文档来源: https://blog.csdn.net/vernonzheng/article/details/8288251

**Java多线程（九）之ReentrantLock与Condition**

2012年12月12日 21:41:39

阅读数：40000

**一、ReentrantLock 类**

**1.1 什么是reentrantlock**

java.util.concurrent.lock 中的 Lock 框架是锁定的一个抽象，它允许把锁定的实现作为 Java 类，而不是作为语言的特性来实现。这就为 Lock 的多种实现留下了空间，各种实现可能有不同的调度算法、性能特性或者锁定语义。 ReentrantLock 类实现了 Lock ，它拥有与 synchronized 相同的并发性和内存语义，但是添加了类似锁投票、定时锁等候和可中断锁等候的一些特性。此外，它还提供了在激烈争用情况下更佳的性能。（换句话说，当许多线程都想访问共享资源时，JVM 可以花更少的时候来调度线程，把更多时间用在执行线程上。）

**reentrant 锁意味着什么呢？**简单来说，它有一个与锁相关的获取计数器，如果拥有锁的某个线程再次得到锁，那么获取计数器就加1，然后锁需要被释放两次才能获得真正释放。这模仿了 synchronized 的语义；如果线程进入由线程已经拥有的监控器保护的 synchronized 块，就允许线程继续进行，当线程退出第二个（或者后续） synchronized 块的时候，不释放锁，只有线程退出它进入的监控器保护的第一个 synchronized 块时，才释放锁。

**1.2 ReentrantLock与synchronized的比较**

**相同：**ReentrantLock提供了synchronized类似的功能和内存语义。

**不同：**

（1）ReentrantLock功能性方面更全面，比如时间锁等候，可中断锁等候，锁投票等，因此更有扩展性。在多个条件变量和高度竞争锁的地方，用ReentrantLock更合适，ReentrantLock还提供了Condition，对线程的等待和唤醒等操作更加灵活，一个ReentrantLock可以有多个Condition实例，所以更有扩展性。

（2）ReentrantLock 的性能比synchronized会好点。

（3）ReentrantLock提供了可轮询的锁请求，他可以尝试的去取得锁，如果取得成功则继续处理，取得不成功，可以等下次运行的时候处理，所以不容易产生死锁，而synchronized则一旦进入锁请求要么成功，要么一直阻塞，所以更容易产生死锁。

**1.3 ReentrantLock扩展的功能**

**1.3.1 实现可轮询的锁请求**

在内部锁中，死锁是致命的——唯一的恢复方法是重新启动程序，唯一的预防方法是在构建程序时不要出错。而可轮询的锁获取模式具有更完善的错误恢复机制，可以规避死锁的发生。   
如果你不能获得所有需要的锁，那么使用可轮询的获取方式使你能够重新拿到控制权，它会释放你已经获得的这些锁，然后再重新尝试。可轮询的锁获取模式，由tryLock()方法实现。此方法仅在调用时锁为空闲状态才获取该锁。如果锁可用，则获取锁，并立即返回值true。如果锁不可用，则此方法将立即返回值false。此方法的典型使用语句如下： 

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. Lock lock = ...;
2. **if** (lock.tryLock()) {
3. **try** {
4. // manipulate protected state
5. } **finally** {
6. lock.unlock();
7. }
8. } **else** {
9. // perform alternative actions
10. }

**1.3.2 实现可定时的锁请求**

当使用内部锁时，一旦开始请求，锁就不能停止了，所以内部锁给实现具有时限的活动带来了风险。为了解决这一问题，可以使用定时锁。当具有时限的活   
动调用了阻塞方法，定时锁能够在时间预算内设定相应的超时。如果活动在期待的时间内没能获得结果，定时锁能使程序提前返回。可定时的锁获取模式，由tryLock(long, TimeUnit)方法实现。

