

AQS应用之Lock

并发之父

ReentrantLock

AQS具备特性

同步等待队列

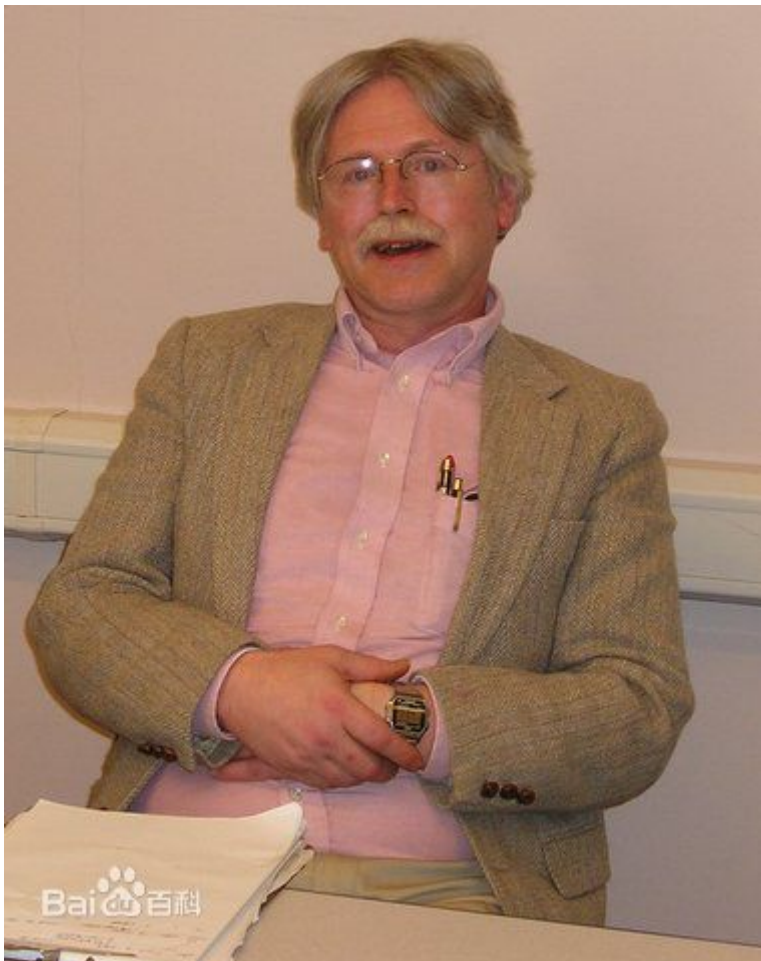
条件等待队列

AQS源码分析

AQS应用之Lock

并发之父

Doug Lea(小名: 李二狗)



生平不识Doug Lea, 学懂并发也枉然

Java并发编程核心在于java.concurrent.util包而juc当中的大多数同步器实现都是围绕着共同的基础行为，比如等待队列、条件队列、独占获取、共享获取等，而这个行为的抽象就是基于AbstractQueuedSynchronizer简称AQS，AQS定义了一套多线程访问共享资源的同步器框架，是一个依赖状态(state)的同步器。

ReentrantLock

ReentrantLock是一种基于AQS框架的应用实现，是JDK中的一种线程并发访问的同步手段，它的功能类似于synchronized是一种互斥锁，可以保证线程安全。而且它具有比synchronized更多的特性，比如它支持手动加锁与解锁，支持加锁的公平性。

```
1 使用ReentrantLock进行同步
2 ReentrantLock lock = new ReentrantLock(false); // false为非公平锁，true为公平锁
3 lock.lock() // 加锁
4 lock.unlock() // 解锁
```

ReentrantLock如何实现synchronized不具备的公平与非公平性呢？

在ReentrantLock内部定义了一个Sync的内部类，该类继承AbstractQueuedSynchronizer，对该抽象类的部分方法做了实现；并且还定义了两个子类：

1、FairSync 公平锁的实现

2、NonfairSync 非公平锁的实现

这两个类都继承自Sync，也就是间接继承了AbstractQueuedSynchronizer，所以这一个ReentrantLock同时具备公平与非公平特性。

上面主要涉及的设计模式：模板模式-子类根据需要做具体业务实现

AQS具备特性

- 阻塞等待队列
- 共享/独占
- 公平/非公平
- 可重入
- 允许中断

除了Lock外，Java.concurrent.util当中同步器的实现如Latch,Barrier,BlockingQueue等，都是基于AQS框架实现

- 一般通过定义内部类Sync继承AQS
- 将同步器所有调用都映射到Sync对应的方法

AQS内部维护属性**volatile int state (32位)**

- state表示资源的可用状态

State三种访问方式

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

- Share-共享，多个线程可以同时执行，如Semaphore/CountDownLatch

AQS定义两种队列

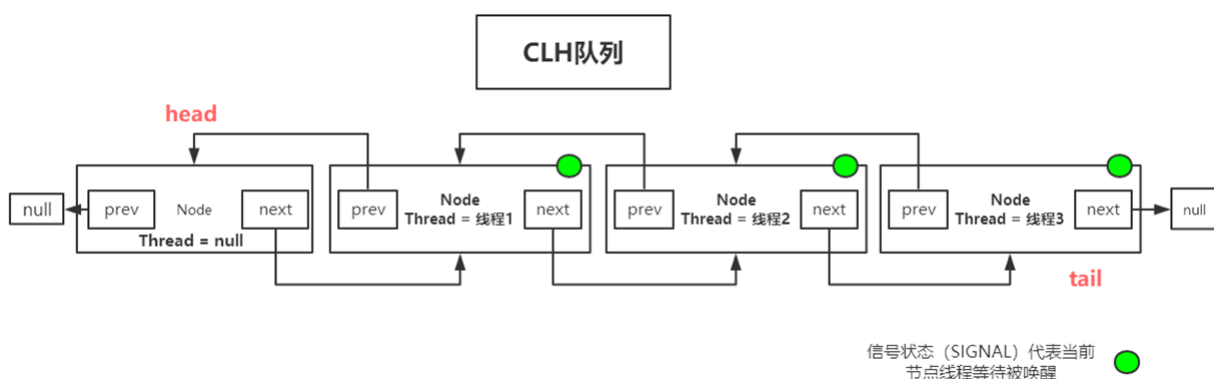
- 同步等待队列
- 条件等待队列

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，如果释放后允许唤醒后续等待结点返回true，否则返回false。

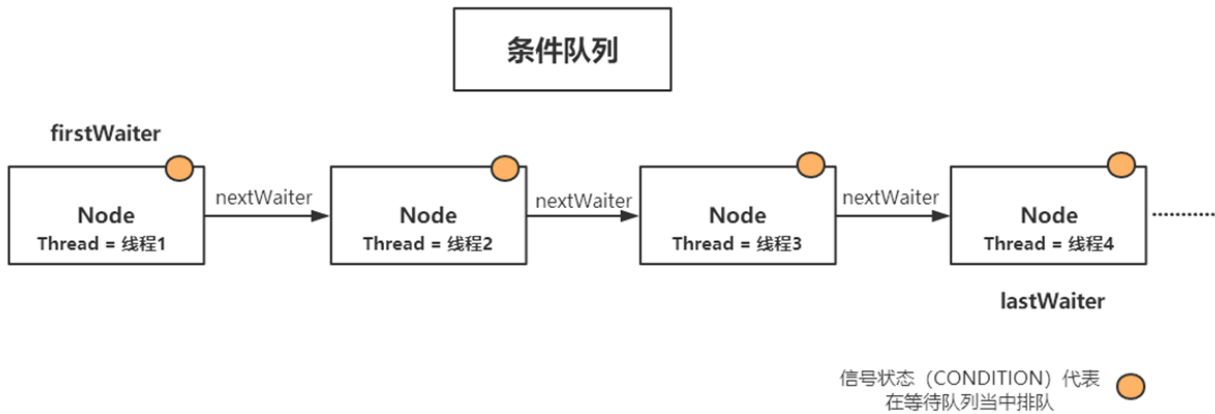
同步等待队列

AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagersten三人发明的一种基于双向链表数据结构的队列，是FIFO先入先出线程等待队列，Java中的CLH队列是原CLH队列的一个变种，线程由原自旋机制改为阻塞机制。



