 AQS是什么？

     字面上看，它被称为抽象队列式的同步器(AbstractQueuedSynchronizer)。简单说，它就是一个同步队列容器。

AQS有什么用？

1. 为什么会产生ArrayList、LinkedList、HashMap这些容器？它们底层实现无非都是对数组、链表、树的操作，至于它们的产生，就是因为对编程人员对于数组、链表、树的增删改查操作非常繁琐而提出的解决方案。
2. 那为什么会产生AQS呢？谈到同步，大家最容易想到的就是在多线程中如何确保安全的资源共享。那同步队列就是为了解决资源共享的同步容器。像上述容器一样，在顶层就设计好，编程人员只需要调用接口就能轻易实现复杂的资源共享问题。

既然谈到资源共享，那同步容器怎么实现资源共享呢？

     AQS定义两种资源共享方式：Exclusive(独占、只有一个线程执行，如ReentrantLock)和Share(共享，多个线程可同时执行，如Semaphore/CountDownLatch)。

那什么是独占式？

     在谈synchronized的资源共享实现方式的时候，当线程A访问共享资源的时候，其它的线程全部被堵塞，直到线程A读写完毕，其它线程才能申请同步互斥锁从而访问共享资源。如果之前看过我关于synchronized的讨论，这里应该不难理解，为了照顾未了解过的读者，再重新回顾一下。

     以RenentrantLock为例，如何知道共享资源是否有线程正在被访问呢？其实，它有一个state变量初始值为0，表示未锁定状态。当线程A访问的时候，state+1，就代表该线程锁定了共享资源，其他线程将无法访问，而当线程A访问完共享资源以后，state-1，直到state等于0，就将释放对共享变量的锁定，其他线程将可以抢占式或者公平式争夺。当然，它支持可重入，那什么是可重入呢？同一线程可以重复锁定共享资源，每锁定一次state+1，也就是锁定多次。说明：锁定多少次就要释放多少次。

什么是共享式呢？

     以CountDownLatch为例，共享资源可以被N个线程访问，也就是初始化的时候，state就被指定为N（N与线程个数相等），线程countDown()一次，state会CAS减1，直到所有线程执行完（state=0），那些await()的线程将被唤醒去执行执行剩余动作。

     什么是CAS？CAS的定义为Compare-And-Swap，语义为比较并且交换。在深入理解JVM书中，谈到自旋锁，因为锁的堵塞释放对于cpu资源的损害很高，那么自旋锁就是当线程A访问共享资源的时候，其他线程并不放弃对锁的持有，它们在不停循环，不断尝试的获取锁，直到获得锁就停止循环，自旋锁是对于资源共享的一种优化手段，但是它适用于对锁持有时间比较短的情况。

独占式lock流程(unlock同理)：

1. 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功就返回。
2. 没成功，则addWaiter()将线程加入等待队列的尾部，并标记为独享模式。
3. acquireQueued()使线程在等待队列中休息，有机会时会去尝试获得资源。获得资源后返回。如果整个过程有中断过返回true，否则返回false。
4. 如果线程在等待过程中中断过，它是不响应的。只是获得资源后才再进行自我中断selfInterrupt()，将中断补上。



共享式流程(类似于独占式 )：

1. tryAcquireShared()尝试获取资源，成功则直接返回。
2. 失败则通过 doAcquireShared()进入等待队列，直到被唤醒或者中断并且成功获取资源才返回。
3. 不同：独占式是只唤醒后继节点。共享式是唤醒后继，后继还会去唤醒它的后继，从而实现共享。

以上是核心的关于CountDownLatch、ReentrantLock的分析。由于博主研究程度有限，想更深层次研究，请参考：

# [Java并发AQS详解](http://www.cnblogs.com/waterystone/p/4920797.html)

## 一、概述

　　谈到并发，不得不谈ReentrantLock；而谈到ReentrantLock，不得不谈AbstractQueuedSynchronized（AQS）！

　　类如其名，抽象的队列式的同步器，AQS定义了一套多线程访问共享资源的同步器框架，许多同步类实现都依赖于它，如常用的ReentrantLock/Semaphore/CountDownLatch...。

　　以下是本文的目录大纲：

* 1. 概述
  2. 框架
  3. 源码详解
  4. 简单应用

　　若有不正之处，请谅解和批评指正，不胜感激。

　　请尊重作者劳动成果，转载请标明原文链接：<http://www.cnblogs.com/waterystone/p/4920797.html>

## 二、框架



　　它维护了一个volatile int state（代表共享资源）和一个FIFO线程等待队列（多线程争用资源被阻塞时会进入此队列）。这里volatile是核心关键词，具体volatile的语义，在此不述。state的访问方式有三种:

* getState()
* setState()
* compareAndSetState()

　　AQS定义两种资源共享方式：Exclusive（独占，只有一个线程能执行，如ReentrantLock）和Share（共享，多个线程可同时执行，如Semaphore/CountDownLatch）。

　　不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。

　　以ReentrantLock为例，state初始化为0，表示未锁定状态。A线程lock()时，会调用tryAcquire()独占该锁并将state+1。此后，其他线程再tryAcquire()时就会失败，直到A线程unlock()到state=0（即释放锁）为止，其它线程才有机会获取该锁。当然，释放锁之前，A线程自己是可以重复获取此锁的（state会累加），这就是可重入的概念。但要注意，获取多少次就要释放多么次，这样才能保证state是能回到零态的。