**1.3.3 实现可中断的锁获取请求**

可中断的锁获取操作允许在可取消的活动中使用。lockInterruptibly()方法能够使你获得锁的时候响应中断。

**1.4 ReentrantLock不好与需要注意的地方**

（1） lock 必须在 finally 块中释放。否则，如果受保护的代码将抛出异常，锁就有可能永远得不到释放！这一点区别看起来可能没什么，但是实际上，它极为重要。忘记在 finally 块中释放锁，可能会在程序中留下一个定时炸弹，当有一天炸弹爆炸时，您要花费很大力气才有找到源头在哪。而使用同步，JVM 将确保锁会获得自动释放

（2） 当 JVM 用 synchronized 管理锁定请求和释放时，JVM 在生成线程转储时能够包括锁定信息。这些对调试非常有价值，因为它们能标识死锁或者其他异常行为的来源。 Lock 类只是普通的类，JVM 不知道具体哪个线程拥有 Lock 对象。

**二、条件变量Condition**

**切记:condition的相关方法都必须在 lock.lock() … lock.unlock(之间使用),否则会产生死锁,正确使用如下**

|  |
| --- |
| **try**{  lock.lock();  condition.signalAll();  }**finally**{  **if**(lock.isLocked()){  lock.unlock();  }  } |

条件变量很大一个程度上是为了解决Object.wait/notify/notifyAll难以使用的问题。

**条件**（也称为*条件队列* 或*条件变量*）为线程提供了一个含义，以便在某个状态条件现在可能为 true 的另一个线程通知它之前，一直挂起该线程（即让其“等待”）。因为访问此共享状态信息发生在不同的线程中，所以它必须受保护，因此要将某种形式的锁与该条件相关联。等待提供一个条件的主要属性是：*以原子方式* 释放相关的锁，并挂起当前线程，就像 Object.wait 做的那样。

上述API说明表明条件变量需要与锁绑定，而且多个Condition需要绑定到同一锁上。前面的**Lock**中提到，获取一个条件变量的方法是**Lock.newCondition()**。

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. **void** await() **throws** InterruptedException;
3. **void** awaitUninterruptibly();
5. **long** awaitNanos(**long** nanosTimeout) **throws** InterruptedException;
7. **boolean** await(**long** time, TimeUnit unit) **throws** InterruptedException;
9. **boolean** awaitUntil(Date deadline) **throws** InterruptedException;
11. **void** signal();
13. **void** signalAll();

以上是**Condition**接口定义的方法，*await\**对应于*Object.wait*，*signal*对应于*Object.notify*，*signalAll*对应于*Object.notifyAll*。特别说明的是**Condition**的接口改变名称就是为了避免与Object中的*wait/notify/notifyAll*的语义和使用上混淆，因为Condition同样有*wait/notify/notifyAll*方法。

每一个**Lock**可以有任意数据的**Condition**对象，**Condition**是与**Lock**绑定的，所以就有**Lock**的公平性特性：如果是公平锁，线程为按照FIFO的顺序从*Condition.await*中释放，如果是非公平锁，那么后续的锁竞争就不保证FIFO顺序了。

