记一次 Reentrantlock 深入学习

作者：425570408
原文链接：https://ld246.com/article/1575425672268
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

<h2 id="初始化">初始化</h2>
Reentrantlock 可以初始化为公平锁和非公平锁
<h4 id="公平锁">公平锁</h4>
公平锁指能让等待时间长的线程首先获得锁。可以防止线程饥饿。
<h4 id="非公平锁">非公平锁</h4>
指随机分配锁的使用权。其效率更高。
在创建 ReentrantLock 的时候通过传进参数 <code>true</code> 创建公平锁,如果传入的是 <
ode>false</code> 或没传参数则创建的是非公平锁
<pre><code class="highlight-chroma"><span class="highlight
cl"> /**
  * Creates an ins
ance of {@code ReentrantLock}.
  * This is equiva
ent to using {@code ReentrantLock(false)}.
  */
  public Reentran
Lock() {
  sync = new
onfairSync();
  }

  /**
  * Creates an ins
ance of {@code ReentrantLock} with the
  * given fairness
policy.
  *
  * @param fair
@code true} if this lock should use a fair ordering policy
  */
  public Reentran
Lock(boolean fair) {
  sync = fair ?
ew FairSync() : new NonfairSync();
  }
</code></pre>
<h2 id="加锁-LOCK-">加锁（LOCK） </h2>
公平锁
<pre><code class="highlight-chroma"><span class="highlight
cl">final void lock() {
  acquire(1);
  }
</code></pre>
非公平锁
<pre><code class="highlight-chroma"><span class="highlight
cl"> /**
  * Performs l
ck. Try immediate barge, backing up to normal
  * acquire on
ailure.
  */
  final void loc
() {
  if (compar

AndSetState(0, 1))
            setExclu
iveOwnerThread(Thread.currentThread());
        else
            acquire(
);
    }
</code></pre>
非公平锁可以立即通过用 CAS 去尝试修改 state，修改成功立即获得锁使用权。否则去执行 acqu
re 方法。
state:是关键性的变量，它标识了锁的状态，0 是空闲，大于或者等于 1 表示有线程占用了
锁。因为是可重入的锁，所以同一线程可以多次获取该锁，并且每次获取 state+1。当 state 为 0 时
他线程才获取到该锁。
<h3 id="AQS的acquire方法">AQS 的 acquire 方法</h3>
<pre><code class="highlight-chroma"><span class="highlight
cl">public final void acquire(int arg) {
    if (!tryAcquire
arg) &&
        acquireQu
ued(addWaiter(Node.EXCLUSIVE), arg))
        selfInterru
t();
  }
</code></pre>
当 tryAcquire 方法尝试获取锁成功时直接返回。意味着该线程获得了锁，业务代码继续执行。当
ryAcquire 尝试获得锁失败时。将该线程加入线程队列中，并中断该线程。
<h3 id="tryAcquire方法">tryAcquire 方法</h3>
公平锁实现
<pre><code class="highlight-chroma"><span class="highlight
cl">   protected final boolean tryAcquire(int acquires) {
        final Threa
 current = Thread.currentThread();
        int c = get
tate();
        if (c == 0) {
            if (!has
ueuedPredecessors() &&
                comp
reAndSetState(0, acquires)) {
                setExc
usiveOwnerThread(current);
                return
true;
            }
        }
        else if (cur
ent == getExclusiveOwnerThread()) {
            int nextc
= c + acquires;
            if (nextc
< 0)
                throw
new Error("Maximum lock count exceeded");
            setState

nextc);
            return t
ue;
        }
        return false

    }
  }

</code></pre>
获得当前线程，判断 state 是否为 0 如果为 0 则判断（hasQueuedPredecessors 方法）当前
程是否需要排队(不需要排队和 cas 修改成功直接返回 true)。当需要排队和 cas 修改不成功时。判断
前线程是否是锁持有线程，如果是锁持有线程则 state+1(返回 true)，否则返回 false;
hasQueuedPredecessors 方法
<pre><code class="highlight-chroma">>><span class="highlight
cl">public final boolean hasQueuedPredecessors() {
> // The correctness
of this depends on head being initialized
> // before tail and
on head.next being accurate if the current
> // thread is first in
queue.
> Node t = tail; //
ead fields in reverse initialization order
> Node h = head;
> Node s;
> return h != t &a
p;&
> ((s = h.next) ==
ull || s.thread != Thread.currentThread());
> }
</code></pre>
当初始化线程队列时会实例化一个 node（该 node 的 thread 属性为 null）并且队列的 tail 和 h
ad（头尾）都会引用该 node，所以当 h!=t 为 true 表示有等待线程，false 表示没有等待线程。因
队列头部的 node 的 thread 永远都是 null（后面会粘代码），所以 h.next 就是最先等待获取锁的线
。所以该方法判断如果 头 node 和尾 node 引用统一为 node 或者 h.next 不为空且 h.next 就是当
线程时不需要排队。（笔者对后面判断理解尚有不足）
非公平锁实现
<pre><code class="highlight-chroma">>><span class="highlight
cl"> protected final boolean tryAcquire(int acquires) {
>   return non
airTryAcquire(acquires);
>   }
> final boolean non
airTryAcquire(int acquires) {
>   final Threa
 current = Thread.currentThread();
>   int c = get
tate();
>   if (c == 0) {
>   if (comp
reAndSetState(0, acquires)) {
>   setExc