　　再以CountDownLatch以例，任务分为N个子线程去执行，state也初始化为N（注意N要与线程个数一致）。这N个子线程是并行执行的，每个子线程执行完后countDown()一次，state会CAS减1。等到所有子线程都执行完后(即state=0)，会unpark()主调用线程，然后主调用线程就会从await()函数返回，继续后余动作。

　　一般来说，自定义同步器要么是独占方法，要么是共享方式，他们也只需实现tryAcquire-tryRelease、tryAcquireShared-tryReleaseShared中的一种即可。但AQS也支持自定义同步器同时实现独占和共享两种方式，如ReentrantReadWriteLock。

## 三、源码详解

　　本节开始讲解AQS的源码实现。依照acquire-release、acquireShared-releaseShared的次序来。

### 3.1 acquire(int)

　　此方法是独占模式下线程获取共享资源的顶层入口。如果获取到资源，线程直接返回，否则进入等待队列，直到获取到资源为止，且整个过程忽略中断的影响。这也正是lock()的语义，当然不仅仅只限于lock()。获取到资源后，线程就可以去执行其临界区代码了。下面是acquire()的源码：

1 public final void acquire(int arg) {

2 if (!tryAcquire(arg) &&

3 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

4 selfInterrupt();

5 }

　　函数流程如下：

* 1. tryAcquire()尝试直接去获取资源，如果成功则直接返回；
  2. addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
  3. acquireQueued()使线程在等待队列中获取资源，一直获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
  4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

　　这时单凭这4个抽象的函数来看流程还有点朦胧，不要紧，看完接下来的分析后，你就会明白了。就像《大话西游》里唐僧说的：等你明白了舍生取义的道理，你自然会回来和我唱这首歌的。

#### 3.1.1 tryAcquire(int)

　　此方法尝试去获取独占资源。如果获取成功，则直接返回true，否则直接返回false。这也正是tryLock()的语义，还是那句话，当然不仅仅只限于tryLock()。如下是tryAcquire()的源码：

1 protected boolean tryAcquire(int arg) {

2 throw new UnsupportedOperationException();

3 }

　　什么？直接throw异常？说好的功能呢？好吧，**还记得概述里讲的AQS只是一个框架，具体资源的获取/释放方式交由自定义同步器去实现吗？**就是这里了！！！AQS这里只定义了一个接口，具体资源的获取交由自定义同步器去实现了（通过state的get/set/CAS）！！！至于能不能重入，能不能加塞，那就看具体的自定义同步器怎么去设计了！！！当然，自定义同步器在进行资源访问时要考虑线程安全的影响。

　　这里之所以没有定义成abstract，是因为独占模式下只用实现tryAcquire-tryRelease，而共享模式下只用实现tryAcquireShared-tryReleaseShared。如果都定义成abstract，那么每个模式也要去实现另一模式下的接口。说到底，Doug Lea还是站在咱们开发者的角度，尽量减少不必要的工作量。

#### 3.1.2 addWaiter(Node)

　　此方法用于将当前线程加入到等待队列的队尾，并返回当前线程所在的结点。还是上源码吧：

[复制代码](javascript:void(0);)

复制代码

1 private Node addWaiter(Node mode) {

2 //以给定模式构造结点。mode有两种：EXCLUSIVE（独占）和SHARED（共享）

3 Node node = new Node(Thread.currentThread(), mode);

4

5 //尝试快速方式直接放到队尾。

6 Node pred = tail;

7 if (pred != null) {

8 node.prev = pred;

9 if (compareAndSetTail(pred, node)) {

10 pred.next = node;

11 return node;

12 }

13 }

14

15 //上一步失败则通过enq入队。

16 enq(node);

17 return node;

18 }

复制代码

[复制代码](javascript:void(0);)

 不用再说了，直接看注释吧。

##### 3.1.2.1 enq(Node)

 　　此方法用于将node加入队尾。源码如下：

[复制代码](javascript:void(0);)

复制代码

1 private Node enq(final Node node) {

2 //CAS"自旋"，直到成功加入队尾

3 for (;;) {

4 Node t = tail;

5 if (t == null) { // 队列为空，创建一个空的标志结点作为head结点，并将tail也指向它。

6 if (compareAndSetHead(new Node()))

7 tail = head;

8 } else {//正常流程，放入队尾

9 node.prev = t;

10 if (compareAndSetTail(t, node)) {

11 t.next = node;

12 return t;

13 }

14 }

15 }

16 }

复制代码

[复制代码](javascript:void(0);)

如果你看过AtomicInteger.getAndIncrement()函数源码，那么相信你一眼便看出这段代码的精华。**CAS自旋volatile变量**，是一种很经典的用法。还不太了解的，自己去百度一下吧。