条件等待队列

Condition是一个多线程间协调通信的工具类，使得某个，或者某些线程一起等待某个条件（Condition），只有当该条件具备时，这些等待线程才会被唤醒，从而重新争夺锁



AQS源码分析

```

1 public abstract class AbstractQueuedSynchronizer
2     extends AbstractOwnableSynchronizer
3     implements java.io.Serializable {
4     private static final long serialVersionUID = 737398497257241469
5         1L;
6
7     /**
8      * Creates a new {@code AbstractQueuedSynchronizer} instance
9      * with initial synchronization state of zero.
10    */
11
12    protected AbstractQueuedSynchronizer() { }
13
14    /**
15     * Wait queue node class.
16     *
17     * 不管是条件队列，还是CLH等待队列
18     * 都是基于Node类
19     *
20     * AQS当中的同步等待队列也称CLH队列，CLH队列是Craig、Landin、Hagerson三人
21     * 发明的一种基于双向链表数据结构的队列，是FIFO先入先出线程等待队列，
22     * Java中的
23     * CLH队列是原CLH队列的一个变种，线程由原自旋机制改为阻塞机制。
24    */
25
26    static final class Node {
27
28        /**
29         * 标记节点未共享模式

```

```
25  * */
26  static final Node SHARED = new Node();
27  /**
28  * 标记节点为独占模式
29  */
30  static final Node EXCLUSIVE = null;
31
32  /**
33  * 在同步队列中等待的线程等待超时或者被中断，需要从同步队列中取消等待
34  * */
35  static final int CANCELLED = 1;
36  /**
37  * 后继节点的线程处于等待状态，而当前的节点如果释放了同步状态或者被取消，
38  * 将会通知后继节点，使后继节点的线程得以运行。
39  */
40  static final int SIGNAL = -1;
41  /**
42  * 节点在等待队列中，节点的线程等待在Condition上，当其他线程对Condition调用了signal()方法后，
43  * 该节点会从等待队列中转移到同步队列中，加入到同步状态的获取中
44  */
45  static final int CONDITION = -2;
46  /**
47  * 表示下一次共享式同步状态获取将会被无条件地传播下去
48  */
49  static final int PROPAGATE = -3;
50
51  /**
52  * 标记当前节点的信号量状态 (1,0,-1,-2,-3)5种状态
53  * 使用CAS更改状态，volatile保证线程可见性，高并发场景下，
54  * 即被一个线程修改后，状态会立马让其他线程可见。
55  */
56  volatile int waitStatus;
57
58  /**
```

```
59  * 前驱节点，当前节点加入到同步队列中被设置
60  */
61  volatile Node prev;
62
63  /**
64  * 后继节点
65  */
66  volatile Node next;
67
68  /**
69  * 节点同步状态的线程
70  */
71  volatile Thread thread;
72
73  /**
74  * 等待队列中的后继节点，如果当前节点是共享的，那么这个字段是一个SHAR
75  ED常量，
76  * 也就是说节点类型(独占和共享)和等待队列中的后继节点共用同一个字段。
77  */
78  Node nextWaiter;
79
80  /**
81  * Returns true if node is waiting in shared mode.
82  */
83  final boolean isShared() {
84      return nextWaiter == SHARED;
85  }
86
87  /**
88  * 返回前驱节点
89  */
90  final Node predecessor() throws NullPointerException {
91      Node p = prev;
92      if (p == null)
93          throw new NullPointerException();
94  }
```

```
93     else
94     return p;
95 }
96 //空节点，用于标记共享模式
97 Node() { // Used to establish initial head or SHARED marker
98 }
99 //用于同步队列CLH
100 Node(Thread thread, Node mode) { // Used by addWaiter
101     this.nextWaiter = mode;
102     this.thread = thread;
103 }
104 //用于条件队列
105 Node(Thread thread, int waitStatus) { // Used by Condition
106     this.waitStatus = waitStatus;
107     this.thread = thread;
108 }
109 }
110
111 /**
112  * 指向同步等待队列的头节点
113  */
114 private transient volatile Node head;
115
116 /**
117  * 指向同步等待队列的尾节点
118  */
119 private transient volatile Node tail;
120
121 /**
122  * 同步资源状态
123  */
124 private volatile int state;
125
126 /**
127  *
```

```

128  * @return current state value
129  */
130  protected final int getState() {
131      return state;
132  }
133
134  protected final void setState(int newState) {
135      state = newState;
136  }
137
138  /**
139   * Atomically sets synchronization state to the given updated
140   * value if the current state value equals the expected value.
141   * This operation has memory semantics of a {@code volatile} re
142   * ad
143   * and write.
144   *
145   * @param expect the expected value
146   * @param update the new value
147   * @return {@code true} if successful. False return indicates t
148   * hat the actual
149   * value was not equal to the expected value.
150   */
151  protected final boolean compareAndSetState(int expect, int up
152  ate) {
153      // See below for intrinsics setup to support this
154      return unsafe.compareAndSwapInt(this, stateOffset, expect, up
155  ate);
156  }
157
158  // Queuing utilities
159
160  /**
161   * The number of nanoseconds for which it is faster to spin
162   * rather than to use timed park. A rough estimate suffices
163   * to improve responsiveness with very short timeouts.

```



```

160  */
161  static final long spinForTimeoutThreshold = 1000L;
162
163  /**
164   * 节点加入CLH同步队列
165   */
166  private Node enq(final Node node) {
167      for (;;) {
168          Node t = tail;
169          if (t == null) { // Must initialize
170              //队列为空需要初始化，创建空的头节点
171              if (compareAndSetHead(new Node()))
172                  tail = head;
173          } else {
174              node.prev = t;
175              //set尾部节点
176              if (compareAndSetTail(t, node)) { //当前节点置为尾部
177                  t.next = node; //前驱节点的next指针指向当前节点
178              }
179          }
180      }
181  }
182  }
183
184  /**
185   * Creates and enqueues node for current thread and given mode
186   *
187   * @param mode Node.EXCLUSIVE for exclusive, Node.SHARED for shared
188   * @return the new node
189   */
190  private Node addWaiter(Node mode) {
191      // 1. 将当前线程构建成Node类型
192      Node node = new Node(Thread.currentThread(), mode);
193      // Try the fast path of enq; backup to full enq on failure

```

```
194 Node pred = tail;
195 // 2. 1当前尾节点是否为null?
196 if (pred != null) {
197     // 2.2 将当前节点尾插入的方式
198     node.prev = pred;
199     // 2.3 CAS将节点插入同步队列的尾部
200     if (compareAndSetTail(pred, node)) {
201         pred.next = node;
202         return node;
203     }
204 }
205 enq(node);
206 return node;
207 }
208
209 /**
210  * Sets head of queue to be node, thus dequeuing. Called only by
211  * acquire methods. Also nulls out unused fields for sake of GC
212  * and to suppress unnecessary signals and traversals.
213  *
214  * @param node the node
215  */
216 private void setHead(Node node) {
217     head = node;
218     node.thread = null;
219     node.prev = null;
220 }
221
222 /**
223  *
224  */
225 private void unparkSuccessor(Node node) {
226     //获取wait状态
227     int ws = node.waitStatus;
```

```

228     if (ws < 0)
229         compareAndSetWaitStatus(node, ws, 0); // 将等待状态waitStatus设置为初始值0
230
231     /**
232     * 若后继结点为空，或状态为CANCEL（已失效），则从后尾部往前遍历找到
    最前的一个处于正常阻塞状态的结点
233     * 进行唤醒
234     */
235     Node s = node.next; //head.next = Node1 ,thread = T3
236     if (s == null || s.waitStatus > 0) {
237         s = null;
238         for (Node t = tail; t != null && t != node; t = t.prev)
239             if (t.waitStatus <= 0)
240                 s = t;
241     }
242     if (s != null)
243         LockSupport.unpark(s.thread); //唤醒线程,T3唤醒
244     }
245
246     /**
247     * 把当前结点设置为SIGNAL或者PROPAGATE
248     * 唤醒head.next(B节点)，B节点唤醒后可以竞争锁，成功后head->B，然后
    又会唤醒B.next，一直重复直到共享节点都唤醒
249     * head节点状态为SIGNAL，重置head.waitStatus->0，唤醒head节点线
    程，唤醒后线程去竞争共享锁
250     * head节点状态为0，将head.waitStatus->Node.PROPAGATE传播状态，表
    示需要将状态向后继节点传播
251     */
252     private void doReleaseShared() {
253         for (;;) {
254             Node h = head;
255             if (h != null && h != tail) {
256                 int ws = h.waitStatus;
257                 if (ws == Node.SIGNAL) { //head是SIGNAL状态
258                     /* head状态是SIGNAL，重置head节点waitStatus为0，E这里不直接设为Node.PROPAGAT,

