深入浅出 Java Concurrency (11): 锁机制 part 6 CyclicBarrier

如果说CountDownLatch是一次性的，那么**CyclicBarrier**正好可以循环使用。它允许一组线程互相等待，直到到达某个公共屏障点 (common barrier point)。所谓屏障点就是一组任务执行完毕的时刻。

清单1 一个使用CyclicBarrier的例子

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

import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierDemo {

    final CyclicBarrier barrier;

    final int MAX_TASK;

    public CyclicBarrierDemo(int cnt) {
    barrier = new CyclicBarrier(cnt + 1);
    MAX_TASK = cnt;
}

    public void doWork(final Runnable work) {
    new Thread() {

        public void run() {
        work.run();
        try {
            int index = barrier.await();
            doWithIndex(index);
        } catch (InterruptedException e) {
            return;
        } catch (BrokenBarrierException e) {
            return;
        }
      }
    }.start();
}

    private void doWithIndex(int index) {
    if (index == MAX_TASK / 3) {
        System.out.println("Left 30%.");
    } else if (index == MAX_TASK / 2) {
        System.out.println("Left 50%");
    } else if (index == 0) {
        System.out.println("run over");
```

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

```
        }
    }

        public void waitForNext() {
        try {
            doWithIndex(barrier.await());
        } catch (InterruptedException e) {
            return;
        } catch (BrokenBarrierException e) {
            return;
        }
    }

        public static void main(String[] args) {
        final int count = 10;
        CyclicBarrierDemo demo = new
CyclicBarrierDemo(count);
        for (int i = 0; i < 100; i++) {
            demo.doWork(new Runnable() {

                public void run() {
                //do something
                try {
                    Thread.sleep(1000L);
                } catch (Exception e) {
                    return;
                }
            }
        });
        if ((i + 1) % count == 0) {
            demo.waitForNext();
        }
    }
}

    }
```

清单1描述的是一个周期性处理任务的例子，在这个例子中有一对的任务（100个），希望每10个为一组进行处理，当前仅当上一组任务处理完成后才能进行下一组，另外在每一组任务中，当任务剩下50%，30%以及所有任务执行完成时向观察者发出通知。

在这个例子中，CyclicBarrierDemo 构建了一个count+1的任务组（其中一个任务时为了外界方便挂起主线程）。每一个子任务里，人物本身执行完毕后都需要等待同组内其它任务执行完成后才能继续。同时在剩下任务50%、30%已经0时执行特殊的其他任务（发通知）。

很显然CyclicBarrier有以下几个特点：

- await()方法将挂起线程，直到同组的其它线程执行完毕才能继续
- await()方法返回线程执行完毕的索引，注意，索引时从任务数-1开始的，也就是第一个执行完成的任务索引为parties-1，最后一个为0，这个parties为总任务数，清单中是cnt+1
- CyclicBarrier 是可循环的，显然名称说明了这点。在清单1中，每一组任务执行完毕就能够执行下一组任务。

另外除了CyclicBarrier除了以上特点外，还有以下几个特点：

- 如果屏障操作不依赖于挂起的线程，那么任何线程都可以执行屏障操作。在清单1中可以看到并没有指定那个线程执行50%、30%、0%的操作，而是一组线程（cnt+1）个中任何一个线程只要到达了屏障点都可以执行相应的操作
- CyclicBarrier 的构造函数允许携带一个任务，这个任务将在0%屏障点执行，它将在await()==0后执行。
- CyclicBarrier 如果在await时因为中断、失败、超时等原因提前离开了屏障点，那么任务组中的其他任务将立即被中断，以InterruptedException异常离开线程。
- 所有await()之前的操作都将在屏障点之前运行，也就是 CyclicBarrier 的内存一致性效果

CyclicBarrier 的所有API如下：

- *public CyclicBarrier(int parties)* 创建一个新的 `CyclicBarrier`，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。
- *public CyclicBarrier(int parties, Runnable barrierAction)* 创建一个新的 `CyclicBarrier`，它将在给定数量的参与者（线程）处于等待状态时启动，并在启动 barrier 时执行给定的屏障操作，该操作由最后一个进入 barrier 的线程执行。
- *public int await() throws InterruptedException, BrokenBarrierException* 在所有参与者都已经在此 barrier 上调用 `await` 方法之前，将一直等待。
- *public int await(long timeout,TimeUnit unit) throws InterruptedException, BrokenBarrierException,TimeoutException* 在所有参与者都已经在此屏障上调用 await 方法之前将一直等待,或者超出了指定的等待时间。
- *public int getNumberWaiting()* 返回当前在屏障处等待的参与者数目。此方法主要用于调试和断言。
- *public int getParties()* 返回要求启动此 barrier 的参与者数目。
- *public boolean isBroken()* 查询此屏障是否处于损坏状态。
- *public void reset()* 将屏障重置为其初始状态。

针对以上API，下面来探讨下CyclicBarrier 的实现原理，以及为什么有这样的API。

清单2 CyclicBarrier.await*()的实现片段