#### 3.1.3 acquireQueued(Node, int)

　　OK，通过tryAcquire()和addWaiter()，该线程获取资源失败，已经被放入等待队列尾部了。聪明的你立刻应该能想到该线程下一部该干什么了吧：**进入等待状态休息，直到其他线程彻底释放资源后唤醒自己，自己再拿到资源，然后就可以去干自己想干的事了**。没错，就是这样！是不是跟医院排队拿号有点相似~~acquireQueued()就是干这件事：**在等待队列中排队拿号（中间没其它事干可以休息），直到拿到号后再返回**。这个函数非常关键，还是上源码吧：

[复制代码](javascript:void(0);)

复制代码

1 final boolean acquireQueued(final Node node, int arg) {

2 boolean failed = true;//标记是否成功拿到资源

3 try {

4 boolean interrupted = false;//标记等待过程中是否被中断过

5

6 //又是一个“自旋”！

7 for (;;) {

8 final Node p = node.predecessor();//拿到前驱

9 //如果前驱是head，即该结点已成老二，那么便有资格去尝试获取资源（可能是老大释放完资源唤醒自己的，当然也可能被interrupt了）。

10 if (p == head && tryAcquire(arg)) {

11 setHead(node);//拿到资源后，将head指向该结点。所以head所指的标杆结点，就是当前获取到资源的那个结点或null。

12 p.next = null; // setHead中node.prev已置为null，此处再将head.next置为null，就是为了方便GC回收以前的head结点。也就意味着之前拿完资源的结点出队了！

13 failed = false;

14 return interrupted;//返回等待过程中是否被中断过

15 }

16

17 //如果自己可以休息了，就进入waiting状态，直到被unpark()

18 if (shouldParkAfterFailedAcquire(p, node) &&

19 parkAndCheckInterrupt())

20 interrupted = true;//如果等待过程中被中断过，哪怕只有那么一次，就将interrupted标记为true

21 }

22 } finally {

23 if (failed)

24 cancelAcquire(node);

25 }

26 }

复制代码

[复制代码](javascript:void(0);)

到这里了，我们先不急着总结acquireQueued()的函数流程，先看看shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()具体干些什么。

##### 3.1.3.1 shouldParkAfterFailedAcquire(Node, Node)

　　此方法主要用于检查状态，看看自己是否真的可以去休息了（进入waiting状态，如果线程状态转换不熟，可以参考本人上一篇写的[Thread详解](http://www.cnblogs.com/waterystone/p/4920007.html)），万一队列前边的线程都放弃了只是瞎站着，那也说不定，对吧！

[复制代码](javascript:void(0);)

复制代码

1 private static boolean shouldParkAfterFailedAcquire(Node pred, Node node) {

2 int ws = pred.waitStatus;//拿到前驱的状态

3 if (ws == Node.SIGNAL)

4 //如果已经告诉前驱拿完号后通知自己一下，那就可以安心休息了

5 return true;

6 if (ws > 0) {

7 /\*

8 \* 如果前驱放弃了，那就一直往前找，直到找到最近一个正常等待的状态，并排在它的后边。

9 \* 注意：那些放弃的结点，由于被自己“加塞”到它们前边，它们相当于形成一个无引用链，稍后就会被保安大叔赶走了(GC回收)！

10 \*/

11 do {

12 node.prev = pred = pred.prev;

13 } while (pred.waitStatus > 0);

14 pred.next = node;

15 } else {

16 //如果前驱正常，那就把前驱的状态设置成SIGNAL，告诉它拿完号后通知自己一下。有可能失败，人家说不定刚刚释放完呢！

17 compareAndSetWaitStatus(pred, ws, Node.SIGNAL);

18 }

19 return false;

20 }

复制代码

[复制代码](javascript:void(0);)

整个流程中，如果前驱结点的状态不是SIGNAL，那么自己就不能安心去休息，需要去找个安心的休息点，同时可以再尝试下看有没有机会轮到自己拿号。

##### 3.1.3.2 parkAndCheckInterrupt()

　　如果线程找好安全休息点后，那就可以安心去休息了。此方法就是让线程去休息，真正进入等待状态。

1 private final boolean parkAndCheckInterrupt() {

2 LockSupport.park(this);//调用park()使线程进入waiting状态

3 return Thread.interrupted();//如果被唤醒，查看自己是不是被中断的。

4 }

 　　park()会让当前线程进入waiting状态。在此状态下，有两种途径可以唤醒该线程：1）被unpark()；2）被interrupt()。（再说一句，如果线程状态转换不熟，可以参考本人写的[Thread详解](http://www.cnblogs.com/waterystone/p/4920007.html)）。需要注意的是，Thread.interrupted()会清除当前线程的中断标记位。

##### 3.1.3.3 小结

　　OK，看了shouldParkAfterFailedAcquire()和parkAndCheckInterrupt()，现在让我们再回到acquireQueued()，总结下该函数的具体流程：

1. 结点进入队尾后，检查状态，找到安全休息点；
2. 调用park()进入waiting状态，等待unpark()或interrupt()唤醒自己；
3. 被唤醒后，看自己是不是有资格能拿到号。如果拿到，head指向当前结点，并返回从入队到拿到号的整个过程中是否被中断过；如果没拿到，继续流程1。

#### 3.1.4 小结

　　OKOK，acquireQueued()分析完之后，我们接下来再回到acquire()！再贴上它的源码吧：

1 public final void acquire(int arg) {

2 if (!tryAcquire(arg) &&

3 acquireQueued(addWaiter(Node.EXCLUSIVE), arg))