一个使用Condition实现**生产者消费者**的模型例子如下。

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. ***import****java.util.concurrent.locks.Condition;*
2. ***import****java.util.concurrent.locks.Lock;*
3. ***import****java.util.concurrent.locks.ReentrantLock;*
5. ***public******class****ProductQueue<T> {*
7. ***private******final****T[] items;*
9. ***private******final****Lock lock =****new****ReentrantLock();*
11. ***private****Condition notFull = lock.newCondition();*
13. ***private****Condition notEmpty = lock.newCondition();*
15. *//*
16. ***private******int****head, tail, count;*
18. ***public****ProductQueue(****int****maxSize) {*
19. *items = (T[])****new****Object[maxSize];*
20. *}*
22. ***public****ProductQueue() {*
23. ***this****(10);*
24. *}*
26. ***public******void****put(T t)****throws****InterruptedException {*
27. *lock.lock();*
28. ***try****{*
29. ***while****(count == getCapacity()) {*
30. *notFull.await();*
31. *}*
32. *items[tail] = t;*
33. ***if****(++tail == getCapacity()) {*
34. *tail = 0;*
35. *}*
36. *++count;*
37. *notEmpty.signalAll();*
38. *}****finally****{*
39. *lock.unlock();*
40. *}*
41. *}*
43. ***public****T take()****throws****InterruptedException {*
44. *lock.lock();*
45. ***try****{*
46. ***while****(count == 0) {*
47. *notEmpty.await();*
48. *}*
49. *T ret = items[head];*
50. *items[head] =****null****;//GC*
51. *//*
52. ***if****(++head == getCapacity()) {*
53. *head = 0;*
54. *}*
55. *--count;*
56. *notFull.signalAll();*
57. ***return****ret;*
58. *}****finally****{*
59. *lock.unlock();*
60. *}*
61. *}*
63. ***public******int****getCapacity() {*
64. ***return****items.length;*
65. *}*
67. ***public******int****size() {*
68. *lock.lock();*
69. ***try****{*
70. ***return****count;*
71. *}****finally****{*
72. *lock.unlock();*
73. *}*
74. *}*
76. *}*

在这个例子中消费*take()*需要 队列不为空，如果为空就挂起（*await()*），直到收到*notEmpty*的信号；生产*put()*需要队列不满，如果满了就挂起（*await()*），直到收到*notFull*的信号。

可能有人会问题，如果一个线程*lock()*对象后被挂起还没有*unlock*，那么另外一个线程就拿不到锁了（*lock()*操作会挂起），那么就无法通知(*notify*)前一个线程，这样岂不是“死锁”了？

**2.1 await\* 操作**

上一节中说过多次*ReentrantLock*是独占锁，一个线程拿到锁后如果不释放，那么另外一个线程肯定是拿不到锁，所以在*lock.lock()*和*lock.unlock()*之间可能有一次释放锁的操作（同样也必然还有一次获取锁的操作）。我们再回头看代码，不管*take()*还是*put()*，在进入*lock.lock()*后唯一可能释放锁的操作就是*await()*了。也就是说*await()*操作实际上就是释放锁，然后挂起线程，一旦条件满足就被唤醒，再次获取锁！

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. ***public******final******void****await()****throws****InterruptedException {*
2. ***if****(Thread.interrupted())*
3. ***throw******new****InterruptedException();*
4. *Node node = addConditionWaiter();*
5. ***int****savedState = fullyRelease(node);*
6. ***int****interruptMode = 0;*
7. ***while****(!isOnSyncQueue(node)) {*
8. *LockSupport.park(****this****);*
9. ***if****((interruptMode = checkInterruptWhileWaiting(node)) != 0)*
10. ***break****;*
11. *}*
12. ***if****(acquireQueued(node, savedState) && interruptMode != THROW\_IE)*
13. *interruptMode = REINTERRUPT;*
14. ***if****(node.nextWaiter !=****null****)*
15. *unlinkCancelledWaiters();*
16. ***if****(interruptMode != 0)*
17. *reportInterruptAfterWait(interruptMode);*
18. *}*

上面是*await()*的代码片段。上一节中说过,AQS在获取锁的时候需要有一个**[CHL](http://blog.csdn.net/a511596982/article/details/8275624" \t "_blank)**的FIFO队列，所以对于一个*Condition.await()*而言，如果释放了锁，要想再一次获取锁那么就需要进入队列，等待被通知获取锁。完整的await()操作是安装如下步骤进行的：

* 1. 将当前线程加入*Condition*锁队列。特别说明的是，这里不同于**[AQS](http://blog.csdn.net/a511596982/article/details/8275624" \t "_blank)**的队列，这里进入的是*Condition*的FIFO队列。后面会具体谈到此结构。进行2。
  2. 释放锁。这里可以看到将锁释放了，否则别的线程就无法拿到锁而发生死锁。进行3。
  3. 自旋(while)挂起，直到被唤醒或者超时或者CACELLED等。进行4。
  4. 获取锁(*acquireQueued*)。并将自己从*Condition*的FIFO队列中释放，表明自己不再需要锁（我已经拿到锁了）。

