深入浅出 Java Concurrency (10): 锁机制 part 5 闭锁 (CountDownLatch)

此小节介绍几个与锁有关的有用工具。

闭锁（Latch）

闭锁（Latch）：一种同步方法，可以延迟线程的进度直到线程到达某个终点状态。通俗的讲就是，一个闭锁相当于一扇大门，在大门打开之前所有线程都被阻断，一旦大门打开所有线程都将通过，但是一旦大门打开，所有线程都通过了，那么这个闭锁的状态就失效了，门的状态也就不能变了，只能是打开状态。也就是说闭锁的状态是一次性的，它确保在闭锁打开之前所有特定的活动都需要在闭锁打开之后才能完成。

CountDownLatch是JDK 5+里面闭锁的一个实现，允许一个或者多个线程等待某个事件的发生。**CountDownLatch**有一个正数计数器，*countDown*方法对计数器做减操作，*await*方法等待计数器达到0。所有*await*的线程都会阻塞直到计数器为0或者等待线程中断或者超时。

CountDownLatch的API如下。

- public void await() throws InterruptedException
- public boolean await(long timeout, TimeUnit unit) throws InterruptedException
- public void countDown()
- public long getCount()

其中*getCount()*描述的是当前计数，通常用于调试目的。

下面的例子中描述了闭锁的两种常见的用法。

```
package xylz.study.concurrency.lock;

import java.util.concurrent.CountDownLatch;

public class PerformanceTestTool {

    public long timecost(final int times, final Runnable
task) throws InterruptedException {
    if (times <= 0) throw new IllegalArgumentException();
    final CountDownLatch startLatch = new
CountDownLatch(1);
    final CountDownLatch overLatch = new
CountDownLatch(times);
    for (int i = 0; i < times; i++) {
        new Thread(new Runnable() {
            public void run() {
                try {
                    startLatch.await();
                    //
                    task.run();
                } catch (InterruptedException ex) {
```

2012年2月 (3)
2012年1月 (2)
2011年12月 (6)
2011年11月 (1)
2011年10月 (1)
2011年7月 (2)
2011年6月 (3)
2011年4月 (1)
2011年2月 (2)
2011年1月 (5)
2010年12月 (4)
2010年11月 (4)
2010年8月 (3)
2010年7月 (24)
2010年6月 (2)
2010年1月 (5)
2009年12月 (12)
2009年11月 (2)
2009年9月 (1)
2009年7月 (1)

文章分类(12)
Eclipse (rss)
Java (rss)
Python(12) (rss)

文章档案(12)
2010年6月 (6)
2010年5月 (3)
2010年4月 (3)

友情链接
imxylz (rss)
jafka
a fast distributed publish-subscribe messaging system

搜索
搜索

积分与排名
积分 - 1918335
排名 - 4

最新评论 XML
1. re: 深入浅出 Java Concurrency (9): 锁机制 part 4
评论内容较长,点击标题查看
--guanzhisong
2. 提供参考链接
评论内容较长,点击标题查看
--33
3. re: 处理Zookeeper的session过期问题
评论内容较长,点击标题查看
--codewarrior
4. re: 当Ajax遭遇GBK编码 (完全解决方案)
前面说的都同意 就是解决方法说的太抽象 没看懂 给个具体的方法
--穆

```
                    Thread.currentThread().interrupt();
                } finally {
                    overLatch.countDown();
                }
            }
        }).start();
    }
    //
    long start = System.nanoTime();
    startLatch.countDown();
    overLatch.await();
    return System.nanoTime() - start;
}

}
```

在上面的例子中使用了两个闭锁，第一个闭锁确保在所有线程开始执行任务前，所有准备工作都已经完成，一旦准备工作完成了就调用*startLatch.countDown()*打开闭锁，所有线程开始执行。第二个闭锁在于确保所有任务执行完成后主线程才能继续进行，这样保证了主线程等待所有任务线程执行完成后才能得到需要的结果。在第二个闭锁当中，初始化了一个N次的计数器，每个任务执行完成后都会将计数器减一，所有任务完成后计数器就变为了0，这样主线程闭锁overLatch拿到此信号后就可以继续往下执行了。

根据前面的happend-before法则可以知道闭锁有以下特性：

> ***内存一致性效果：线程中调用 `countDown()` 之前的操作 *happen-before* 紧跟在从另一个线程中对应 `await()` 成功返回的操作。***

在上面的例子中第二个闭锁相当于把一个任务拆分成N份，每一份独立完成任务，主线程等待所有任务完成后才能继续执行。这个特性在后面的线程池框架中会用到，其实**FutureTask**就可以看成一个闭锁。后面的章节还会具体分析**FutureTask**的。

同样基于探索精神，仍然需要"窥探"下**CountDownLatch**里面到底是如何实现*await**和*countDown*的。

首先，研究下*await()*方法。内部直接调用了**AQS**的*acquireSharedInterruptibly(1)*。

```
    public final void acquireSharedInterruptibly(int arg)
throws InterruptedException {
    if (Thread.interrupted())
        throw new InterruptedException();
    if (tryAcquireShared(arg) < 0)
        doAcquireSharedInterruptibly(arg);
}
```

前面一直提到的都是独占锁（排它锁、互斥锁），现在就用到了另外一种锁，共享锁。

所谓共享锁是说所有共享锁的线程共享同一个资源，一旦任意一个线程拿到共享资源，那么所有线程就都拥有的同一份资源。也就是通常情况下共享锁只是一个标志，所有线程都等待这个标识是否满足，一旦满足所有线程都被激活（相当于所有线程都拿到锁一样）。这里的闭锁**CountDownLatch**就是基于共享锁的实现。

