经获取写锁的线程，则当前线程进入等待状态

|  |
| --- |
| **protected** **final** **boolean** tryAcquire(**int** acquires) {  Thread current = Thread.*currentThread*();  **int** c = getState();  **int** w = *exclusiveCount*(c);  **if** (c != 0) {  **if** (w == 0 || current != getExclusiveOwnerThread())  **return** **false**;  **if** (w + *exclusiveCount*(acquires) > ***MAX\_COUNT***)  **throw** **new** Error("Maximum lock count exceeded");  // Reentrant acquire  setState(c + acquires);  **return** **true**;  }  **if** (writerShouldBlock() ||  !compareAndSetState(c, c + acquires))  **return** **false**;  setExclusiveOwnerThread(current);  **return** **true**;  } |

得到当前线程，，获取状态值，获取独占值，判断写锁的状态值，c&0x0000FFFF；如果c不为0，w==0表明存在读锁，阻塞；如果c不为0，w不为0，且当前线程不是占有锁的线程，阻塞；如果w+acquires>MAX-count 即0x0000FFFF，抛出异常；否则的话说明当前线程占有写锁，进入重入状态，设置状态值；判断是否需要阻塞，根据公平锁与非公平锁的方法判断，使用cas设置状态值，然后将当前线程设置为独占线程。

**写锁的释放**

|  |
| --- |
| **protected** **final** **boolean** tryRelease(**int** releases) {  **if** (!isHeldExclusively())  **throw** **new** IllegalMonitorStateException();  **int** nextc = getState() - releases;  **boolean** free = *exclusiveCount*(nextc) == 0;  **if** (free)  setExclusiveOwnerThread(**null**);  setState(nextc);  **return** free;  } |

**4.2.3 读锁的获取与释放**

读锁是一个支持**重进入的共享锁**，它能够被多个线程同时获取，在没有其他写线程访问（或者写状态为0）时，读锁总会被成功地获取，而所做的也只是**（线程安全的）增加读状态**。如果当前线程已经获取了读锁，**则增加读状态**。如果当前线程在获取读锁时，写锁已被其他线程获取，则进入**等待状态**。

|  |
| --- |
| **protected** **final** **int** tryAcquireShared(**int** unused) {  Thread current = Thread.*currentThread*();  **int** c = getState();  **if** (*exclusiveCount*(c) != 0 &&  getExclusiveOwnerThread() != current)  **return** -1;  **int** r = *sharedCount*(c);  **if** (!readerShouldBlock() &&  r < ***MAX\_COUNT*** &&  compareAndSetState(c, c + ***SHARED\_UNIT***)) {  **if** (r == 0) {  firstReader = current;  firstReaderHoldCount = 1;  } **else** **if** (firstReader == current) {  firstReaderHoldCount++;  } **else** {  HoldCounter rh = cachedHoldCounter;  **if** (rh == **null** || rh.tid != *getThreadId*(current))  cachedHoldCounter = rh = readHolds.get();  **else** **if** (rh.count == 0)  readHolds.set(rh);  rh.count++;  }  **return** 1;  }  **return** fullTryAcquireShared(current);  } |

获取当前线程和state值，判断独占写锁为0，且当前线程就是拥有锁的线程，然后计算读锁的重入值r，如果读线程不需要阻塞，且重入值r小于0x0000FFFF，并且通过CAS将state更新成功，然后对r进行判断，如果等于0.说明第一个读线程，firstReader设置为current，然后firstReaderHoldCount赋值为1，如果firstReader=current，说明当前线程重复获取锁，将firstReaderHoldCount++；返回fullTryAcquireShared（current）