这里再回头介绍*Condition*的数据结构。我们知道一个*Condition*可以在多个地方被*await\*()*，那么就需要一个FIFO的结构将这些*Condition*串联起来，然后根据需要唤醒一个或者多个（通常是所有）。所以在*Condition*内部就需要一个FIFO的队列。

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. ***private******transient****Node firstWaiter;*
2. ***private******transient****Node lastWaiter;*

上面的两个节点就是描述一个FIFO的队列。我们再结合前面提到的节点（Node）数据结构。我们就发现*Node.nextWaiter*就派上用场了！*nextWaiter*就是将一系列的*Condition.await\**串联起来组成一个FIFO的队列。

**2.2 signal/signalAll 操作**

*await\*()*清楚了，现在再来看*signal/signalAll*就容易多了。按照*signal/signalAll*的需求，就是要将*Condition.await\*()*中FIFO队列中第一个**Node**唤醒（或者全部**Node**）唤醒。尽管所有**Node**可能都被唤醒，但是要知道的是仍然只有一个线程能够拿到锁，其它没有拿到锁的线程仍然需要自旋等待，就上上面提到的第4步(acquireQueued)。

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. ***private******void****doSignal(Node first) {*
2. ***do****{*
3. ***if****( (firstWaiter = first.nextWaiter) ==****null****)*
4. *lastWaiter =****null****;*
5. *first.nextWaiter =****null****;*
6. *}****while****(!transferForSignal(first) &&*
7. *(first = firstWaiter) !=****null****);*
8. *}*
10. ***private******void****doSignalAll(Node first) {*
11. *lastWaiter = firstWaiter  =****null****;*
12. ***do****{*
13. *Node next = first.nextWaiter;*
14. *first.nextWaiter =****null****;*
15. *transferForSignal(first);*
16. *first = next;*
17. *}****while****(first !=****null****);*
18. *}*

上面的代码很容易看出来，*signal*就是唤醒**Condition**队列中的第一个非CANCELLED节点线程，而signalAll就是唤醒所有非CANCELLED节点线程。当然了遇到CANCELLED线程就需要将其从FIFO队列中剔除。

**[java]** [view plain](https://blog.csdn.net/vernonzheng/article/details/8288251) [copy](https://blog.csdn.net/vernonzheng/article/details/8288251)

1. ***final******boolean****transferForSignal(Node node) {*
2. ***if****(!compareAndSetWaitStatus(node, Node.CONDITION, 0))*
3. ***return******false****;*
5. *Node p = enq(node);*
6. ***int****c = p.waitStatus;*
7. ***if****(c > 0 || !compareAndSetWaitStatus(p, c, Node.SIGNAL))*
8. *LockSupport.unpark(node.thread);*
9. ***return******true****;*
10. *}*

上面就是唤醒一个*await\*()*线程的过程，根据前面的小节介绍的，如果要*unpark*线程，并使线程拿到锁，那么就需要线程节点进入**AQS**的队列。所以可以看到在*LockSupport.unpark*之前调用了*enq(node)*操作，将当前节点加入到**AQS**队列。

参考：

《深入浅出 Java Concurrency》—锁机制(一)Lock与ReentrantLock  
http://blog.csdn.net/fg2006/article/details/6397894  
Java多线程基础总结七：ReentrantLock(2)  
http://www.bianceng.cn/Programming/Java/201206/34155\_2.htm  
再谈重入锁--ReentrantLock  
http://tenyears.iteye.com/blog/48750  
深入浅出 Java Concurrency (9): 锁机制 part 4  
http://www.blogjava.net/xylz/archive/2010/07/08/325540.html