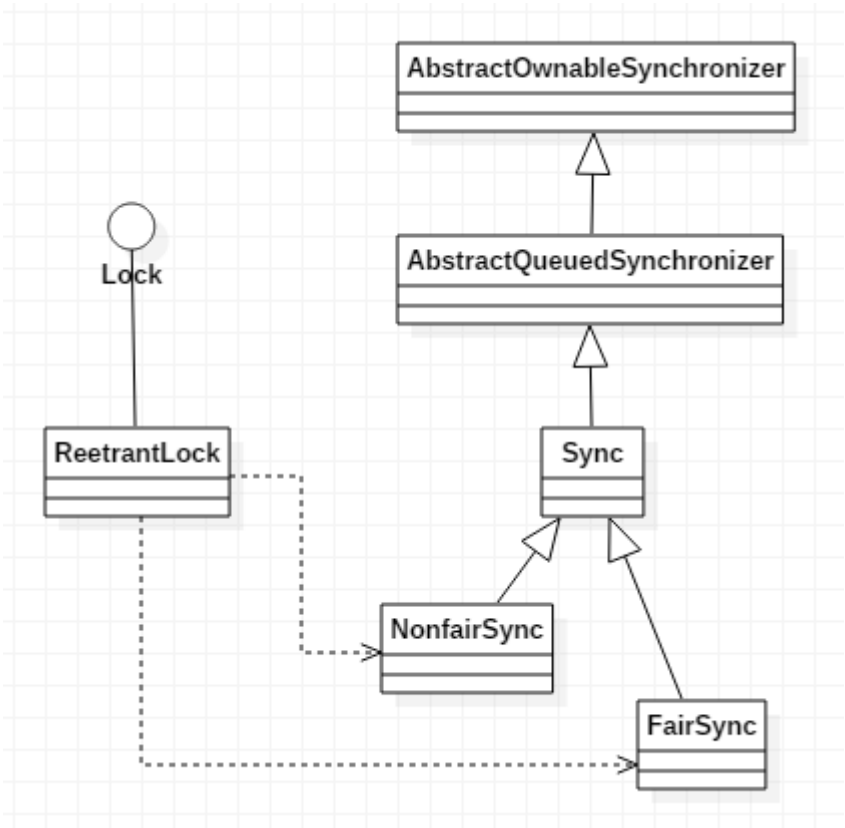
Concurrent包下各种类的实现机制

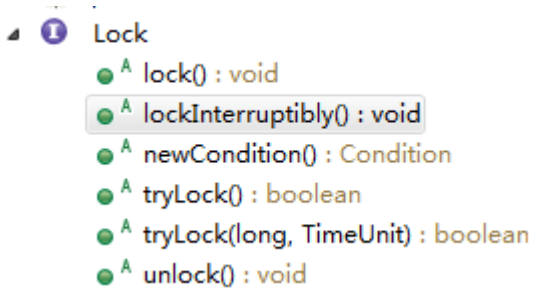
ReentrantLock

层次结构：



Lock定义了锁的接口规范。

ReenTrantLock实现了Lock的接口



AbstractQueueSynchronizer中以队列的形式实现线程间的同步

ReentrantLock的方法都依赖于AbstractQueueSynchronizer实现。

lock方法的实现：

进入lock方法，内部调用的是sync.lock()

```
public void lock() {
    sync.lock();
  }
```

sync是在ReentrantLock的构造函数中实现的。其中fair参数的不同可实现公平锁和费公平锁。由于在锁释放阶段，锁处于无线程占有状态，此时其他线程和队列中等待的线程都可以抢占该锁，从而出现公平锁和非公平锁的区别。

非公平锁：当锁处于无线程占有的状态，此时其他线程和在队列中等待的线程都可以抢占该锁。

公平锁：当锁处于无线程占有的状态，在其他线程抢占该锁的时候，都需要先进入队列中等待。

```
public ReentrantLock() {
    sync = new NonfairSync();
  }
public ReentrantLock(boolean fair) {
    sync = fair ? new FairSync() : new NonfairSync();
  }
```