```

```

259  * 是因为unparkSuccessor(h)中, 如果ws < 0会设置为0, 所以ws先设置为
    0, 再设置为PROPAGATE
260  * 这里需要控制并发, 因为入口有setHeadAndPropagate跟release两个,
    避免两次unpark
261  */
262  if (!compareAndSetWaitStatus(h, Node.SIGNAL, 0))
263      continue; //设置失败, 重新循环
264  /* head状态为SIGNAL, 且成功设置为0之后, 唤醒head.next节点线程
265  * 此时head、head.next的线程都唤醒了, head.next会去竞争锁, 成功后h
    ead会指向获取锁的节点,
266  * 也就是head发生了变化。看最底下一行代码可知, head发生变化后会重新
    循环, 继续唤醒head的下一个节点
267  */
268  unparkSuccessor(h);
269  /*
270  * 如果本身头节点的waitStatus是出于重置状态 (waitStatus==0) 的, 将
    其设置为“传播”状态。
271  * 意味着需要将状态向后一个节点传播
272  */
273  }
274  else if (ws == 0 &&
275      !compareAndSetWaitStatus(h, 0, Node.PROPAGATE))
276      continue; // loop on failed CAS
277  }
278  if (h == head) //如果head变了, 重新循环
279      break;
280  }
281  }
282
283  /**
284  * 把node节点设置成head节点, 且Node.waitStatus->Node.PROPAGATE
285  */
286  private void setHeadAndPropagate(Node node, int propagate) {
287      Node h = head; //h用来保存旧的head节点
288      setHead(node); //head引用指向node节点
289      /* 这里意思有两种情况是需要执行唤醒操作

```

```

290  * 1.propagate > 0 表示调用方指明了后继节点需要被唤醒
291  * 2.头节点后面的节点需要被唤醒 (waitStatus<0)，不论是老的头结点还是新的头结点
292  */
293  if (propagate > 0 || h == null || h.waitStatus < 0 ||
294      (h = head) == null || h.waitStatus < 0) {
295      Node s = node.next;
296      if (s == null || s.isShared())//node是最后一个节点或者 node的后继节点是共享节点
297      /* 如果head节点状态为SIGNAL，唤醒head节点线程，重置head.waitStatus->0
298      * head节点状态为0(第一次添加时是0)，设置head.waitStatus->Node.PROPAGATE表示状态需要向后继节点传播
299      */
300      doReleaseShared();
301  }
302  }
303
304  // Utilities for various versions of acquire
305
306  /**
307   * 终结掉正在尝试去获取锁的节点
308   * @param node the node
309   */
310  private void cancelAcquire(Node node) {
311      // Ignore if node doesn't exist
312      if (node == null)
313          return;
314
315      node.thread = null;
316
317      // 剔除掉一件被cancel掉的节点
318      Node pred = node.prev;
319      while (pred.waitStatus > 0)
320          node.prev = pred = pred.prev;
321

```

```

322 // predNext is the apparent node to unsplice. CASes below will
323 // fail if not, in which case, we lost race vs another cancel
324 // or signal, so no further action is necessary.
325 Node predNext = pred.next;
326
327 // Can use unconditional write instead of CAS here.
328 // After this atomic step, other Nodes can skip past us.
329 // Before, we are free of interference from other threads.
330 node.waitStatus = Node.CANCELLED;
331
332 // If we are the tail, remove ourselves.
333 if (node == tail && compareAndSetTail(node, pred)) {
334     compareAndSetNext(pred, predNext, null);
335 } else {
336     // If successor needs signal, try to set pred's next-link
337     // so it will get one. Otherwise wake it up to propagate.
338     int ws;
339     if (pred != head &&
340         ((ws = pred.waitStatus) == Node.SIGNAL ||
341          (ws <= 0 && compareAndSetWaitStatus(pred, ws, Node.SIGNAL))) &
342         pred.thread != null) {
343         Node next = node.next;
344         if (next != null && next.waitStatus <= 0)
345             compareAndSetNext(pred, predNext, next);
346         } else {
347             unparkSuccessor(node);
348         }
349
350     node.next = node; // help GC
351 }
352 }
353
354 /**
355  *

```

```
356  */
357  private static boolean shouldParkAfterFailedAcquire(Node pred,
Node node) {
358  int ws = pred.waitStatus;
359  if (ws == Node.SIGNAL)
360  /*
361   * 若前驱结点的状态是SIGNAL，意味着当前结点可以被安全地park
362   */
363  return true;
364  if (ws > 0) {
365  /*
366   * 前驱节点状态如果被取消状态，将被移除出队列
367   */
368  do {
369  node.prev = pred = pred.prev;
370  } while (pred.waitStatus > 0);
371  pred.next = node;
372  } else {
373  /*
374   * 当前驱节点waitStatus为 0 or PROPAGATE状态时
375   * 将其设置为SIGNAL状态，然后当前结点才可以可以被安全地park
376   */
377  compareAndSetWaitStatus(pred, ws, Node.SIGNAL);
378  }
379  return false;
380  }
381
382  /**
383   * 中断当前线程
384   */
385  static void selfInterrupt() {
386  Thread.currentThread().interrupt();
387  }
388
389  /**
```

```

390  * 阻塞当前节点，返回当前Thread的中断状态
391  * LockSupport.park 底层实现逻辑调用系统内核功能 pthread_mutex_lo
ck 阻塞线程
392  */
393  private final boolean parkAndCheckInterrupt() {
394      LockSupport.park(this); //阻塞
395      return Thread.interrupted();
396  }
397
398  /**
399  * 已经在队列当中的Thread节点，准备阻塞等待获取锁
400  */
401  final boolean acquireQueued(final Node node, int arg) {
402      boolean failed = true;
403      try {
404          boolean interrupted = false;
405          for (;;) { //死循环
406              final Node p = node.predecessor(); //找到当前结点的前驱结点
407              if (p == head && tryAcquire(arg)) { //如果前驱结点是头结点，才try
Acquire，其他结点是没有机会tryAcquire的。
408                  setHead(node); //获取同步状态成功，将当前结点设置为头结点。
409                  p.next = null; // help GC
410                  failed = false;
411                  return interrupted;
412              }
413              /**
414              * 如果前驱节点不是Head，通过shouldParkAfterFailedAcquire判断是否
应该阻塞
415              * 前驱节点信号量为-1，当前线程可以安全被parkAndCheckInterrupt用来
阻塞线程
416              */
417              if (shouldParkAfterFailedAcquire(p, node) &&
418                  parkAndCheckInterrupt())
419                  interrupted = true;
420              }
421          } finally {