|  |
| --- |
| **final** **int** fullTryAcquireShared(Thread current) {  HoldCounter rh = **null**;  **for** (;;) {  **int** c = getState();  **if** (*exclusiveCount*(c) != 0) {  **if** (getExclusiveOwnerThread() != current)  **return** -1;  } **else** **if** (readerShouldBlock()) {  **if** (firstReader == current) {  } **else** {  **if** (rh == **null**) {  rh = cachedHoldCounter;  **if** (rh == **null** || rh.tid != *getThreadId*(current)) {  rh = readHolds.get();  **if** (rh.count == 0)  readHolds.remove();  }  }  **if** (rh.count == 0)  **return** -1;  }  }  **if** (*sharedCount*(c) == ***MAX\_COUNT***)  **throw** **new** Error("Maximum lock count exceeded");  **if** (compareAndSetState(c, c + ***SHARED\_UNIT***)) {  **if** (*sharedCount*(c) == 0) {  firstReader = current;  firstReaderHoldCount = 1;  } **else** **if** (firstReader == current) {  firstReaderHoldCount++;  } **else** {  **if** (rh == **null**)  rh = cachedHoldCounter;  **if** (rh == **null** || rh.tid != *getThreadId*(current))  rh = readHolds.get();  **else** **if** (rh.count == 0)  readHolds.set(rh);  rh.count++;  cachedHoldCounter = rh; // cache for release  }  **return** 1;  }  }  } |

在tryAcquireShared(int unused)方法中，如果其他线程已经获取了写锁，则当前线程获取读锁失败，进入等待状态。如果当前线程获取了写锁或者写锁未被获取，则当前线程（线程安全依靠CAS保证）增加读状态，成功获取读锁。

读锁的每次释放（线程安全的，可能有多个读线程同时释放读锁）均减少读状态，减少的

值是（1<<16）。

**4.2.4 锁降级**

锁降级指的是写锁降级为读锁，线程持有写锁，先获取到读锁，随后释放（先前拥有的）写锁的过程。

|  |
| --- |
| **public** **void** processData() {  readLock.lock();  **if** (!update) {  // 必须先释放读锁  readLock.unlock();  // 锁降级从写锁获取到开始  writeLock.lock();  **try** {  **if** (!update) {  // 准备数据的流程（略）  update = **true**;  }  readLock.lock();----①  } **finally** {  writeLock.unlock();  }  // 锁降级完成，写锁降级为读锁  }  **try** {  // 使用数据的流程（略）  此处会有使用update的代码----②  } **finally** {  readLock.unlock();  }  } |

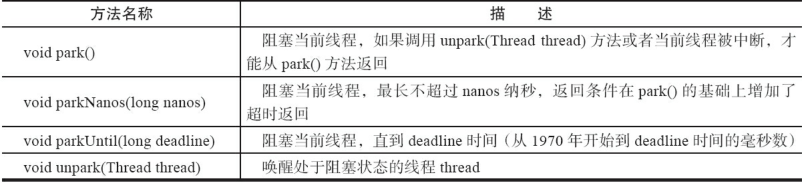
为什么锁降级释放写锁之前，要先获取读锁？

设当前线程为a，如果当前不获取读锁，即①代码不存在，直接释放写锁，当执行到代码②的时候，需要使用update的数据，此时线程b对update重新获取写锁并进行修改，那么代码②无法看到更新后的update数据，这样会使数据不可见。如果在释放写锁前，加入获取读锁代码①，那么其他线程只能读update数据，不能更新update数据。

**4.5 LockSupport工具**

LockSupport定义了一组的公共静态方法，这些方法提供了最基本的**线程阻塞**和**唤醒**功能，而LockSupport也成为构建同步组件的基础工具。

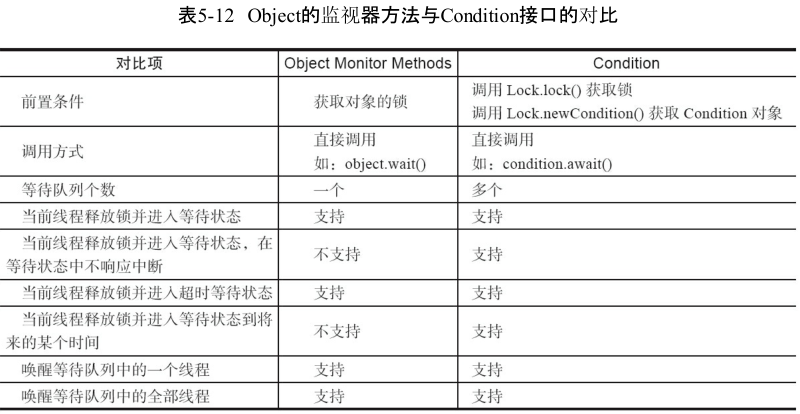
LockSupport定义了一组以**park开头**的方法用来**阻塞当**前线程，以及**unpark**(Thread thread)方法来**唤醒一**个被阻塞的线程。