NonfairSync继承自Sync，因此也继承了AbstractQueuedSynchronizer中的所有方法实现。接着进入NofairSync的lock方法。

```
final void lock() {
        // 利用cas置状态位，如果成功，则表示占有锁成功
        if (compareAndSetState(0, 1))
          // 记录当前线程为锁拥有者
                    setExclusiveOwnerThread(Thread.currentThread());
        else
          acquire(1);
    }
```

在lock方法中，利用CAS实现ReentrantLock位置置位。如果成功，则表示占有锁成功，并记录当前线程为锁拥有者。当占有锁失败，则调用acquire(1)方法继续处理。

```
public final void acquire(int arg) {
    //尝试获得锁，如果失败，则加入到队列中进行等待
    if (!tryAcquire(arg) &&
        acquireQueued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterrupt();
  }
```

acquire()是AbstractQueuedSynchronizer的方法。他首先会调用tryAcquire()去尝试获得锁，如果获得锁失败，则将当前线程加入到CLH队列中进行等待。tryAcquire()方法在NonFairSync中有实现，但是最终还是调用Sync中的nonfairTryAcquire()方法。

```
protected final boolean tryAcquire(int acquires) {
        return nonfairTryAcquire(acquires);
    }
```

```

```

```
final boolean nonfairTryAcquire(int acquires) {
        final Thread current = Thread.currentThread();
        // 获得状态
        int c = getState();
        // 如果状态为0，则表示该锁未被其他线程占有
        if (c == 0) {
            // 此时要再次利用cas去尝试占有锁
            if (compareAndSetState(0, acquires)) {
                // 标记当前线程为锁拥有者
                setExclusiveOwnerThread(current);
                return true;
            }
        }
        // 如果当前线程已经占有了，则state + 1,记录占有次数
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            // 此时无需利用cas去赋值，因为该锁肯定被当前线程占有
            setState(nextc);
            return true;
        }
        return false;
    }
```

在nonfairTryAcquire()中，首先会获得锁的状态，如果为0，则表示锁还未被其他线程占有，此时会利用CAS去尝试将锁的状态锁定，并标记当前线程为锁的拥有者；如果所的状态大于0，则会判断锁是否被当前线程占有，如果是，则State+1，这也是为什么lock()和unlock()的次数要对等；如果锁占有失效就返回false.

如果nonfairTryAcquire()返回false的情况下，就会调用acquireQueued(addWaiter(Node.EXCLUSICE),arg)方法，将当前线程加入队列中继续尝试获取锁。

上面是NonfairSync的实现，在fairSync中是采用另一种方法实现的

```
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;
    }
```

在fairSync中会首先判断队列中是否有线程等待，才会获取锁，这就是为什么是公平锁。

```
private Node addWaiter(Node mode) {
    //  创建当前线程的节点
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    // 如果尾节点不为空
    if (pred != null) {
        // 则将当前线程的节点加入到尾节点之后，成为新的尾节点
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }

    enq(node);
    return node;
}

private Node enq(final Node node) {

    // CAS方法有可能失败，因此要循环调用，直到当前线程的节点加入到队
```

```
    // CAS方法有可能失败，因此要循环调用，直到当前线程的节点加入到队列中
    for (;;) {
      Node t = tail;
      if (t == null) { // Must initialize
        Node h = new Node(); // Dummy header，头节点为虚拟节点
        h.next = node;
        node.prev = h;
          if (compareAndSetHead(h)) {
          tail = node;
          return h;
        }
      }
      else {
        node.prev = t;
        if (compareAndSetTail(t, node)) {
          t.next = node;
          return t;
        }
      }
    }
  }
```

addWaiter()是AbstactQueuedSynchronizer的方法，会以节点的形式来标记当前线程，并加入到尾节点中。enq()方法是在节点加入到尾节点失败的情况下，通过for(;;)循环反复调用cas方法，直到节点加入成功。由于enq()方法是非线程安全的，所以在增加节点的时候，需要使用cas设置head节点和tail节点。此时添加成功的结点状态为Node.EXCLUSIVE。