```



```
422     if (failed)
423         cancelAcquire(node);
424     }
425 }
426
427 /**
428  * 与acquireQueued逻辑相似，唯一区别节点还不在队列当中需要先进行入
    队操作
429  */
430 private void doAcquireInterruptibly(int arg)
431     throws InterruptedException {
432     final Node node = addWaiter(Node.EXCLUSIVE); //以独占模式放入队
    列尾部
433     boolean failed = true;
434     try {
435         for (;;) {
436             final Node p = node.predecessor();
437             if (p == head && tryAcquire(arg)) {
438                 setHead(node);
439                 p.next = null; // help GC
440                 failed = false;
441                 return;
442             }
443             if (shouldParkAfterFailedAcquire(p, node) &&
444                 parkAndCheckInterrupt())
445                 throw new InterruptedException();
446         }
447     } finally {
448         if (failed)
449             cancelAcquire(node);
450     }
451 }
452
453 /**
454  * 独占模式定时获取
455  */
```

```
456 private boolean doAcquireNanos(int arg, long nanosTimeout)
457 throws InterruptedException {
458     if (nanosTimeout <= 0L)
459         return false;
460     final long deadline = System.nanoTime() + nanosTimeout;
461     final Node node = addWaiter(Node.EXCLUSIVE); //加入队列
462     boolean failed = true;
463     try {
464         for (;;) {
465             final Node p = node.predecessor();
466             if (p == head && tryAcquire(arg)) {
467                 setHead(node);
468                 p.next = null; // help GC
469                 failed = false;
470                 return true;
471             }
472             nanosTimeout = deadline - System.nanoTime();
473             if (nanosTimeout <= 0L)
474                 return false; //超时直接返回获取失败
475             if (shouldParkAfterFailedAcquire(p, node) &&
476                 nanosTimeout > spinForTimeoutThreshold)
477                 //阻塞指定时长，超时则线程自动被唤醒
478                 LockSupport.parkNanos(this, nanosTimeout);
479             if (Thread.interrupted()) //当前线程中断状态
480                 throw new InterruptedException();
481         }
482     } finally {
483         if (failed)
484             cancelAcquire(node);
485     }
486 }
487
488 /**
489  * 尝试获取共享锁
490  */
```

```
491 private void doAcquireShared(int arg) {
492     final Node node = addWaiter(Node.SHARED); // 入队
493     boolean failed = true;
494     try {
495         boolean interrupted = false;
496         for (;;) {
497             final Node p = node.predecessor(); // 前驱节点
498             if (p == head) {
499                 int r = tryAcquireShared(arg); // 非公平锁实现，再尝试获取锁
500                 // state==0时tryAcquireShared会返回>=0(CountDownLatch中返回的是
501                 // 1)。
502                 // state为0说明共享次数已经到了，可以获取锁了
503                 if (r >= 0) { // r>0表示state==0,前继节点已经释放锁，锁的状态为可被
504                     获取
505                     // 这一步设置node为head节点设置node.waitStatus->Node.PROPAGATE,
506                     然后唤醒node.thread
507                     setHeadAndPropagate(node, r);
508                     p.next = null; // help GC
509                     if (interrupted)
510                         selfInterrupt();
511                     failed = false;
512                     return;
513                 }
514             }
515             // 前继节点非head节点，将前继节点状态设置为SIGNAL，通过park挂起node
516             // 节点的线程
517             if (shouldParkAfterFailedAcquire(p, node) &&
518                 parkAndCheckInterrupt())
519                 interrupted = true;
520         }
521     } finally {
522         if (failed)
523             cancelAcquire(node);
524     }
525 }
```

```
523  /**
524   * Acquires in shared interruptible mode.
525   * @param arg the acquire argument
526   */
527  private void doAcquireSharedInterruptibly(int arg)
528  throws InterruptedException {
529      final Node node = addWaiter(Node.SHARED);
530      boolean failed = true;
531      try {
532          for (;;) {
533              final Node p = node.predecessor();
534              if (p == head) {
535                  int r = tryAcquireShared(arg);
536                  if (r >= 0) {
537                      setHeadAndPropagate(node, r);
538                      p.next = null; // help GC
539                      failed = false;
540                      return;
541                  }
542              }
543              if (shouldParkAfterFailedAcquire(p, node) &&
544                  parkAndCheckInterrupt())
545                  throw new InterruptedException();
546          }
547      } finally {
548          if (failed)
549              cancelAcquire(node);
550      }
551  }
552
553  /**
554   * Acquires in shared timed mode.
555   *
556   * @param arg the acquire argument
557   * @param nanosTimeout max wait time
```

```
558 * @return {@code true} if acquired
559 */
560 private boolean doAcquireSharedNanos(int arg, long nanosTimeout)
561     throws InterruptedException {
562     if (nanosTimeout <= 0L)
563         return false;
564     final long deadline = System.nanoTime() + nanosTimeout;
565     final Node node = addWaiter(Node.SHARED);
566     boolean failed = true;
567     try {
568         for (;;) {
569             final Node p = node.predecessor();
570             if (p == head) {
571                 int r = tryAcquireShared(arg);
572                 if (r >= 0) {
573                     setHeadAndPropagate(node, r);
574                     p.next = null; // help GC
575                     failed = false;
576                     return true;
577                 }
578             }
579             nanosTimeout = deadline - System.nanoTime();
580             if (nanosTimeout <= 0L)
581                 return false;
582             if (shouldParkAfterFailedAcquire(p, node) &&
583                 nanosTimeout > spinForTimeoutThreshold)
584                 LockSupport.parkNanos(this, nanosTimeout);
585             if (Thread.interrupted())
586                 throw new InterruptedException();
587         }
588     } finally {
589         if (failed)
590             cancelAcquire(node);
591     }
```

```
592 }
593
594 // Main exported methods
595
596 /**
597  * 尝试获取独占锁，可指定锁的获取数量
598  */
599 protected boolean tryAcquire(int arg) {
600     throw new UnsupportedOperationException();
601 }
602
603 /**
604  * 尝试释放独占锁，在子类当中实现
605  */
606 protected boolean tryRelease(int arg) {
607     throw new UnsupportedOperationException();
608 }
609
610 /**
611  * 共享式：共享式地获取同步状态。对于独占式同步组件来讲，同一时刻只有一个线程能获取到同步状态，
612  * 其他线程都得去排队等待，其待重写的尝试获取同步状态的方法tryAcquire返回值为boolean，这很容易理解；
613  * 对于共享式同步组件来讲，同一时刻可以有多个线程同时获取到同步状态，这也是“共享”的意义所在。
614  * 本方法待被子类覆盖实现具体逻辑
615  * 1.当返回值大于0时，表示获取同步状态成功，同时还有剩余同步状态可供其他线程获取；
616  *
617  * 2.当返回值等于0时，表示获取同步状态成功，但没有可用同步状态了；
618
619  * 3.当返回值小于0时，表示获取同步状态失败。
620  */
621 protected int tryAcquireShared(int arg) {
622     throw new UnsupportedOperationException();
623 }
```

```
624
625  /**
626  * 释放共享锁，具体实现在子类当中实现
627  */
628  protected boolean tryReleaseShared(int arg) {
629  throw new UnsupportedOperationException();
630  }
631
632  /**
633  * 当前线程是否持有独占锁
634  */
635  protected boolean isHeldExclusively() {
636  throw new UnsupportedOperationException();
637  }
638
639  /**
640  * 获取独占锁
641  */
642  public final void acquire(int arg) {
643  //尝试获取锁
644  if (!tryAcquire(arg) &&
645  acquireQueued(addWaiter(Node.EXCLUSIVE), arg))//独占模式
646  selfInterrupt();
647  }
648
649  /**
650  *
651  */
652  public final void acquireInterruptibly(int arg)
653  throws InterruptedException {
654  if (Thread.interrupted())
655  throw new InterruptedException();
656  if (!tryAcquire(arg))
657  doAcquireInterruptibly(arg);
658  }
```

```
659
660  /**
661   * 获取独占锁，设置最大等待时间
662   */
663   public final boolean tryAcquireNanos(int arg, long nanosTimeout)
664   throws InterruptedException {
665     if (Thread.interrupted())
666       throw new InterruptedException();
667     return tryAcquire(arg) ||
668            doAcquireNanos(arg, nanosTimeout);
669   }
670
671  /**
672   * 释放独占模式持有的锁
673   */
674   public final boolean release(int arg) {
675     if (tryRelease(arg)) { // 释放一次锁
676       Node h = head;
677       if (h != null && h.waitStatus != 0)
678         unparkSuccessor(h); // 唤醒后继结点
679       return true;
680     }
681     return false;
682   }
683
684  /**
685   * 请求获取共享锁
686   */
687   public final void acquireShared(int arg) {
688     if (tryAcquireShared(arg) < 0) // 返回值小于0，获取同步状态失败，挂
689     队去；获取同步状态成功，直接返回去干自己的事儿。
689     doAcquireShared(arg);
690   }
691
692
```



```

693  /**
694  * Releases in shared mode. Implemented by unblocking one or mo
695  * threads if {@link #tryReleaseShared} returns true.
696  *
697  * @param arg the release argument. This value is conveyed to
698  * {@link #tryReleaseShared} but is otherwise uninterpreted
699  * and can represent anything you like.
700  * @return the value returned from {@link #tryReleaseShared}
701  */
702  public final boolean releaseShared(int arg) {
703      if (tryReleaseShared(arg)) {
704          doReleaseShared();
705          return true;
706      }
707      return false;
708  }
709
710  // Queue inspection methods
711
712  public final boolean hasQueuedThreads() {
713      return head != tail;
714  }
715
716  public final boolean hasContended() {
717      return head != null;
718  }
719
720  public final Thread getFirstQueuedThread() {
721      // handle only fast path, else relay
722      return (head == tail) ? null : fullGetFirstQueuedThread();
723  }
724
725  /**
726  * Version of getFirstQueuedThread called when fastpath fails