4 selfInterrupt();

5 }

再来总结下它的流程吧：

1. 调用自定义同步器的tryAcquire()尝试直接去获取资源，如果成功则直接返回；
2. 没成功，则addWaiter()将该线程加入等待队列的尾部，并标记为独占模式；
3. acquireQueued()使线程在等待队列中休息，有机会时（轮到自己，会被unpark()）会去尝试获取资源。获取到资源后才返回。如果在整个等待过程中被中断过，则返回true，否则返回false。
4. 如果线程在等待过程中被中断过，它是不响应的。只是获取资源后才再进行自我中断selfInterrupt()，将中断补上。

由于此函数是重中之重，我再用流程图总结一下：



至此，acquire()的流程终于算是告一段落了。这也就是ReentrantLock.lock()的流程，不信你去看其lock()源码吧，整个函数就是一条acquire(1)！！！

### 3.2 release(int)

 　　上一小节已经把acquire()说完了，这一小节就来讲讲它的反操作release()吧。此方法是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。这也正是unlock()的语义，当然不仅仅只限于unlock()。下面是release()的源码：

[复制代码](javascript:void(0);)

复制代码

1 public final boolean release(int arg) {

2 if (tryRelease(arg)) {

3 Node h = head;//找到头结点

4 if (h != null && h.waitStatus != 0)

5 unparkSuccessor(h);//唤醒等待队列里的下一个线程

6 return true;

7 }

8 return false;

9 }

复制代码

[复制代码](javascript:void(0);)

　　逻辑并不复杂。它调用tryRelease()来释放资源。有一点需要注意的是，**它是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！所以自定义同步器在设计tryRelease()的时候要明确这一点！！**

#### 3.2.1 tryRelease(int)

　　此方法尝试去释放指定量的资源。下面是tryRelease()的源码：

1 protected boolean tryRelease(int arg) {

2 throw new UnsupportedOperationException();

3 }

　　跟tryAcquire()一样，这个方法是需要独占模式的自定义同步器去实现的。正常来说，tryRelease()都会成功的，因为这是独占模式，该线程来释放资源，那么它肯定已经拿到独占资源了，直接减掉相应量的资源即可(state-=arg)，也不需要考虑线程安全的问题。但要注意它的返回值，上面已经提到了，**release()是根据tryRelease()的返回值来判断该线程是否已经完成释放掉资源了！**所以自义定同步器在实现时，如果已经彻底释放资源(state=0)，要返回true，否则返回false。

#### 3.2.2 unparkSuccessor(Node)

　　此方法用于唤醒等待队列中下一个线程。下面是源码：

[复制代码](javascript:void(0);)

复制代码

1 private void unparkSuccessor(Node node) {

2 //这里，node一般为当前线程所在的结点。

3 int ws = node.waitStatus;

4 if (ws < 0)//置零当前线程所在的结点状态，允许失败。

5 compareAndSetWaitStatus(node, ws, 0);

6

7 Node s = node.next;//找到下一个需要唤醒的结点s

8 if (s == null || s.waitStatus > 0) {//如果为空或已取消

9 s = null;

10 for (Node t = tail; t != null && t != node; t = t.prev)

11 if (t.waitStatus <= 0)//从这里可以看出，<=0的结点，都是还有效的结点。

12 s = t;

13 }

14 if (s != null)

15 LockSupport.unpark(s.thread);//唤醒

16 }

复制代码

[复制代码](javascript:void(0);)

　　这个函数并不复杂。一句话概括：**用unpark()唤醒等待队列中最前边的那个未放弃线程**，这里我们也用s来表示吧。此时，再和acquireQueued()联系起来，s被唤醒后，进入if (p == head && tryAcquire(arg))的判断（即使p!=head也没关系，它会再进入shouldParkAfterFailedAcquire()寻找一个安全点。这里既然s已经是等待队列中最前边的那个未放弃线程了，那么通过shouldParkAfterFailedAcquire()的调整，s也必然会跑到head的next结点，下一次自旋p==head就成立啦），然后s把自己设置成head标杆结点，表示自己已经获取到资源了，acquire()也返回了！！And then, DO what you WANT!

#### 3.2.3 小结

　　release()是独占模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。

### 3.3 acquireShared(int)

　　此方法是共享模式下线程获取共享资源的顶层入口。它会获取指定量的资源，获取成功则直接返回，获取失败则进入等待队列，直到获取到资源为止，整个过程忽略中断。下面是acquireShared()的源码：

1 public final void acquireShared(int arg) {

2 if (tryAcquireShared(arg) < 0)

3 doAcquireShared(arg);

4 }

　　这里tryAcquireShared()依然需要自定义同步器去实现。但是AQS已经把其返回值的语义定义好了：负值代表获取失败；0代表获取成功，但没有剩余资源；正数表示获取成功，还有剩余资源，其他线程还可以去获取。所以这里acquireShared()的流程就是：

* 1. tryAcquireShared()尝试获取资源，成功则直接返回；
  2. 失败则通过doAcquireShared()进入等待队列，直到获取到资源为止才返回。