usiveOwnerThread(current);
                return true;
            }
        }
        else if (current == getExclusiveOwnerThread()) {
            int nextc = c + acquires;
            if (nextc < 0) // overflow
                throw new Error("Maximum lock count exceeded");
            setState(nextc);
            return true;
        }
        return false;
    }
</code></pre>
非公平锁与公平锁雷同，但是非公平锁不需要排队。
<h3 id="addWaiter">addWaiter</h3>
<pre><code class="highlight-chroma">private Node addWaiter(Node mode) {
    Node node = new Node(Thread.currentThread(), mode);
    // Try the fast path of enq; backup to full enq on failure
    Node pred = tail;
    if (pred != null) {
        node.prev = pred;
        if (compareAndSetTail(pred, node)) {
            pred.next = node;
            return node;
        }
    }
    enq(node);
    return node;
}
</code></pre>
传入的 mode 为 <code>Node.EXCLUSIVE</code> 表示实例化的 node 为排他的 node，其该方法就是将该线程所处 node 插入队列尾部。
enq 方法
<pre><code class="highlight-chroma">private Node enq(final Node node) {
    for (;;) {

```
</span></span><span class="highlight-line"><span class="highlight-cl">            Node t = t
il;
</span></span><span class="highlight-line"><span class="highlight-cl">            if (t == nul
) { // Must initialize
</span></span><span class="highlight-line"><span class="highlight-cl">                if (comp
reAndSetHead(new Node()))
</span></span><span class="highlight-line"><span class="highlight-cl">                    tail =
head;
</span></span><span class="highlight-line"><span class="highlight-cl">            } else {
</span></span><span class="highlight-line"><span class="highlight-cl">                node.pr
v = t;
</span></span><span class="highlight-line"><span class="highlight-cl">                if (comp
reAndSetTail(t, node)) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    t.next
= node;
</span></span><span class="highlight-line"><span class="highlight-cl">                    return
t;
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">            }
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span></code></pre>
<p>如果队列没有初始化则先将队列初始化，初始化后再将 node 插入队列。（thread 属性永远都
插入到队尾，所以初始化的头部节点 thread 属性永远都是为 null）;</p>
<h3 id="acquireQueued方法">acquireQueued 方法</h3>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl"> final boolean acquireQueued(final Node node, int arg) {
</span></span><span class="highlight-line"><span class="highlight-cl">        boolean faile
 = true;
</span></span><span class="highlight-line"><span class="highlight-cl">        try {
</span></span><span class="highlight-line"><span class="highlight-cl">            boolean i
terrupted = false;
</span></span><span class="highlight-line"><span class="highlight-cl">            //死循环,
常情况下线程只有获得锁才能跳出循环
</span></span><span class="highlight-line"><span class="highlight-cl">            for (;;) {
</span></span><span class="highlight-line"><span class="highlight-cl">                final No
e p = node.predecessor();//获得当前线程所在结点的前驱结点
</span></span><span class="highlight-line"><span class="highlight-cl">                //第一个i
分句
</span></span><span class="highlight-line"><span class="highlight-cl">                if (p ==
head &amp;amp;&amp;amp; tryAcquire(arg)) {
</span></span><span class="highlight-line"><span class="highlight-cl">                    setHe
d(node); //将当前结点设置为队列头结点
</span></span><span class="highlight-line"><span class="highlight-cl">                    p.next
= null; // help GC
</span></span><span class="highlight-line"><span class="highlight-cl">                    failed
= false;
</span></span><span class="highlight-line"><span class="highlight-cl">                    return
interrupted;//正常情况下死循环唯一的出口
</span></span><span class="highlight-line"><span class="highlight-cl">                }
</span></span><span class="highlight-line"><span class="highlight-cl">                //第二个i
分句
</span></span><span class="highlight-line"><span class="highlight-cl">                if (shoul
ParkAfterFailedAcquire(p, node) &amp;amp;&amp;amp;  //判断是否要阻塞当前线程
```