在节点加入到队列成功之后，会接着调用acquireQueued()方法去尝试获得锁。

```
final boolean acquireQueued(final Node node, int arg) {
    try {
      boolean interrupted = false;
      for (;;) {
        // 获得前一个节点
        final Node p = node.predecessor();
        // 如果前一个节点是头结点，那么直接去尝试获得锁
        // 因为其他线程有可能随时会释放锁，没必要Park等待
        if (p == head && tryAcquire(arg)) {
          setHead(node);
          p.next = null; // help GC
          return interrupted;
        }
        if (shouldParkAfterFailedAcquire(p, node) &&
          parkAndCheckInterrupt())

          interrupted = true;
```

```
                interrupted = true;
            }
        } catch (RuntimeException ex) {
            cancelAcquire(node);
            throw ex;
        }
    }
```

在acquireQueued()方法中，会利用for (;;)一直去获得锁，如果前一个节点为head节点，则表示可以直接尝试去获得锁了，因为占用锁的线程随时都有可能去释放锁并且该线程是被unpark唤醒的CLH队列中的第一个节点，获得锁成功后返回。

如果该线程的节点在CLH队列中比较靠后或者获得锁失败，即其他线程依然占用着锁，则会接着调用shouldParkAfterFailedAcquire()方法来阻塞当前线程，以让出CPU资源。在阻塞线程之前，会执行一些额外的操作以提高CLH队列的性能。由于队列中前面的节点有可能在等待过程中被取消掉了，因此当前线程的节点需要提前，并将前一个节点置状态位为SIGNAL，表示可以阻塞当前节点。因此该函数在判断到前一个节点为SIGNAL时，直接返回true即可。此处虽然存在对CLH队列的同步操作，但由于局部变量节点肯定是不一样的，所以对CLH队列操作是线程安全的。由于在compareAndSetWaitStatus(pred, ws, Node.SIGNAL)执行之前可能发生pred节点抢占锁成功或pred节点被取消掉，因此此处需要返回false以允许该节点可以抢占锁。

当shouldParkAfterFailedAcquire()返回true时，会进入parkAndCheckInterrupt()方法。parkAndCheckInterrupt()方法最终调用safe.park()阻塞该线程，以免该线程在等待过程中无线循环消耗cpu资源。至此，当前线程便被park了。那么线程何时被unpark，这将在unlock()方法中进行。

这里有一个小细节需要注意，在线程被唤醒之后，会调用Thread.interrupted()将线程中断状态置位为false，然后记录下中断状态并返回上层函数去抛出异常。我想这样设计的目的是为了可以让该线程可以完成抢占锁的操作，从而可以使当前节点称为CLH的虚拟头节点。

```
 private static boolean shouldParkAfterFailedAcquire(Node pred, Node node)
{
    int ws = pred.waitStatus;
    if (ws == Node.SIGNAL)
        /*
         * This node has already set status asking a release
         * to signal it, so it can safely park
         */
        return true;

    if (ws > 0) {
        // 如果前面的节点是CANCELLED状态，则一直提前
        do {

            node.prev = pred = pred.prev;
```

```
            node.prev = pred = pred.prev;
        } while (pred.waitStatus > 0);
        pred.next = node;
    } else {
        compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
    }
    return false;
}

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

public static void park(Object blocker) {
    Thread t = Thread.currentThread();
    setBlocker(t, blocker);
    unsafe.park(false, 0L);
    setBlocker(t, null);
}
```

2、unlock()方法的实现

同lock()方法，unlock()方法依然调用的是sync.release(1)。

```
public final boolean release(int arg) {
    // 释放锁
    if (tryRelease(arg)) {
        Node h = head;
        // 此处有个疑问，为什么需要判断h.waitStatus != 0
        if (h != null && h.waitStatus != 0)
            unparkSuccessor(h);
        return true;
    }
    return false;
}

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;
```

```
        return free;
    }
```