#### 3.3.1 doAcquireShared(int)

　　此方法用于将当前线程加入等待队列尾部休息，直到其他线程释放资源唤醒自己，自己成功拿到相应量的资源后才返回。下面是doAcquireShared()的源码：

[复制代码](javascript:void(0);)

复制代码

1 private void doAcquireShared(int arg) {

2 final Node node = addWaiter(Node.SHARED);//加入队列尾部

3 boolean failed = true;//是否成功标志

4 try {

5 boolean interrupted = false;//等待过程中是否被中断过的标志

6 for (;;) {

7 final Node p = node.predecessor();//前驱

8 if (p == head) {//如果到head的下一个，因为head是拿到资源的线程，此时node被唤醒，很可能是head用完资源来唤醒自己的

9 int r = tryAcquireShared(arg);//尝试获取资源

10 if (r >= 0) {//成功

11 setHeadAndPropagate(node, r);//将head指向自己，还有剩余资源可以再唤醒之后的线程

12 p.next = null; // help GC

13 if (interrupted)//如果等待过程中被打断过，此时将中断补上。

14 selfInterrupt();

15 failed = false;

16 return;

17 }

18 }

19

20 //判断状态，寻找安全点，进入waiting状态，等着被unpark()或interrupt()

21 if (shouldParkAfterFailedAcquire(p, node) &&

22 parkAndCheckInterrupt())

23 interrupted = true;

24 }

25 } finally {

26 if (failed)

27 cancelAcquire(node);

28 }

29 }

复制代码

[复制代码](javascript:void(0);)

　　有木有觉得跟acquireQueued()很相似？对，其实流程并没有太大区别。只不过这里将补中断的selfInterrupt()放到doAcquireShared()里了，而独占模式是放到acquireQueued()之外，其实都一样，不知道Doug Lea是怎么想的。

　　跟独占模式比，还有一点需要注意的是，这里只有线程是head.next时（“老二”），才会去尝试获取资源，有剩余的话还会唤醒之后的队友。那么问题就来了，假如老大用完后释放了5个资源，而老二需要6个，老三需要1个，老四需要2个。因为老大先唤醒老二，老二一看资源不够自己用继续park()，也更不会去唤醒老三和老四了。独占模式，同一时刻只有一个线程去执行，这样做未尝不可；但共享模式下，多个线程是可以同时执行的，现在因为老二的资源需求量大，而把后面量小的老三和老四也都卡住了。

#### 3.3.1.1 setHeadAndPropagate(Node, int)

[复制代码](javascript:void(0);)

复制代码

1 private void setHeadAndPropagate(Node node, int propagate) {

2 Node h = head;

3 setHead(node);//head指向自己

4 //如果还有剩余量，继续唤醒下一个邻居线程

5 if (propagate > 0 || h == null || h.waitStatus < 0) {

6 Node s = node.next;

7 if (s == null || s.isShared())

8 doReleaseShared();

9 }

10 }

复制代码

[复制代码](javascript:void(0);)

　　此方法在setHead()的基础上多了一步，就是自己苏醒的同时，如果条件符合（比如还有剩余资源），还会去唤醒后继结点，毕竟是共享模式！

　　doReleaseShared()我们留着下一小节的releaseShared()里来讲。

#### 3.3.2 小结

　　OK，至此，acquireShared()也要告一段落了。让我们再梳理一下它的流程：

* 1. tryAcquireShared()尝试获取资源，成功则直接返回；
  2. 失败则通过doAcquireShared()进入等待队列park()，直到被unpark()/interrupt()并成功获取到资源才返回。整个等待过程也是忽略中断的。

　　其实跟acquire()的流程大同小异，只不过多了个**自己拿到资源后，还会去唤醒后继队友的操作（这才是共享嘛）**。

### 3.4 releaseShared()

　　上一小节已经把acquireShared()说完了，这一小节就来讲讲它的反操作releaseShared()吧。此方法是共享模式下线程释放共享资源的顶层入口。它会释放指定量的资源，如果彻底释放了（即state=0）,它会唤醒等待队列里的其他线程来获取资源。下面是releaseShared()的源码：

[复制代码](javascript:void(0);)

复制代码

1 public final boolean releaseShared(int arg) {

2 if (tryReleaseShared(arg)) {//尝试释放资源

3 doReleaseShared();//唤醒后继结点

4 return true;

5 }

6 return false;

7 }

复制代码

[复制代码](javascript:void(0);)

　　此方法的流程也比较简单，一句话：释放掉资源后，唤醒后继。跟独占模式下的release()相似，但有一点稍微需要注意：独占模式下的tryRelease()在完全释放掉资源（state=0）后，才会返回true去唤醒其他线程，这主要是基于可重入的考量；而共享模式下的releaseShared()则没有这种要求，一是共享的实质--多线程可并发执行；二是共享模式基本也不会重入吧（至少我还没见过），所以自定义同步器可以根据需要决定返回值。

#### 3.4.1 doReleaseShared()

　　此方法主要用于唤醒后继。下面是它的源码：

[复制代码](javascript:void(0);)

复制代码

1 private void doReleaseShared() {

2 for (;;) {

3 Node h = head;

4 if (h != null && h != tail) {

5 int ws = h.waitStatus;