par
AndCheckInterrupt()) //阻塞当前线程
interr
pted = true;
}
} finally {
if (failed)
cancelA
quire(node);
}
}
</code></pre>
这段代码主要的内容都在 for 循环中,这是一个死循环,主要有两个 if 分句构成。第一个 if 分句中,前线程首先会判断前驱结点是否是头结点,如果是则尝试获取锁,获取锁成功则会设置当前结点为头结点更新头指针)。为什么必须前驱结点为头结点才尝试去获取锁?因为头结点表示当前正占有锁的线程,常情况下该线程释放锁后会通知后面结点中阻塞的线程,阻塞线程被唤醒后去获取锁,这是我们希望看的。然而还有一种情况,就是前驱结点取消了等待,此时当前线程也会被唤醒,这时候就不应该去获取锁,是往前回溯一直找到一个没有取消等待的结点,然后将自身连接在它后面。一旦我们成功获取了锁并成将自身设置为头结点,就会跳出 for 循环。否则就会执行第二个 if 分句:确保前驱结点的状态为 SIGNAL 然后阻塞当前线程。
先来看 shouldParkAfterFailedAcquire(p, node)，从方法名上我们可以大概猜出这是判断是否阻塞当前线程的,方法内容如下
<pre><code class="highlight-chroma">/**
* Checks and upd
tes status for a node that failed to acquire.
* Returns true if t
read should block. This is the main signal
* control in all ac
uire loops. Requires that pred == node.prev.
*
* @param pred n
de's predecessor holding status
* @param node t
e node
* @return {@cod
true} if thread should block
*/
private static bool
an shouldParkAfterFailedAcquire(Node pred, Node node) {
int ws = pred.w
itStatus;
if (ws == Node.
IGNAL) //状态为SIGNAL
/*
* This node
as already set status asking a release
* to signal it,
so it can safely park.
*/
return true;
if (ws > 0) { /
状态为CANCELLED,

原文链接：记一次 Reentrantlock 深入学习

```
</span></span><span class="highlight-line"><span class="highlight-cl">        /*
</span></span><span class="highlight-line"><span class="highlight-cl">         * Predecesso
 was cancelled. Skip over predecessors and
</span></span><span class="highlight-line"><span class="highlight-cl">         * indicate ret
y.
</span></span><span class="highlight-line"><span class="highlight-cl">         */
</span></span><span class="highlight-line"><span class="highlight-cl">        do {
</span></span><span class="highlight-line"><span class="highlight-cl">            node.prev
= pred = pred.prev;
</span></span><span class="highlight-line"><span class="highlight-cl">        } while (pred
waitStatus &gt; 0);
</span></span><span class="highlight-line"><span class="highlight-cl">        pred.next =
ode;
</span></span><span class="highlight-line"><span class="highlight-cl">    } else { //状态
初始化状态(ReentrentLock语境下)
</span></span><span class="highlight-line"><span class="highlight-cl">        /*
</span></span><span class="highlight-line"><span class="highlight-cl">         * waitStatus
must be 0 or PROPAGATE.  Indicate that we
</span></span><span class="highlight-line"><span class="highlight-cl">         * need a sig
al, but don't park yet.  Caller will need to
</span></span><span class="highlight-line"><span class="highlight-cl">         * retry to ma
e sure it cannot acquire before parking.
</span></span><span class="highlight-line"><span class="highlight-cl">         */
</span></span><span class="highlight-line"><span class="highlight-cl">        compareAnd
etWaitStatus(pred, ws, Node.SIGNAL);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span><span class="highlight-line"><span class="highlight-cl">    return false;
</span></span><span class="highlight-line"><span class="highlight-cl">}
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span></code></pre>
<p>可以看到针对前驱结点 pred 的状态会进行不同的处理</p>
<ul>
<li>1.pred 状态为 SIGNAL,则返回 true,表示要阻塞当前线程。</li>
<li>2.pred 状态为 CANCELLED,则一直往队列头部回溯直到找到一个状态不为 CANCELLED 的结点,
当前节点 node 挂在这个结点的后面。</li>
<li>3.pred 的状态为初始化状态,此时通过 compareAndSetWaitStatus(pred, ws, Node.SIGNAL)
法将 pred 的状态改为 SIGNAL。</li>
</ul>
<p>其实这个方法的含义很简单,就是确保当前结点的前驱结点的状态为 SIGNAL,SIGNAL 意味着线
释放锁后会唤醒后面阻塞的线程。毕竟,只有确保能够被唤醒，当前线程才能放心的阻塞。</p>
<p>但是要注意只有在前驱结点已经是 SIGNAL 状态后才会执行后面的方法立即阻塞,对应上面的第
种情况。其他两种情况则因为返回 false 而重新执行一遍<br>
for 循环。这种延迟阻塞其实也是一种高并发场景下的优化,试想我如果在重新执行循环的时候成功获
了锁,是不是线程阻塞唤醒的开销就省了呢？</p>
<p>最后我们来看看阻塞线程的方法 parkAndCheckInterrupt</p>
<p>shouldParkAfterFailedAcquire 返回 true 表示应该阻塞当前线程,则会执行 parkAndCheckInter
upt 方法,这个方法比较简单,底层调用了 LockSupport 来阻塞当前线程,源码如下:</p>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">/**
</span></span><span class="highlight-line"><span class="highlight-cl">  * Convenience m
thod to park and then check if interrupted
</span></span><span class="highlight-line"><span class="highlight-cl">  *
</span></span><span class="highlight-line"><span class="highlight-cl">  * @return {@cod
 true} if interrupted
```