闭锁中关于**AQS**的*tryAcquireShared*的实现是如下代码（**java.util.concurrent.CountDownLatch.Sync.tryAcquireShared**）：

```
public int tryAcquireShared(int acquires) {
return getState() == 0? 1 : -1;
}
```

在这份逻辑中，对于闭锁而言第一次await时tryAcquireShared应该总是-1，因为对于闭锁**CountDownLatch**而言*state*的值就是初始化的*count*值。这也就解释了为什么在*countDown*调用之前闭锁的*count*总是>0。

```
private void doAcquireSharedInterruptibly(int arg)
throws InterruptedException {
final Node node = addWaiter(Node.SHARED);
try {
   for (;;) {
      final Node p = node.predecessor();
      if (p == head) {
         int r = tryAcquireShared(arg);
         if (r >= 0) {
            setHeadAndPropagate(node, r);
            p.next = null; // help GC
            return;
         }
      }
      if (shouldParkAfterFailedAcquire(p, node) &&
         parkAndCheckInterrupt())
         break;
   }
} catch (RuntimeException ex) {
   cancelAcquire(node);
   throw ex;
}
// Arrive here only if interrupted
cancelAcquire(node);
throw new InterruptedException();
}
```

上面的逻辑展示了如何通过*await*将所有线程串联并挂起，直到被唤醒或者条件满足或者被中断。整个过程是这样的：

1. 将当前线程节点以共享模式加入**AQS**的**CLH**队列中（相关概念参考这里和这里）。进行2。
2. 检查当前节点的前任节点，如果是头结点并且当前闭锁计数为0就将当前节点设置为头结点，唤醒继任节点，返回（结束线程阻塞）。否则进行3。
3. 检查线程是否该阻塞，如果应该就阻塞(park)，直到被唤醒（unpark）。重复2。
4. 如果2、3有异常就抛出异常（结束线程阻塞）。

这里有一点值得说明下，设置头结点并唤醒继任节点*setHeadAndPropagate*。由于前面*tryAcquireShared*总是返回1或者-1，而进入*setHeadAndPropagate*时总是*propagate>=0*，所以这里*propagate==1*。后面唤醒继任节点操作就非常熟悉了。

@高帆
jstack 打印出来 线程栈信息，能看到 线程栈目前运行在那个地方，等待什么资源
--shaw
22. re: 《深入浅出 Java Concurrency》目录[未登录]
谢谢楼主!
--luke
23. re: 世界邦旅行网(北京)招聘Java高级/资深工程师前端工程师/移动开发工程师等_20150616更新
评论内容较长,点击标题查看
--songxin
24. re: 深入浅出 Java Concurrency (9): 锁机制 part 4
评论内容较长,点击标题查看
--imxylz
25. re: 深入浅出 Java Concurrency (9): 锁机制 part 4
评论内容较长,点击标题查看
--mitisky
26. re: 一次简单却致命的错误
请交大侠！查看java线程是怎么看的
--高帆
27. re: 深入浅出 Java Concurrency (20): 并发容器 part 5 ConcurrentLinkedQueue
@mashiguang
我猜测应该是ArrayBlockingQueue用一把锁而LinkedBlockingQueue用两把锁的原因
--liubey
28. John
Thanksamundo for the post.Really thank you! Awesome. edebdbceaekadffd
--Smithc667
29. re: 捕获Java线程池执行任务抛出的异常
@imxylz
谢谢指点，你这种方式更优雅些，我自己是new了个exceptionQueue，new线程的时候set进去，然后执行完子线程后查看这个Queue
--liubey
30. re: 使用Nginx的proxyCache缓存功能

浏览器收到302之后，就直接去源链接下载了，消息都不经过nginx了，然后怎么缓存呢?
--wd
31. re: 使用Nginx的proxyCache缓存功能
nginx支持302这种非200状态码，但是当有302经过nginx，它缓存的是什么？是源文件还是链接？我从浏览器输入的时候还是会收到302啊。
--wd

```
    private void setHeadAndPropagate(Node node, int propagate) {
        setHead(node);
        if (propagate > 0 && node.waitStatus != 0) {
            Node s = node.next;
            if (s == null || s.isShared())
                unparkSuccessor(node);
        }
    }
```

从上面的所有逻辑可以看出*countDown*应该就是在条件满足（计数为0）时唤醒头结点（时间最长的一个节点），然后头结点就会根据FIFO队列唤醒整个节点列表（如果有的话）。

从**CountDownLatch**的*countDown*代码中看到，直接调用的是**AQS**的*releaseShared(1)*，参考前面的知识，这就印证了上面的说法。

tryReleaseShared中正是采用CAS操作减少计数（每次减-1）。

```
    public boolean tryReleaseShared(int releases) {
        for (;;) {
            int c = getState();
            if (c == 0)
                return false;
            int nextc = c-1;
            if (compareAndSetState(c, nextc))
                return nextc == 0;
        }
    }
```

整个**CountDownLatch**就是这个样子的。其实有了前面原子操作和**AQS**的原理及实现，分析**CountDownLatch**还是比较容易的。

锁释放与条件变量 (Lock.unlock & Condition) 目 录 锁机制 part 6 CyclicBarrier

posted on 2010-07-09 09:21 imxylz 阅读(26167) 评论(6) 编辑 收藏 所属分类: J2EE

评论

 # re: 深入浅出 Java Concurrency (10): 锁机制 part 5 闭锁 (CountDownLatch)[未登录] 2010-07-12 19:40 行云流水

坚持。几天没有更新了，坚持一天3篇，支持支持。。 回复 更多评论

 # re: 深入浅出 Java Concurrency (10): 锁机制 part 5 闭锁 (CountDownLatch) 2010-07-12 23:40 xylz

@行云流水