在Java 6中，LockSupport增加了park(Object blocker)、parkNanos(Object blocker,long nanos)和parkUntil(Object blocker,long deadline)3个方法，用于实现**阻塞当前线程**的功能，其中参数**blocker**是用来标识**当前线程在等待的对象**（以下称为**阻塞对象**），该对象主要用于**问题排查和系统监控**。

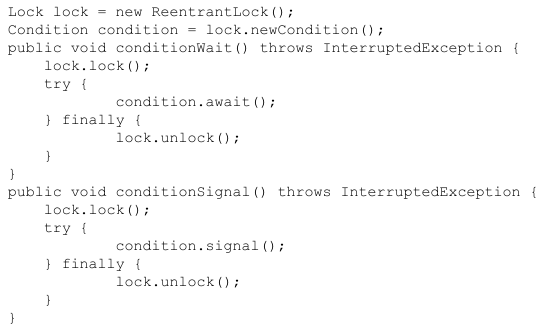
**4.6Condition接口**

Condition接口也提供了类似Object的监视器方法，与**Lock**配合可以实现**等待/通知**模式，但是这两者在使用方式以及功能特性上还是有差别的。

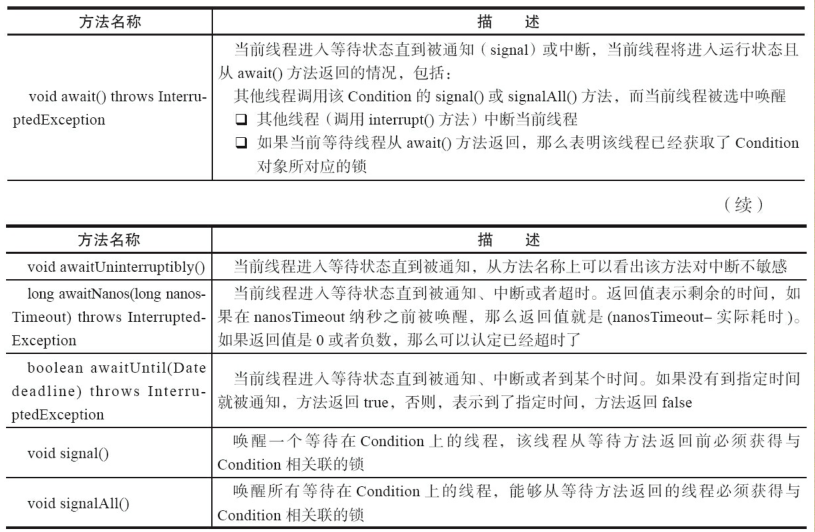


4.6.1Condition接口与示例

Condition定义了等待/通知两种类型的方法，当前线程调用这些方法时，需要提前获取到**Condition对象关联的锁**。Condition对象是由Lock对象（调用Lock对象的newCondition()方法）创建出来的，换句话说，Condition是**依赖Lock**对象的。



当调用await()方法后，当前线程会释放锁并在此等待，而其他线程调用Condition对象的signal()方法，通知当前线程后，当前线程才从await()方法返回，并且在返回前已经获取了锁。



|  |
| --- |
| **package** cn.huangwei.fifth;  **import** java.util.concurrent.locks.Condition;  **import** java.util.concurrent.locks.Lock;  **import** java.util.concurrent.locks.ReentrantLock;  **public** **class** BoundedQueue<T> {  **private** Object[] items;  // 添加的下标，删除的下标和数组当前数量  **private** **int** addIndex, removeIndex, count;  **private** Lock lock = **new** ReentrantLock();  **private** Condition notEmpty = lock.newCondition();  **private** Condition notFull = lock.newCondition();  **public** BoundedQueue(**int** size) {  items = **new** Object[size];  }  // 添加一个元素，如果数组满，则添加线程进入等待状态，直到有"空位"  **public** **void** add(T t) **throws** InterruptedException {  lock.lock();  **try** {  **while** (count == items.length)  notFull.await();  items[addIndex] = t;  **if** (++addIndex == items.length)  addIndex = 0;  ++count;  notEmpty.signal();  } **finally** {  lock.unlock();  }  }  // 由头部删除一个元素，如果数组空，则删除线程进入等待状态，直到有新添加元素  @SuppressWarnings("unchecked")  **public** T remove() **throws** InterruptedException {  lock.lock();  **try** {  **while** (count == 0)  notEmpty.await();  Object x = items[removeIndex];  **if** (++removeIndex == items.length)  removeIndex = 0;  --count;  notFull.signal();  **return** (T) x;  } **finally** {  lock.unlock();  }  }  } |