  */
private final boole
n parkAndCheckInterrupt() {
    LockSupport.pa
k(this);
    return Thread.in
errupted();
}
</code></pre>
该方法内部通过调用 LockSupport 的 park 方法来阻塞当前线程。
下面通过一张流程图来说明线程从加入同步队列到成功获取锁的过程
<img src="https://ld246.com/images/img-loading.svg" alt="1422237201808051842424381 6
0878927.png" data-src="https://b3logfile.com/file/2019/11/1422237201808051842424381 6
3878927-6c964f2d.png?imageView2/2/interlace/1/format/jpg">
概括的说,线程在同步队列中会尝试获取锁,失败则被阻塞,被唤醒后会不停的重复这个过程,直到线
真正持有了锁,并将自身结点置于队列头部。
<h2 id="解锁-unlock-">解锁（unlock）</h2>
<pre><code class="highlight-chroma"><span class="highlight
cl">public final boolean release(int arg) {
        if (tryRelease
arg)) {
            Node h =
ead;
            if (h != null
&& h.waitStatus != 0)
                unparkS
ccessor(h);
            return true

        }
        return false;
    }
</code></pre>
尝试解锁，成功后如果头部 node 不为空且 waitState 状态为 0 则唤醒该线程;
<h2 id="tryRelease方法">tryRelease 方法</h2>
<pre><code class="highlight-chroma">
<span class="highlight
cl">protected final boolean tryRelease(int releases) {
            int c = get
tate() - releases;
            if (Thread.
urrentThread() != getExclusiveOwnerThread())
                throw n
w IllegalMonitorStateException();
            boolean fr
e = false;
            if (c == 0) {
                free = t
ue;
                setExclu
iveOwnerThread(null);
            }
            setState(c);
            return free;
        }

```
</span></span></code></pre>
<p>如果当前线程不是持有锁的线程则抛出 IllegalMonitorStateException 异常。判断 state-releas
s 如果为 0 则设置当前占用锁的线程为空。不为 0 则表示该锁被线程重入了，只是设置 state 值。最
返回锁的状态。</p>
<h3 id="unparkSuccessor方法">unparkSuccessor 方法</h3>
<pre><code class="highlight-chroma"><span class="highlight-line"><span class="highlight
cl">    private void unparkSuccessor(Node node) {
</span></span><span class="highlight-line"><span class="highlight-cl">        /*
</span></span><span class="highlight-line"><span class="highlight-cl">         * If status is
egative (i.e., possibly needing signal) try
</span></span><span class="highlight-line"><span class="highlight-cl">         * to clear in
nticipation of signalling.  It is OK if this
</span></span><span class="highlight-line"><span class="highlight-cl">         * fails or if st
tus is changed by waiting thread.
</span></span><span class="highlight-line"><span class="highlight-cl">         */
</span></span><span class="highlight-line"><span class="highlight-cl">        int ws = node
waitStatus;
</span></span><span class="highlight-line"><span class="highlight-cl">        if (ws &lt; 0)
</span></span><span class="highlight-line"><span class="highlight-cl">            compareA
dSetWaitStatus(node, ws, 0);
</span></span><span class="highlight-line"><span class="highlight-cl">
</span></span><span class="highlight-line"><span class="highlight-cl">        /*
</span></span><span class="highlight-line"><span class="highlight-cl">         * Thread to
npark is held in successor, which is normally
</span></span><span class="highlight-line"><span class="highlight-cl">         * just the ne
t node.  But if cancelled or apparently null,
</span></span><span class="highlight-line"><span class="highlight-cl">         * traverse ba
kwards from tail to find the actual
</span></span><span class="highlight-line"><span class="highlight-cl">         * non-cancel
ed successor.
</span></span><span class="highlight-line"><span class="highlight-cl">         */
</span></span><span class="highlight-line"><span class="highlight-cl">        Node s = no
e.next;
</span></span><span class="highlight-line"><span class="highlight-cl">        if (s == null ||
s.waitStatus &gt; 0) {
</span></span><span class="highlight-line"><span class="highlight-cl">            s = null;
</span></span><span class="highlight-line"><span class="highlight-cl">            for (Node t
= tail; t != null &amp;&amp; t != node; t = t.prev)
</span></span><span class="highlight-line"><span class="highlight-cl">                if (t.wait
tatus &lt;= 0)
</span></span><span class="highlight-line"><span class="highlight-cl">                    s = t;
</span></span><span class="highlight-line"><span class="highlight-cl">        }
</span></span><span class="highlight-line"><span class="highlight-cl">        if (s != null)
</span></span><span class="highlight-line"><span class="highlight-cl">            LockSuppo
t.unpark(s.thread);
</span></span><span class="highlight-line"><span class="highlight-cl">    }
</span></span></code></pre>
<p>这里涉及到 waitStatus 的一个知识，从百度上粘了一部分来，原文地址：<br>
<a href="https://ld246.com/forward?goto=https%3A%2F%2Fwww.jianshu.com%2Fp%2Fc021
144a565" target="_blank" rel="nofollow ugc">https://www.jianshu.com/p/c021f144a565</a
</p>
<blockquote>
<p><strong>waitStatus</strong></p>
</blockquote>
```