可以看到，tryRelease()方法实现了锁的释放，逻辑上即是将锁的状态置为0。当释放锁成功之后，通常情况下不需要唤醒队列中线程，因此队列中总是有一个线程处于活跃状态。

总结： ReentrantLock的锁资源以state状态描述，利用CAS则实现对锁资源的抢占，并通过一个CLH队列阻塞所有竞争线程，在后续则逐个唤醒等待中的竞争线程。ReentrantLock继承AQS完全从代码层面实现了java的同步机制，相对于synchronized，更容易实现对各类锁的扩展。同时，AbstractQueuedSynchronizer中的Condition配合ReentrantLock使用，实现了wait/notify的功能。

AbstractQueuedSynchronizer

同步器对于锁的实现主要通过三个方法来管理他的状态：

```
    java.util.concurrent.locks.AbstractQueuedSynchronizer.getState()
    java.util.concurrent.locks.AbstractQueuedSynchronizer.setState(int)

 java.util.concurrent.locks.AbstractQueuedSynchronizer.compareAndSetStat
e(int, int)
```

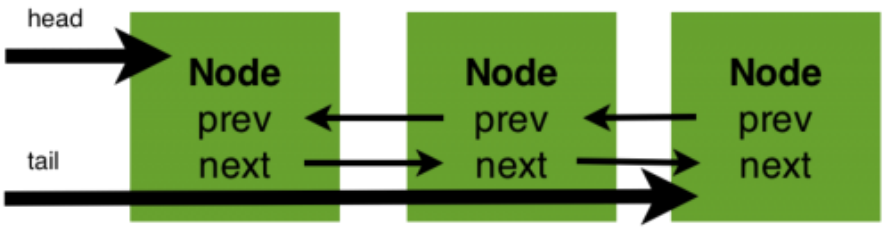
同步器开始依赖于一个FIFO队列，那么队列中的元素Node就是保存着线程引用和线程状态的容器，每个线程对同步器的访问，都可以看做是队列中的一个节点。Node主要包含以下成员变量：

```
Node{
    int waitStatus;
    Node prev;
    Node next;
    Node nextWaiter;
    Thread thread;
}
```

属性名称	描述
int waitStatus	CANCELLED，值为1，表示当前的线程被取消； SIGNAL，值为-1，表示当前节点的后继节点包含的线程需要运行，也就是unpark； CONDITION，值为-2，表示当前节点在等待condition，也就是在condition队列中； PROPAGATE，值为-3，表示当前场景下后续的acquireShared能够得以执行；值为0，表示当前节点在sync队列中，等待着获取锁。

属性名称	描述
Node prev	前驱节点，比如当前节点被取消，那就需要前驱节点和后继节点来完成连接。
Node next	后继节点。
Node nextWaiter	存储condition队列中的后继节点。
Thread thread	入队列时的当前线程。

节点成为sync队列和condition队列构建的基础，在同步器中就包含了sync队列。同步器拥有三个成员变量：sync队列的头结点head、sync队列的尾节点tail和状态state。对于锁的获取，请求形成节点，将其挂载在尾部，而锁资源的转移（释放再获取）是从头部开始向后进行。对于同步器维护的状态state，多个线程对其的获取将会产生一个链式的结构。



ConditionObject

显示锁可以与多个条件队列相关联，Condition是显示锁的条件队列，他是Object的wait/notify/notifyAll等方法的扩展。提供了在一个对象上设置多个等待集合的功能，即一个对象上设置多个等待条件。

Condition也称为条件队列，与内置锁关联的条件队列类似，他是一种广义的内置条件队列。他提供给线程一种方式使得该线程在调用wait方法后执行挂起操作，直到线程某个条件为真时被唤醒。条件队列必须跟锁一起使用，因为对共享状态变量的访问发生在多线程环境下，原理与内部条件队列一样。一个Codition的实例必须跟一个Lock绑定。因此，Condition一般作为Lock的内部类实现。