首先需要获得锁，目的是确保数组修改的可见性和排他性。当数组数量等于数组长度时，表示数组已满，则调用notFull.await()，当前线程随之释放锁并进入等待状态。如果数组数量不

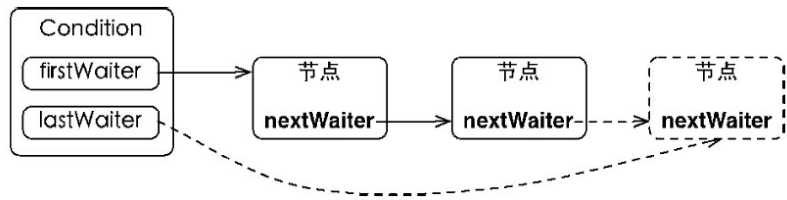
等于数组长度，表示数组未满，则添加元素到数组中，同时通知等待在notEmpty上的线程，数组中已经有新元素可以获取。

在添加和删除方法中使用while循环而非if判断，目的是防止过早或意外的通知，只有条件符合才能够退出循环。

**4.6.2 condition的实现分析**

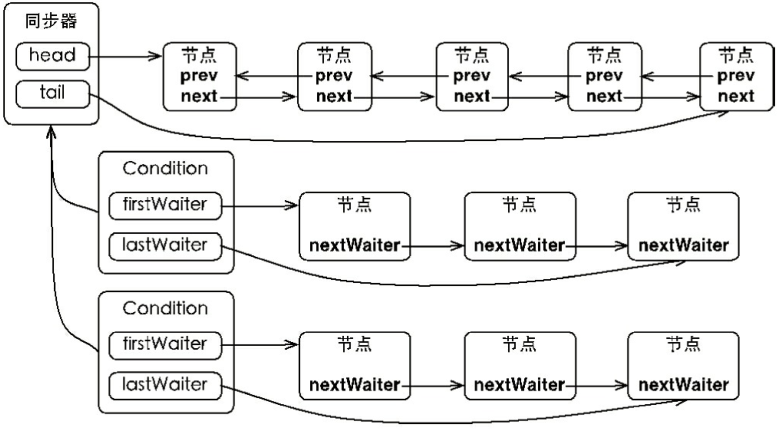
**①等待队列**

一个**FIFO**的队列，在队列中的每个节点都包含了一个**线程引用**，该线程就是在Condition对象上**等待的线程**，如果**一个线程**调用了Condition.await()方法，那么该线程将会**释放锁**、构造成**节点加入等待队列并进入等待状态**



Condition中包含了firstWaiter和lastWaiter成员，是AbstractQueuedSynchronizer.Node 类型，里面包含着一个成员thread保存等待的线程引用。调用awaiter方法后会将当前线程构造成Node，加入到等待队列尾部。**节点引用更新的过程**并没有使用**CAS保证**，原因在于调用await()方法的线程必定是**获取了锁的线程**，也就是说该过程是由**锁来保证线程安全**的。