```

```
727  */
728  private Thread fullGetFirstQueuedThread() {
729      Node h, s;
730      Thread st;
731      if (((h = head) != null && (s = h.next) != null &&
732          s.prev == head && (st = s.thread) != null) ||
733          ((h = head) != null && (s = h.next) != null &&
734          s.prev == head && (st = s.thread) != null))
735          return st;
736
737      Node t = tail;
738      Thread firstThread = null;
739      while (t != null && t != head) {
740          Thread tt = t.thread;
741          if (tt != null)
742              firstThread = tt;
743          t = t.prev;
744      }
745      return firstThread;
746  }
747
748  /**
749   * 判断当前线程是否在队列当中
750   */
751  public final boolean isQueued(Thread thread) {
752      if (thread == null)
753          throw new NullPointerException();
754      for (Node p = tail; p != null; p = p.prev)
755          if (p.thread == thread)
756              return true;
757      return false;
758  }
759
760  final boolean apparentlyFirstQueuedIsExclusive() {
761      Node h, s;
```

```
762     return (h = head) != null &&
763     (s = h.next) != null &&
764     !s.isShared() &&
765     s.thread != null;
766 }
767
768 /**
769  * 判断当前节点是否有前驱节点
770  */
771 public final boolean hasQueuedPredecessors() {
772     Node t = tail; // Read fields in reverse initialization order
773     Node h = head;
774     Node s;
775     return h != t &&
776     ((s = h.next) == null || s.thread != Thread.currentThread());
777 }
778
779
780 // Instrumentation and monitoring methods
781
782 /**
783  * 同步队列长度
784  */
785 public final int getQueueLength() {
786     int n = 0;
787     for (Node p = tail; p != null; p = p.prev) {
788         if (p.thread != null)
789             ++n;
790     }
791     return n;
792 }
793
794 /**
795  * 获取队列等待thread集合
796  */
```

```
797 public final Collection<Thread> getQueuedThreads() {
798     ArrayList<Thread> list = new ArrayList<Thread>();
799     for (Node p = tail; p != null; p = p.prev) {
800         Thread t = p.thread;
801         if (t != null)
802             list.add(t);
803     }
804     return list;
805 }
806
807 /**
808  * 获取独占模式等待thread线程集合
809  */
810 public final Collection<Thread> getExclusiveQueuedThreads() {
811     ArrayList<Thread> list = new ArrayList<Thread>();
812     for (Node p = tail; p != null; p = p.prev) {
813         if (!p.isShared()) {
814             Thread t = p.thread;
815             if (t != null)
816                 list.add(t);
817         }
818     }
819     return list;
820 }
821
822 /**
823  * 获取共享模式等待thread集合
824  */
825 public final Collection<Thread> getSharedQueuedThreads() {
826     ArrayList<Thread> list = new ArrayList<Thread>();
827     for (Node p = tail; p != null; p = p.prev) {
828         if (p.isShared()) {
829             Thread t = p.thread;
830             if (t != null)
831                 list.add(t);
```

```
832 }
833 }
834 return list;
835 }
836
837
838 // Internal support methods for Conditions
839
840 /**
841  * 判断节点是否在同步队列中
842  */
843 final boolean isOnSyncQueue(Node node) {
844     //快速判断1: 节点状态或者节点没有前置节点
845     //注: 同步队列是有头节点的, 而条件队列没有
846     if (node.waitStatus == Node.CONDITION || node.prev == null)
847         return false;
848     //快速判断2: next字段只有同步队列才会使用, 条件队列中使用的是nextWaiter
849     //iter字段
850     if (node.next != null) // If has successor, it must be on queue
851         return true;
852     //上面如果无法判断则进入复杂判断
853     return findNodeFromTail(node);
854 }
855
856 private boolean findNodeFromTail(Node node) {
857     Node t = tail;
858     for (;;) {
859         if (t == node)
860             return true;
861         if (t == null)
862             return false;
863         t = t.prev;
864     }
865 }
```

```
866  /**
867  * 将节点从条件队列当中移动到同步队列当中，等待获取锁
868  */
869  final boolean transferForSignal(Node node) {
870  /**
871  * 修改节点信号量状态为0，失败直接返回false
872  */
873  if (!compareAndSetWaitStatus(node, Node.CONDITION, 0))
874  return false;
875
876  /**
877  * 加入同步队列尾部当中，返回前驱节点
878  */
879  Node p = enq(node);
880  int ws = p.waitStatus;
881  //前驱节点不可用 或者 修改信号量状态失败
882  if (ws > 0 || !compareAndSetWaitStatus(p, ws, Node.SIGNAL))
883  LockSupport.unpark(node.thread); //唤醒当前节点
884  return true;
885  }
886
887  final boolean transferAfterCancelledWait(Node node) {
888  if (compareAndSetWaitStatus(node, Node.CONDITION, 0)) {
889  enq(node);
890  return true;
891  }
892  /**
893  * If we lost out to a signal(), then we can't proceed
894  * until it finishes its enq(). Cancelling during an
895  * incomplete transfer is both rare and transient, so just
896  * spin.
897  */
898  while (!isOnSyncQueue(node))
899  Thread.yield();
900  return false;
```

```
901 }
902
903 /**
904  * 入参就是新创建的节点，即当前节点
905  */
906 final int fullyRelease(Node node) {
907     boolean failed = true;
908     try {
909         //这里这个取值要注意，获取当前的state并释放，这从另一个角度说明必须是独占锁
910         //可以考虑下这个逻辑放在共享锁下面会发生什么？
911         int savedState = getState();
912         if (release(savedState)) {
913             failed = false;
914             return savedState;
915         } else {
916             //如果这里释放失败，则抛出异常
917             throw new IllegalMonitorStateException();
918         }
919     } finally {
920         /**
921          * 如果释放锁失败，则把节点取消，由这里就能看出来上面添加节点的逻辑中
922          * 只需要判断最后一个节点是否被取消就可以了
923          */
924         if (failed)
925             node.waitStatus = Node.CANCELLED;
926     }
927 }
928
929 // Instrumentation methods for conditions
930
931 public final boolean hasWaiters(ConditionObject condition) {
932     if (!owns(condition))
933         throw new IllegalArgumentException("Not owner");
934     return condition.hasWaiters();
935 }
```

```
935 }
936
937 /**
938  * 获取条件队列长度
939  */
940 public final int getWaitQueueLength(ConditionObject condition)
941 {
942     if (!owns(condition))
943         throw new IllegalArgumentException("Not owner");
944     return condition.getWaitQueueLength();
945 }
946
947 /**
948  * 获取条件队列当中所有等待的thread集合
949  */
950 public final Collection<Thread> getWaitingThreads(ConditionObject condition) {
951     if (!owns(condition))
952         throw new IllegalArgumentException("Not owner");
953     return condition.getWaitingThreads();
954 }
955
956 /**
957  * 条件对象，实现基于条件的具体行为
958  */
959 public class ConditionObject implements Condition, java.io.Serializable {
960     private static final long serialVersionUID = 1173984872572414699L;
961     /** First node of condition queue. */
962     private transient Node firstWaiter;
963     /** Last node of condition queue. */
964     private transient Node lastWaiter;
965
966     /**
967      * Creates a new {@code ConditionObject} instance.
```



```

967  */
968  public ConditionObject() { }
969
970  // Internal methods
971
972  /**
973   * 1.与同步队列不同，条件队列头尾指针是firstWaiter跟lastWaiter
974   * 2.条件队列是在获取锁之后，也就是临界区进行操作，因此很多地方不用考虑并发
975   */
976  private Node addConditionWaiter() {
977      Node t = lastWaiter;
978      //如果最后一个节点被取消，则删除队列中被取消的节点
979      //至于为啥是最后一个节点后面会分析
980      if (t != null && t.waitStatus != Node.CONDITION) {
981          //删除所有被取消的节点
982          unlinkCancelledWaiters();
983          t = lastWaiter;
984      }
985      //创建一个类型为CONDITION的节点并加入队列，由于在临界区，所以这里不用并发控制
986      Node node = new Node(Thread.currentThread(), Node.CONDITION);
987      if (t == null)
988          firstWaiter = node;
989      else
990          t.nextWaiter = node;
991      lastWaiter = node;
992      return node;
993  }
994
995  /**
996   * 发信号，通知遍历条件队列当中的节点转移到同步队列当中，准备排队获取锁
997   */
998  private void doSignal(Node first) {
999      do {