<pre><code class="language-java highlight-chroma">volatile int waitStatus;

</code></pre>
<blockquote>
状态属性，只有如下值：
① SIGNAL：
<code>static final int SIGNAL = -1;</code>
这个节点的后继（或者即将被阻塞）被阻塞（通过 park 阻塞）了，所以当前节点需要唤醒它的后继。它被释放或者取消时。为了避免竞争，获取方法必须首先表示他们需要一个通知信号，然后再原子性尝试获取锁，如果失败，则阻塞。
也就是说，在获取锁的操作中，需要确保当前 node 的 preNode 的 waitStatus 状态值为’SIGNAL’，才可以被阻塞，当获取锁失败时。（『shouldParkAfterFailedAcquire』方法的用意就是这）
② CANCELLED：
<code>static final int CANCELLED = 1;</code>
这个节点由于超时或中断被取消了。节点不会离开（改变）这个状态。尤其，一个被取消的线程不再被阻塞了。
③ CONDITION：
<code>static final int CONDITION = -2;</code>
这个节点当前在一个条件队列中。它将不会被用于当做一个同步队列的节点直到它被转移到同步队列，转移的同时状态值（waitStatus）将会被设置为 0。（这里使用这个值将不会做任何事情与该字段他值对比，只是为了简化机制）。
④ PROPAGATE：
<code>static final int PROPAGATE = -3;</code>
一个 releaseShared 操作必须被广播给其他节点。（只有头节点的）该值会在 doReleaseShared 方中被设置去确保持续的广播，即便其他操作的介入。
⑤ 0：不是上面的值的情况。
这个值使用数值排列以简化使用。非负的值表示该节点不需要信号（通知）。因此，大部分代码不需去检查这个特殊的值，只是为了标识。
对于常规的节点该字段会被初始化为 0，竞争节点该值为 CONDITION。这个值使用 CAS 修改（或可能的话，无竞争的 volatile 写）。
</blockquote>
该方法用于唤醒后继节点，如果存在的话
① 如果状态值是负数，则在预期发信号通知时清除这个负数状态值。如果状态被等待的线程修改了或清除负数状态值失败是允许。
② 后继节点的线程被唤醒，后继节点通常就是下一个节点。但是如果下一个节点被取消了或者下一个点为 null，则从队列尾(tail)往前遍历去找真实的未取消的后继节点。
『(s == null || s.waitStatus > 0)』：说明下一个节点为 null 或被取消了（waitStatus 允许的状值中，只有’CANCELLED’是 >0 的）。那么，就从队列尾（tail）开始向前遍历，获取第一个空且未被取消的节点。如果存在这样的一个后继节点的话（即，“s != null”），则执行『LockSupport.unpark(s.thread);』操作来唤醒这个节点的线程，此时等待队列中第一个等待的线程就会被重新启，流程会回到『acquireQueued』方法，该线程会重新重试获取该锁，如果成功 acquireQueued 方返回，否则线程会再次被挂起，等待下次唤醒后再去再去竞争获取锁。