在Object的监视器模型上，一**个对象拥有一个同步队列和等待队列**，而并发包中的Lock（更确切地说是同步器）拥有**一个同步队列和多个等待队列**

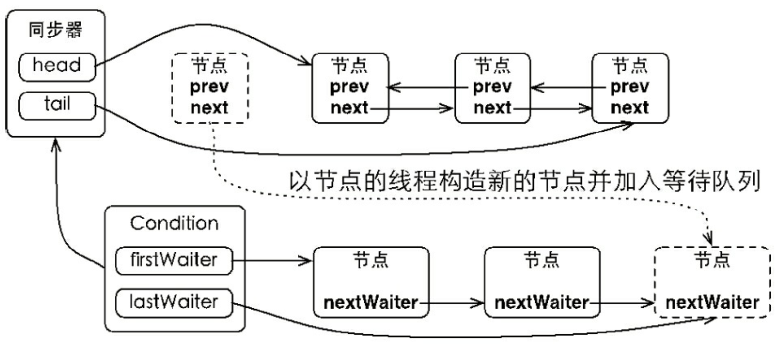


②等待状态

调用Condition的await()方法（或者以await开头的方法），会使当**前线程进入等待队列并释放锁**，同时线程状态变为**等待状态，内部调用LockSupport的park方法，进入等待（只有synchronized锁被占用的时候，会导致阻塞状态，Lock锁被占用的时候进入的是等待状态，因为LockSupport的相关方法）**。当从await()方法返回时，当前线程一定获取了Condition相关联的锁。

如果从队列（同步队列和等待队列）的角度看await()方法，当调用**await()**方法时，相当于同步队列的**首节点**（获取了锁的节点）移动到**Condition的等待队列**中。

|  |
| --- |
| **public** **final** **void** await() **throws** InterruptedException {  **if** (Thread.*interrupted*())  **throw** **new** InterruptedException();  Node node = addConditionWaiter();  **int** savedState = fullyRelease(node);  **int** interruptMode = 0;  **while** (!isOnSyncQueue(node)) {  LockSupport.*park*(**this**);  **if** ((interruptMode = checkInterruptWhileWaiting(node)) != 0)  **break**;  }  **if** (acquireQueued(node, savedState) && interruptMode != ***THROW\_IE***)  interruptMode = ***REINTERRUPT***;  **if** (node.nextWaiter != **null**) // clean up if cancelled  unlinkCancelledWaiters();  **if** (interruptMode != 0)  reportInterruptAfterWait(interruptMode);  }  -----------------------------------------------------------------  将当前线程设为wait线程，加到等待队列末尾，返回当前线程构成的节点。  **private** Node addConditionWaiter() {  Node t = lastWaiter;  **if** (t != **null** && t.waitStatus != Node.***CONDITION***) {  unlinkCancelledWaiters();  t = lastWaiter;  }  Node node = **new** Node(Thread.*currentThread*(), Node.***CONDITION***);  **if** (t == **null**)  firstWaiter = node;  **else**  t.nextWaiter = node;  lastWaiter = node;  **return** node;  }  ---------------------------------------------------------------------------  **final** **int** fullyRelease(Node node) {  **boolean** failed = **true**;  **try** {  **int** savedState = getState();  **if** (release(savedState)) {**release就是用来释放当前节点，唤醒后继节点**  failed = **false**;  **return** savedState;  } **else** {  **throw** **new** IllegalMonitorStateException();  }  } **finally** {  **if** (failed)  node.waitStatus = Node.***CANCELLED***;  }  } |



**4.6.3 通知**

Signal方法，将会唤醒在等待队列中等待时间最长的节点（首节点），在唤醒节点之前，会将节点移到同步队列中。

|  |
| --- |
| **public** **final** **void** signal() {  **if** (!isHeldExclusively())  **throw** **new** IllegalMonitorStateException();  Node first = firstWaiter;  **if** (first != **null**)  doSignal(first);  }  ------------------------------------------------------------------  判断时候有后继节点，如果有就将进行修改当前节点状态，并唤醒后继节点  **private** **void** doSignal(Node first) {  **do** {  **if** ( (firstWaiter = first.nextWaiter) == **null**)  lastWaiter = **null**;  first.nextWaiter = **null**;  } **while** (!transferForSignal(first) &&  (first = firstWaiter) != **null**);  }  ----------------------------------------------------------------  用于改变节点的状态，唤醒节点中的线程。  **final** **boolean** transferForSignal(Node node) {    **if** (!*compareAndSetWaitStatus*(node, Node.***CONDITION***, 0))  **return** **false**;  Node p = enq(node);//将等待队列中的节点，安全的移到同步队列  **int** ws = p.waitStatus;  **if** (ws > 0 || !*compareAndSetWaitStatus*(p, ws, Node.***SIGNAL***))  LockSupport.*unpark*(node.thread);  **return** **true**;  } |

signal()方法进行了**isHeldExclusively()**检查，也就是**当前线程**必须是**获取了锁的线程**。接着获取等待队列的首节点，将其移动到同步队列并使用LockSupport唤醒节点中的线程。