```

```
1000     if ( (firstWaiter = first.nextWaiter) == null)
1001         lastWaiter = null;
1002     first.nextWaiter = null;
1003 } while (!transferForSignal(first) && //转移节点
1004 (first = firstWaiter) != null);
1005 }
1006
1007 /**
1008  * 通知所有节点移动到同步队列当中，并将节点从条件队列删除
1009  */
1010 private void doSignalAll(Node first) {
1011     lastWaiter = firstWaiter = null;
1012     do {
1013         Node next = first.nextWaiter;
1014         first.nextWaiter = null;
1015         transferForSignal(first);
1016         first = next;
1017     } while (first != null);
1018 }
1019
1020 /**
1021  * 删除条件队列当中被取消的节点
1022  */
1023 private void unlinkCancelledWaiters() {
1024     Node t = firstWaiter;
1025     Node trail = null;
1026     while (t != null) {
1027         Node next = t.nextWaiter;
1028         if (t.waitStatus != Node.CONDITION) {
1029             t.nextWaiter = null;
1030             if (trail == null)
1031                 firstWaiter = next;
1032             else
1033                 trail.nextWaiter = next;
1034             if (next == null)
```

```
1035     lastWaiter = trail;
1036 }
1037 else
1038     trail = t;
1039     t = next;
1040 }
1041 }
1042
1043 // public methods
1044
1045 /**
1046  * 发新号，通知条件队列当中节点到同步队列当中去排队
1047  */
1048 public final void signal() {
1049     if (!isHeldExclusively())//节点不能已经持有独占锁
1050         throw new IllegalMonitorStateException();
1051     Node first = firstWaiter;
1052     if (first != null)
1053         /**
1054          * 发信号通知条件队列的节点准备到同步队列当中去排队
1055          */
1056         doSignal(first);
1057 }
1058
1059 /**
1060  * 唤醒所有条件队列的节点转移到同步队列当中
1061  */
1062 public final void signalAll() {
1063     if (!isHeldExclusively())
1064         throw new IllegalMonitorStateException();
1065     Node first = firstWaiter;
1066     if (first != null)
1067         doSignalAll(first);
1068 }
1069
```

```
1070  /**
1071  * Implements uninterruptible condition wait.
1072  * <ol>
1073  * <li> Save lock state returned by {@link #getState}.
1074  * <li> Invoke {@link #release} with saved state as argument,
1075  * throwing IllegalMonitorStateException if it fails.
1076  * <li> Block until signalled.
1077  * <li> Reacquire by invoking specialized version of
1078  * {@link #acquire} with saved state as argument.
1079  * </ol>
1080  */
1081  public final void awaitUninterruptibly() {
1082      Node node = addConditionWaiter();
1083      int savedState = fullyRelease(node);
1084      boolean interrupted = false;
1085      while (!isOnSyncQueue(node)) {
1086          LockSupport.park(this);
1087          if (Thread.interrupted())
1088              interrupted = true;
1089      }
1090      if (acquireQueued(node, savedState) || interrupted)
1091          selfInterrupt();
1092  }
1093
1094  /** 该模式表示在退出等待时重新中断 */
1095  private static final int REINTERRUPT = 1;
1096  /** 异常中断 */
1097  private static final int THROW_IE = -1;
1098
1099  /**
1100  * 这里的判断逻辑是：
1101  * 1.如果现在不是中断的，即正常被signal唤醒则返回0
1102  * 2.如果节点由中断加入同步队列则返回THROW_IE，由signal加入同步队
1103  * 列则返回REINTERRUPT
1104  */
```

```

1104 private int checkInterruptWhileWaiting(Node node) {
1105     return Thread.interrupted() ?
1106         (transferAfterCancelledWait(node) ? THROW_IE : REINTERRUPT) :
1107         0;
1108 }
1109
1110 /**
1111  * 根据中断时机选择抛出异常或者设置线程中断状态
1112  */
1113 private void reportInterruptAfterWait(int interruptMode)
1114     throws InterruptedException {
1115     if (interruptMode == THROW_IE)
1116         throw new InterruptedException();
1117     else if (interruptMode == REINTERRUPT)
1118         selfInterrupt();
1119 }
1120
1121 /**
1122  * 加入条件队列等待，条件队列入口
1123  */
1124 public final void await() throws InterruptedException {
1125
1126     //T2进来
1127     //如果当前线程被中断则直接抛出异常
1128     if (Thread.interrupted())
1129         throw new InterruptedException();
1130     //把当前节点加入条件队列
1131     Node node = addConditionWaiter();
1132     //释放掉已经获取的独占锁资源
1133     int savedState = fullyRelease(node); //T2释放锁
1134     int interruptMode = 0;
1135     //如果不在同步队列中则不断挂起
1136     while (!isOnSyncQueue(node)) {
1137         LockSupport.park(this); //T1被阻塞
1138         //这里被唤醒可能是正常的signal操作也可能是中断

```

```

1139     if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
1140         break;
1141     }
1142     /**
1143     * 走到这里说明节点已经条件满足被加入到了同步队列中或者中断了
1144     * 这个方法很熟悉吧？就跟独占锁调用同样的获取锁方法，从这里可以看出
    条件队列只能用于独占锁
1145     * 在处理中断之前首先要做的是从同步队列中成功获取锁资源
1146     */
1147     if (acquireQueued(node, savedState) && interruptMode != THRO
        _IE)
1148         interruptMode = REINTERRUPT;
1149     //走到这里说明已经成功获取到了独占锁，接下来就做些收尾工作
1150     //删除条件队列中被取消的节点
1151     if (node.nextWaiter != null) // clean up if cancelled
1152         unlinkCancelledWaiters();
1153     //根据不同模式处理中断
1154     if (interruptMode != 0)
1155         reportInterruptAfterWait(interruptMode);
1156     }
1157
1158
1159     /**
1160     * Implements timed condition wait.
1161     * <ol>
1162     * <li> If current thread is interrupted, throw InterruptedException.
1163     * <li> Save lock state returned by {@link #getState}.
1164     * <li> Invoke {@link #release} with saved state as argument,
1165     * throwing IllegalMonitorStateException if it fails.
1166     * <li> Block until signalled, interrupted, or timed out.
1167     * <li> Reacquire by invoking specialized version of
1168     * {@link #acquire} with saved state as argument.
1169     * <li> If interrupted while blocked in step 4, throw Interrupt
        tedException.