6 if (ws == Node.SIGNAL) {

7 if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))

8 continue;

9 unparkSuccessor(h);//唤醒后继

10 }

11 else if (ws == 0 &&

12 !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))

13 continue;

14 }

15 if (h == head)// head发生变化

16 break;

17 }

18 }

复制代码

[复制代码](javascript:void(0);)

### 3.5 小结

　　本节我们详解了独占和共享两种模式下获取-释放资源(acquire-release、acquireShared-releaseShared)的源码，相信大家都有一定认识了。值得注意的是，acquire()和acquireSahred()两种方法下，线程在等待队列中都是忽略中断的。AQS也支持响应中断的，acquireInterruptibly()/acquireSharedInterruptibly()即是，这里相应的源码跟acquire()和acquireSahred()差不多，这里就不再详解了。

## 四、简单应用

　　通过前边几个章节的学习，相信大家已经基本理解AQS的原理了。这里再将“框架”一节中的一段话复制过来：

　　不同的自定义同步器争用共享资源的方式也不同。**自定义同步器在实现时只需要实现共享资源state的获取与释放方式即可**，至于具体线程等待队列的维护（如获取资源失败入队/唤醒出队等），AQS已经在顶层实现好了。自定义同步器实现时主要实现以下几种方法：

* isHeldExclusively()：该线程是否正在独占资源。只有用到condition才需要去实现它。
* tryAcquire(int)：独占方式。尝试获取资源，成功则返回true，失败则返回false。
* tryRelease(int)：独占方式。尝试释放资源，成功则返回true，失败则返回false。
* tryAcquireShared(int)：共享方式。尝试获取资源。负数表示失败；0表示成功，但没有剩余可用资源；正数表示成功，且有剩余资源。
* tryReleaseShared(int)：共享方式。尝试释放资源，成功则返回true，失败则返回false。

　　OK，下面我们就以AQS源码里的Mutex为例，讲一下AQS的简单应用。

### 4.1 Mutex（互斥锁）

　　Mutex是一个不可重入的互斥锁实现。锁资源（AQS里的state）只有两种状态：0表示未锁定，1表示锁定。下边是Mutex的核心源码：

[复制代码](javascript:void(0);)

复制代码

1 class Mutex implements Lock, java.io.Serializable {

2 // 自定义同步器

3 private static class Sync extends AbstractQueuedSynchronizer {

4 // 判断是否锁定状态

5 protected boolean isHeldExclusively() {

6 return getState() == 1;

7 }

8

9 // 尝试获取资源，立即返回。成功则返回true，否则false。

10 public boolean tryAcquire(int acquires) {

11 assert acquires == 1; // 这里限定只能为1个量

12 if (compareAndSetState(0, 1)) {//state为0才设置为1，不可重入！

13 setExclusiveOwnerThread(Thread.currentThread());//设置为当前线程独占资源

14 return true;

15 }

16 return false;

17 }

18

19 // 尝试释放资源，立即返回。成功则为true，否则false。

20 protected boolean tryRelease(int releases) {

21 assert releases == 1; // 限定为1个量

22 if (getState() == 0)//既然来释放，那肯定就是已占有状态了。只是为了保险，多层判断！

23 throw new IllegalMonitorStateException();

24 setExclusiveOwnerThread(null);

25 setState(0);//释放资源，放弃占有状态

26 return true;

27 }

28 }

29

30 // 真正同步类的实现都依赖继承于AQS的自定义同步器！

31 private final Sync sync = new Sync();

32

33 //lock<-->acquire。两者语义一样：获取资源，即便等待，直到成功才返回。

34 public void lock() {

35 sync.acquire(1);

36 }

37

38 //tryLock<-->tryAcquire。两者语义一样：尝试获取资源，要求立即返回。成功则为true，失败则为false。

39 public boolean tryLock() {

40 return sync.tryAcquire(1);

41 }

42

43 //unlock<-->release。两者语文一样：释放资源。

44 public void unlock() {

45 sync.release(1);

46 }

47

48 //锁是否占有状态

49 public boolean isLocked() {

50 return sync.isHeldExclusively();

51 }

52 }

复制代码

[复制代码](javascript:void(0);)

　　同步类在实现时一般都将自定义同步器（sync）定义为内部类，供自己使用；而同步类自己（Mutex）则实现某个接口，对外服务。当然，接口的实现要直接依赖sync，它们在语义上也存在某种对应关系！！而sync只用实现资源state的获取-释放方式tryAcquire-tryRelelase，至于线程的排队、等待、唤醒等，上层的AQS都已经实现好了，我们不用关心。

　　除了Mutex，ReentrantLock/CountDownLatch/Semphore这些同步类的实现方式都差不多，不同的地方就在获取-释放资源的方式tryAcquire-tryRelelase。掌握了这点，AQS的核心便被攻破了！

　　OK，至此，整个AQS的讲解也要落下帷幕了。希望本文能够对学习Java并发编程的同学有所借鉴，中间写的有不对的地方，也欢迎讨论和指正~

作者：[水岩](http://www.cnblogs.com/waterystone/)

出处：<http://www.cnblogs.com/waterystone/>

本博客中未标明转载的文章归作者[水岩](http://www.cnblogs.com/waterystone/)和博客园共有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，否则保留追究法律责任的权利。

# 浅谈CountDownLatch

CountDownLatch是什么? 有什么用?