```

```
1170  * <li> If timed out while blocked in step 4, return false, else true.
1171  * </ol>
1172  */
1173  public final boolean await(long time, TimeUnit unit)
1174  throws InterruptedException {
1175      long nanosTimeout = unit.toNanos(time);
1176      if (Thread.interrupted())
1177          throw new InterruptedException();
1178      Node node = addConditionWaiter();
1179      int savedState = fullyRelease(node);
1180      final long deadline = System.nanoTime() + nanosTimeout;
1181      boolean timedout = false;
1182      int interruptMode = 0;
1183      while (!isOnSyncQueue(node)) {
1184          if (nanosTimeout <= 0L) {
1185              timedout = transferAfterCancelledWait(node);
1186              break;
1187          }
1188          if (nanosTimeout >= spinForTimeoutThreshold)
1189              LockSupport.parkNanos(this, nanosTimeout);
1190          if ((interruptMode = checkInterruptWhileWaiting(node)) != 0)
1191              break;
1192          nanosTimeout = deadline - System.nanoTime();
1193      }
1194      if (acquireQueued(node, savedState) && interruptMode != THROWING
1195          _IE)
1196          interruptMode = REINTERRUPT;
1197      if (node.nextWaiter != null)
1198          unlinkCancelledWaiters();
1199      if (interruptMode != 0)
1200          reportInterruptAfterWait(interruptMode);
1201      return !timedout;
1202  }
1203
```

```

1204     final boolean isOwnedBy(AbstractQueuedSynchronizer sync) {
1205         return sync == AbstractQueuedSynchronizer.this;
1206     }
1207
1208     /**
1209      * Queries whether any threads are waiting on this condition.
1210      * Implements {@link AbstractQueuedSynchronizer#hasWaiters(Con
1211      * ditionObject)}.
1212      *
1213      * @return {@code true} if there are any waiting threads
1214      * @throws IllegalMonitorStateException if {@link #isHeldExclu
1215      sively}
1216      * returns {@code false}
1217      */
1218     protected final boolean hasWaiters() {
1219         if (!isHeldExclusively())
1220             throw new IllegalMonitorStateException();
1221         for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
1222             if (w.waitStatus == Node.CONDITION)
1223                 return true;
1224         }
1225         return false;
1226     }
1227
1228     /**
1229      * Returns an estimate of the number of threads waiting on
1230      * this condition.
1231      * Implements {@link AbstractQueuedSynchronizer#getWaitQueueLe
1232      ngth(ConditionObject)}.
1233      *
1234      * @return the estimated number of waiting threads
1235      * @throws IllegalMonitorStateException if {@link #isHeldExclu
1236      sively}
1237      * returns {@code false}
1238      */
1239     protected final int getWaitQueueLength() {

```



```

1236     if (!isHeldExclusively())
1237         throw new IllegalMonitorStateException();
1238     int n = 0;
1239     for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
1240         if (w.waitStatus == Node.CONDITION)
1241             ++n;
1242     }
1243     return n;
1244 }
1245
1246 /**
1247  * 得到同步队列当中所有在等待的Thread集合
1248  */
1249 protected final Collection<Thread> getWaitingThreads() {
1250     if (!isHeldExclusively())
1251         throw new IllegalMonitorStateException();
1252     ArrayList<Thread> list = new ArrayList<Thread>();
1253     for (Node w = firstWaiter; w != null; w = w.nextWaiter) {
1254         if (w.waitStatus == Node.CONDITION) {
1255             Thread t = w.thread;
1256             if (t != null)
1257                 list.add(t);
1258         }
1259     }
1260     return list;
1261 }
1262 }
1263
1264 /**
1265  * Setup to support compareAndSet. We need to natively implement
1266  * this here: For the sake of permitting future enhancements,
1267  * we
1268  * cannot explicitly subclass AtomicInteger, which would be
1269  * efficient and useful otherwise. So, as the lesser of evils,
1270  * we

```

```
1269  * natively implement using hotspot intrinsics API. And while
we
1270  * are at it, we do the same for other CASable fields (which could
1271  * otherwise be done with atomic field updaters).
1272  * unsafe魔法类，直接绕过虚拟机内存管理机制，修改内存
1273  */
1274  private static final Unsafe unsafe = Unsafe.getUnsafe();
1275  //偏移量
1276  private static final long stateOffset;
1277  private static final long headOffset;
1278  private static final long tailOffset;
1279  private static final long waitStatusOffset;
1280  private static final long nextOffset;
1281
1282  static {
1283      try {
1284          //状态偏移量
1285          stateOffset = unsafe.objectFieldOffset
1286              (AbstractQueuedSynchronizer.class.getDeclaredField("state"));
1287          //head指针偏移量，head指向CLH队列的头部
1288          headOffset = unsafe.objectFieldOffset
1289              (AbstractQueuedSynchronizer.class.getDeclaredField("head"));
1290          tailOffset = unsafe.objectFieldOffset
1291              (AbstractQueuedSynchronizer.class.getDeclaredField("tail"));
1292          waitStatusOffset = unsafe.objectFieldOffset
1293              (Node.class.getDeclaredField("waitStatus"));
1294          nextOffset = unsafe.objectFieldOffset
1295              (Node.class.getDeclaredField("next"));
1296
1297      } catch (Exception ex) { throw new Error(ex); }
1298  }
1299
1300  /**
1301  * CAS 修改头部节点指向。并发入队时使用。
1302  */
```

```
1303     private final boolean compareAndSetHead(Node update) {
1304         return unsafe.compareAndSwapObject(this, headOffset, null, update);
1305     }
1306
1307     /**
1308      * CAS 修改尾部节点指向。并发入队时使用。
1309      */
1310     private final boolean compareAndSetTail(Node expect, Node update) {
1311         return unsafe.compareAndSwapObject(this, tailOffset, expect, update);
1312     }
1313
1314     /**
1315      * CAS 修改信号量状态。
1316      */
1317     private static final boolean compareAndSetWaitStatus(Node node,
1318         int expect,
1319         int update) {
1320         return unsafe.compareAndSwapInt(node, waitStatusOffset,
1321             expect, update);
1322     }
1323
1324     /**
1325      * 修改节点的后继指针。
1326      */
1327     private static final boolean compareAndSetNext(Node node,
1328         Node expect,
1329         Node update) {
1330         return unsafe.compareAndSwapObject(node, nextOffset, expect, update);
1331     }
1332 }
1333
```

```
1334
1335 AQS框架具体实现-独占锁实现ReentrantLock
1336
1337 public class ReentrantLock implements Lock, java.io.Serializable
1338 {
1339     private static final long serialVersionUID = 7373984872572414
1340     699L;
1341     /**
1342     * 内部调用AQS的动作，都基于该成员属性实现
1343     */
1344     private final Sync sync;
1345
1346     /**
1347     * ReentrantLock锁同步操作的基础类,继承自AQS框架。
1348     * 该类有两个继承类，1、NonfairSync 非公平锁，2、FairSync公平锁
1349     */
1350     abstract static class Sync extends AbstractQueuedSynchronizer
1351     {
1352         private static final long serialVersionUID = -517952376203402
1353         5860L;
1354
1355         /**
1356         * 加锁的具体行为由子类实现
1357         */
1358         abstract void lock();
1359
1360         /**
1361         * 尝试获取非公平锁
1362         */
1363         final boolean nonfairTryAcquire(int acquires) {
1364             //acquires = 1
1365             final Thread current = Thread.currentThread();
1366             int c = getState();
1367
1368             /**
1369             * 不需要判断同步队列（CLH）中是否有排队等待线程
1370             * 判断state状态是否为0，不为0可以加锁
1371             */
1372             if (c == 0) {
1373                 if (!hasCallerOwns())
1374                     if (tryAcquireNonfair(acquires))
1375                         return true;
1376             } else if (c < 0) {
1377                 if (tryAcquire(-acquires))
1378                     return true;
1379             }
1380             return false;
1381         }
1382     }
1383 }
```

```
1366  */
1367  if (c == 0) {
1368      //unsafe操作, cas修改state状态
1369      if (compareAndSetState(0, acquires)) {
1370          //独占状态锁持有者指向当前线程
1371          setExclusiveOwnerThread(current);
1372          return true;
1373      }
1374  }
1375  /**
1376   * state状态不为0, 判断锁持有者是否是当前线程,
1377   * 如果是当前线程持有 则state+1
1378   */
1379  else if (current == getExclusiveOwnerThread()) {
1380      int nextc = c + acquires;
1381      if (nextc < 0) // overflow
1382          throw new Error("Maximum lock count exceeded");
1383      setState(nextc);
1384      return true;
1385  }
1386  //加锁失败
1387  return false;
1388  }
1389
1390  /**
1391   * 释放锁
1392   */
1393  protected final boolean tryRelease(int releases) {
1394      int c = getState() - releases;
1395      if (Thread.currentThread() != getExclusiveOwnerThread())
1396          throw new IllegalMonitorStateException();
1397      boolean free = false;
1398      if (c == 0) {
1399          free = true;
1400          setExclusiveOwnerThread(null);
```

```
1401     }
1402     setState(c);
1403     return free;
1404 }
1405
1406 /**
1407  * 判断持有独占锁的线程是否是当前线程
1408  */
1409 protected final boolean isHeldExclusively() {
1410     return getExclusiveOwnerThread() == Thread.currentThread();
1411 }
1412
1413 //返回条件对象
1414 final ConditionObject newCondition() {
1415     return new ConditionObject();
1416 }
1417
1418
1419 final Thread getOwner() {
1420     return getState() == 0 ? null : getExclusiveOwnerThread();
1421 }
1422
1423 final int getHoldCount() {
1424     return isHeldExclusively() ? getState() : 0;
1425 }
1426
1427 final boolean isLocked() {
1428     return getState() != 0;
1429 }
1430
1431 /**
1432  * Reconstitutes the instance from a stream (that is, deserializes it).
1433  */
1434 private void readObject(java.io.ObjectInputStream s)
```

```

1435 throws java.io.IOException, ClassNotFoundException {
1436     s.defaultReadObject();
1437     setState(0); // reset to unlocked state
1438 }
1439 }
1440
1441 /**
1442  * 非公平锁
1443  */
1444 static final class NonfairSync extends Sync {
1445     private static final long serialVersionUID = 7316153563782823
1446     691L;
1447     /**
1448     * 加锁行为
1449     */
1449     final void lock() {
1450         /**
1451         * 第一步：直接尝试加锁
1452         * 与公平锁实现的加锁行为一个最大的区别在于，此处不会去判断同步队列
1453         (CLH队列)中
1454         * 是否有排队等待加锁的节点，上来直接加锁（判断state是否为0,CAS修改
1455         state为1）
1456         * ，并将独占锁持有者 exclusiveOwnerThread 属性指向当前线程
1457         * 如果当前有人占用锁，再尝试去加一次锁
1458         */
1457         if (compareAndSetState(0, 1))
1458             setExclusiveOwnerThread(Thread.currentThread());
1459         else
1460             //AQS定义的方法,加锁
1461             acquire(1);
1462         }
1463
1464     /**
1465     * 父类AbstractQueuedSynchronizer.acquire()中调用本方法
1466     */
1467     protected final boolean tryAcquire(int acquires) {

```

```
1468     return nonfairTryAcquire(acquires);
1469 }
1470 }
1471
1472 /**
1473  * 公平锁
1474  */
1475 static final class FairSync extends Sync {
1476     private static final long serialVersionUID = -300089789709046
1477     6540L;
1478     final void lock() {
1479         acquire(1);
1480     }
1481     /**
1482      * 重写aqs中的方法逻辑
1483      * 尝试加锁，被AQS的acquire()方法调用
1484      */
1485     protected final boolean tryAcquire(int acquires) {
1486         final Thread current = Thread.currentThread();
1487         int c = getState();
1488         if (c == 0) {
1489             /**
1490              * 与非公平锁中的区别，需要先判断队列当中是否有等待的节点
1491              * 如果没有则可以尝试CAS获取锁
1492              */
1493             if (!hasQueuedPredecessors() &&
1494                 compareAndSetState(0, acquires)) {
1495                 //独占线程指向当前线程
1496                 setExclusiveOwnerThread(current);
1497                 return true;
1498             }
1499         } else if (current == getExclusiveOwnerThread()) {
1500             int nextc = c + acquires;
1501             if (nextc < 0)
```



```
1502     throw new Error("Maximum lock count exceeded");
1503     setState(nexttc);
1504     return true;
1505 }
1506 return false;
1507 }
1508 }
1509
1510 /**
1511  * 默认构造函数，创建非公平锁对象
1512  */
1513 public ReentrantLock() {
1514     sync = new NonfairSync();
1515 }
1516
1517 /**
1518  * 根据要求创建公平锁或非公平锁
1519  */
1520 public ReentrantLock(boolean fair) {
1521     sync = fair ? new FairSync() : new NonfairSync();
1522 }
1523
1524 /**
1525  * 加锁
1526  */
1527 public void lock() {
1528     sync.lock();
1529 }
1530
1531 /**
1532  * 尝试获取去锁，获取失败被阻塞，线程被中断直接抛出异常
1533  */
1534 public void lockInterruptibly() throws InterruptedException {
1535     sync.acquireInterruptibly(1);
1536 }
```

```
1537
1538  /**
1539  * 尝试加锁
1540  */
1541  public boolean tryLock() {
1542  return sync.nonfairTryAcquire(1);
1543  }
1544
1545  /**
1546  * 指定等待时间内尝试加锁
1547  */
1548  public boolean tryLock(long timeout, TimeUnit unit)
1549  throws InterruptedException {
1550  return sync.tryAcquireNanos(1, unit.toNanos(timeout));
1551  }
1552
1553  /**
1554  * 尝试去释放锁
1555  */
1556  public void unlock() {
1557  sync.release(1);
1558  }
1559
1560  /**
1561  * 返回条件对象
1562  */
1563  public Condition newCondition() {
1564  return sync.newCondition();
1565  }
1566
1567  /**
1568  * 返回当前线程持有的state状态数量
1569  */
1570  public int getHoldCount() {
1571  return sync.getHoldCount();
```

```
1572     }
1573
1574     /**
1575     * 查询当前线程是否持有锁
1576     */
1577     public boolean isHeldByCurrentThread() {
1578         return sync.isHeldExclusively();
1579     }
1580
1581     /**
1582     * 状态表示是否被Thread加锁持有
1583     */
1584     public boolean isLocked() {
1585         return sync.isLocked();
1586     }
1587
1588     /**
1589     * 是否公平锁？ 是返回true 否则返回 false
1590     */
1591     public final boolean isFair() {
1592         return sync instanceof FairSync;
1593     }
1594
1595     /**
1596     * 获取持有锁的当前线程
1597     */
1598     protected Thread getOwner() {
1599         return sync.getOwner();
1600     }
1601
1602     /**
1603     * 判断队列当中是否有在等待获取锁的Thread节点
1604     */
1605     public final boolean hasQueuedThreads() {
1606         return sync.hasQueuedThreads();
```

```
1607     }
1608
1609     /**
1610     * 当前线程是否在同步队列中等待
1611     */
1612     public final boolean hasQueuedThread(Thread thread) {
1613         return sync.isQueued(thread);
1614     }
1615
1616     /**
1617     * 获取同步队列长度
1618     */
1619     public final int getQueueLength() {
1620         return sync.getQueueLength();
1621     }
1622
1623     /**
1624     * 返回Thread集合，排队中的所有节点Thread会被返回
1625     */
1626     protected Collection<Thread> getQueuedThreads() {
1627         return sync.getQueuedThreads();
1628     }
1629
1630     /**
1631     * 条件队列当中是否有正在等待的节点
1632     */
1633     public boolean hasWaiters(Condition condition) {
1634         if (condition == null)
1635             throw new NullPointerException();
1636         if (!(condition instanceof AbstractQueuedSynchronizer.ConditionObject))
1637             throw new IllegalArgumentException("not owner");
1638         return sync.hasWaiters((AbstractQueuedSynchronizer.ConditionObject)condition);
1639     }
1640
```

1641 }

有道云笔记链接: <http://note.youdao.com/noteshare?id=695b21d540f1a6c8c0dae11c4d696b1f&sub=7183334F03BD493CB95FF6BB532977F7>