     CountDownLatch是一个同步容器，但是有人叫它发令枪，也有人叫它门闩。初始化设定线程的个数，调用countDownLatch.await()阻塞所有线程，直到countDownLatch.countDown()为0，那么将继续执行剩余的操作。例如，跑步比赛，所有线程都await()在起跑线，当所有人告诉裁判准备好了，裁判发令枪一响，运动员开炮。门闩道理一样，门不开全给我等着！

     作用：为了实现同步共享数据的一种更加高效的解决办法。

[复制代码](javascript:void(0);)

/\*\*

\* CountDownLatch相当于指令枪或者门闩，所有线程都awit()阻塞在起跑线，只有countDown到state为0，其他线程才能往下运行。

\* @author qiuyongAaron

\*/

public class CountDownLatchDemo {

private static final int PLAYER\_NUM=5;

public static void main(String[] args) {

CountDownLatch start=new CountDownLatch(1);

CountDownLatch end =new CountDownLatch(PLAYER\_NUM);

Player [] players=new Player[PLAYER\_NUM];

for(int i=0;i<PLAYER\_NUM;i++)

players[i]=new Player(start, end, i);

//指定线程个数的线程池！

ExecutorService exe=Executors.newFixedThreadPool(PLAYER\_NUM);

for(Player player:players)

exe.execute(player);

System.out.println("比赛开始！");

//比赛开始！

start.countDown();

try {

end.await();

} catch (InterruptedException e) {

e.printStackTrace();

}finally{

System.out.println("比赛结束！");

exe.shutdown();

}

}

}

class Player implements Runnable{

private CountDownLatch start;

private CountDownLatch end;

private int id;

Random random=new Random();

public Player(CountDownLatch start,CountDownLatch end,int id) {

this.start=start;

this.end=end;

this.id=id;

}

@Override

public void run() {

try {

//等待比赛开始。

start.await();

TimeUnit.SECONDS.sleep(random.nextInt(10));

System.out.println("Player-"+id+":arrived");

} catch (InterruptedException e) {

e.printStackTrace();

}finally{

//选手-id到达终点,end计数为0结束比赛！

end.countDown();

}

}

}

//运行结果：

比赛开始！

Player-3:arrived

Player-4:arrived

Player-0:arrived

Player-1:arrived

Player-2:arrived

比赛结束！

[复制代码](javascript:void(0);)

# 谈ReentrantLock

1、ReentrantLock是什么?有什么用?

ReentrantLock跟synchronized作用差不多，是在于synchronized基础上的一种简易同步容器，并没有深层次的原理剖析。

2、ReentrantLock的基础用法

2.1 回顾synchronized如何实现线程同步。

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

/\*\*

\* 示例一：同步锁的使用

\* reentrantlock用于替代synchronized

\* 本例中由于m1锁定this,只有m1执行完毕的时候,m2才能执行

\* @author qiuyongAaron

\*/

public class ReentrantLockOne {

public synchronized void m1(){

for(int i=0;i<10;i++){

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

System.out.println(i);

}

}

public synchronized void m2(){

System.out.println("hello m2!");

}

public static void main(String[] args) {

ReentrantLockOne lock=new ReentrantLockOne();

new Thread(()->lock.m1(),"t1").start();

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

new Thread(()->lock.m2(),"t2").start();

}

}

[复制代码](javascript:void(0);)

 2.2 ReentrantLock实现线程同步-与synchronized作用一致！

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

/\*\*

\* 示例二：等价于同步锁

\* 使用reentrantlock可以完成同样的功能

\* 需要注意的是，必须要必须要必须要手动释放锁（重要的事情说三遍）

\* 使用syn锁定的话如果遇到异常，jvm会自动释放锁，但是lock必须手动释放锁，因此经常在finally中进行锁的释放

\* @author qiuyongAaron

\*/

public class ReentrantLockTwo {

ReentrantLock lock =new ReentrantLock();

public void m1(){

try {

lock.lock();

for(int i=0;i<10;i++){

TimeUnit.SECONDS.sleep(1);

System.out.println(i);

}

} catch (InterruptedException e) {

e.printStackTrace();

}finally{

lock.unlock();

}

}

public synchronized void m2(){

lock.lock();

System.out.println("hello m2!");

lock.unlock();

}

public static void main(String[] args) {

ReentrantLockTwo lock=new ReentrantLockTwo();

new Thread(()->lock.m1(),"t1").start();

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

new Thread(()->lock.m2(),"t2").start();

}

}

[复制代码](javascript:void(0);)

 2.3 ReentrantLock尝试获取锁，若指定时间无法获取锁放弃等待！

https://images.cnblogs.com/OutliningIndicators/ExpandedBlockStart.gif

[复制代码](javascript:void(0);)

/\*\*

\* 示例三：tryLock

\* 使用reentrantlock可以进行“尝试锁定”tryLock，这样无法锁定，或者在指定时间内无法锁定，线程可以决定是否继续等待

\* @author qiuyongAaron

\*/

public class ReentrantLockThree {

ReentrantLock lock=new ReentrantLock();

public void m1(){

try {

lock.lock();

for(int i=0;i<10;i++){

TimeUnit.SECONDS.sleep(1);

System.out.println(i);

}

} catch (Exception e) {

e.printStackTrace();

}finally{

lock.unlock();

}

}

boolean locked=false;

public void m2(){

try {

lock.tryLock(5,TimeUnit.SECONDS);

System.out.println("m2:"+locked);

} catch (Exception e) {

e.printStackTrace();

}finally{

if(locked) lock.unlock();

}

}

public static void main(String[] args) {

ReentrantLockThree lock=new ReentrantLockThree();

new Thread(()->lock.m1(),"t1").start();

try {

TimeUnit.SECONDS.sleep(1);

} catch (InterruptedException e) {

e.printStackTrace();

}

new Thread(()->lock.m2(),"t2").start();

}

}

[复制代码](javascript:void(0);)

 2.4 指定公平锁或者抢占式锁

https://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif ReentrantLock公平锁

3、ReentrantLock实现线程通信

[复制代码](javascript:void(0);)

/\*\*

\* 模拟生产者消费者模式-线程之间通信 synchronized-notifyAll/wait

\* @author qiuyongAaron

\*/

public class MyContainerOne {

LinkedList<Integer> list=new LinkedList<Integer>();

static final int MAX=10;

int count=0;

//生产者线程

public synchronized void put(int i){

while(list.size()==MAX){

try {

this.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

list.add(i);

++count;

this.notifyAll();//通知消费者来消费

}

//消费者线程

public synchronized int get(){

while(list.size()==0){

try {

this.wait();

} catch (InterruptedException e) {

e.printStackTrace();

}

}

int num=list.removeFirst();

count--;

this.notifyAll();//通知生产者生产

return num;

}

public static void main(String[] args) {

MyContainerOne container=new MyContainerOne();

//制造10个消费者

for(int i=0;i<10;i++){

new Thread(()->{

for(int j=0;j<5;j++) System.out.println(container.get());

},

"c"+i).start();

}

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

//制造2个生产者

for(int i=0;i<2;i++){

new Thread(()->{

for(int j=0;j<25;j++) container.put(j);

},

"p"+i).start();

}

}

}

[复制代码](javascript:void(0);)

[复制代码](javascript:void(0);)

/\*\*

\* 模拟生产者消费者模式-reentrantLock-awit/signAll

\* @author qiuyongAaron

\*/

public class MyContainerTwo {

LinkedList<Integer> list=new LinkedList<Integer>();

static final int MAX=10;

int count=0;

ReentrantLock lock=new ReentrantLock();

Condition producer=lock.newCondition();

Condition consumer=lock.newCondition();

//生产者线程

public void put(int i){

try {

lock.lock();

while(list.size()==MAX){

producer.await();

}

list.add(i);

++count;

consumer.signalAll();//通知消费者来消费

} catch (InterruptedException e){

e.printStackTrace();

}finally{

lock.unlock();

}

}

//消费者线程

public int get(){

try{

lock.lock();

while(list.size()==0){

consumer.await();

}

int num=list.removeFirst();

count--;

producer.signalAll();//通知生产者生产

return num;

}catch(Exception e){

e.printStackTrace();

}finally{

lock.unlock();

}

return 0;

}

public static void main(String[] args) {

MyContainerTwo container=new MyContainerTwo();

//制造10个消费者

for(int i=0;i<10;i++){

new Thread(()->{

for(int j=0;j<5;j++) System.out.println(container.get());

},

"c"+i).start();

}

try {

TimeUnit.SECONDS.sleep(2);

} catch (InterruptedException e) {

e.printStackTrace();

}

//制造2个生产者

for(int i=0;i<2;i++){

new Thread(()->{

for(int j=0;j<25;j++) container.put(j);

},

"p"+i).start();

}

}

}

[复制代码](javascript:void(0);)

总结：synchronized实现线程的消费者-生产者模式是通过wait/notifyAll实现，ReentrantLock是通过condition+await/signAll。那他们有什么区别呢？synchronized要么通过notify随机唤醒一个，或者notifyAll唤醒所有不管你是消费者还是生产者、而ReentrantLock是唤醒指定的线程的，更加精确效率更高。

# 四、版权声明

　　作者：邱勇Aaron

　　出处：<http://www.cnblogs.com/qiuyong/>

　　您的支持是对博主深入思考总结的最大鼓励。

　　本文版权归作者所有，欢迎转载，但未经作者同意必须保留此段声明，且在文章页面明显位置给出原文连接，尊重作者的劳动成果。

　　参考：马士兵并发编程、并发编程实践

　　　　　AQS详解：[http://www.cnblogs.com/waterystone/p/4920797.html](http://www.cnblogs.com/yezhenhan/archive/2012/01/07/2315652.html)

　　　　　CountDownLatch详解：<http://www.cnblogs.com/yezhenhan/archive/2012/01/07/2315652.html>

一直特立独行的二本僧，书写属于他的天空

分类: [Java并发编程](http://www.cnblogs.com/qiuyong/category/1023137.html)