```
    private int dowait(boolean timed, long nanos)
throws InterruptedException, BrokenBarrierException,
    TimeoutException {
final ReentrantLock lock = this.lock;
lock.lock();
try {
   final Generation g = generation;
   if (g.broken)
       throw new BrokenBarrierException();
```

@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

```
                if (Thread.interrupted()) {
                breakBarrier();
                throw new InterruptedException();
            }

                int index = --count;
            if (index == 0) {  // tripped
                boolean ranAction = false;
                try {
                    final Runnable command = barrierCommand;
                    if (command != null)
                        command.run();
                    ranAction = true;
                    nextGeneration();
                    return 0;
                } finally {
                    if (!ranAction)
                        breakBarrier();
                }
            }

            // loop until tripped, broken, interrupted, or timed
        out
            for (;;) {
                try {
                    if (!timed)
                        trip.await();
                    else if (nanos > 0L)
                        nanos = trip.awaitNanos(nanos);
                } catch (InterruptedException ie) {
                    if (g == generation && ! g.broken) {
                        breakBarrier();
                        throw ie;
                    } else {
                        Thread.currentThread().interrupt();
                    }
                }

                    if (g.broken)
                    throw new BrokenBarrierException();

                    if (g != generation)
                    return index;

                    if (timed && nanos <= 0L) {
                    breakBarrier();
                    throw new TimeoutException();
                }
            }
        } finally {
            lock.unlock();
        }
    }
```

清单2有点复杂，这里一点一点的剖析，并且还原到最原始的状态。

利用前面学到的知识，我们知道要想让线程等待其他线程执行完毕，那么已经执行完毕的线程（进入await*()方法）就需要park()，直到超时或者被中断，或者被

32. re: 捕获Java线程池执行任务抛出的异常
@liubey
友好的做法是子线程不抛出异常，返回不同的结果，或者将异常封装到return对象中。父对象根据此结果/异常封装友好的提示给界面。
--imxylz
33. re: 捕获Java线程池执行任务抛出的异常
我想问下LZ，如果主线程想拿到子线程的异常，比如展示给界面，该怎么做=。=
--liubey
34. cialis fast delivery
Hello!
--fast
35. viagra fast delivery
Hello!
--fast
36. re: 《深入浅出 Java Concurrency》目录
真在学习，楼主，顶你
--Jerran
37. re: Crack JRebel 5.3.1
could you help me to crack jrebel-6.0.0? thanhnhadang@yahoo.com Thanks!
--MCOT Nha
38. re: 一个查找jar包中类的小工具[未登录]
这东西做得不错，确实是用来找自定义的jar（第三方公司，不是开源项目）包中class的利器。感谢楼主
--jiajia
39. re: Google Guice 入门教程02 - 依赖注入(2)
评论内容较长,点击标题查看
--库特
40. re: 深入浅出 Java Concurrency (29): 线程池 part 2 Executor 以及Executors
写得很不错，持续学习中
--Windchill_Fan

阅读排行榜

1. 《深入浅出 Java Concurrency》目录(110269)
2. 深入浅出 Java Concurrency (1)：J.U.C的整体认识(49424)
3. 深入浅出 Java Concurrency (2): 原子操作 part 1(49299)
4. 深入浅出 Java Concurrency (7): 锁机制 part 2 AQS(43938)
5. JRebel 6.0.0 Crack (20141216更新)(41100)
6. 深入浅出 Java Concurrency (5): 原子操作 part 4(39014)
7. 深入浅出 Java Concurrency (4): 原子操作 part 3 指令重排序与happens-before法则(38210)
8. 深入浅出 Java Concurrency (3): 原子操作 part 2(37706)

其它线程唤醒。

前面说过CyclicBarrier 的特点是要么大家都正常执行完毕，要么大家都异常被中断，不会其中有一个被中断而其它正常执行完毕的现象存在。这种特点叫all-or-none。类似的概念是原子操作中的要么大家都执行完，要么一个操作都不执行完。当前这其实是两个概念了。要完成这样的特点就必须有一个状态来描述曾经是否有过线程被中断（broken)了，这样后面执行完的线程就该知道是否需要继续等待了。而在CyclicBarrier 中Generation 就是为了完成这件事情的。Generation的定义非常简单，整个结构就只有一个变量*boolean broken = false;*，定义是否发生了broken操作。

由于有竞争资源的存在（broken/index），所以毫无疑问需要一把锁lock。拿到锁后整个过程是这样的：

1. 检查是否存在中断位(broken)，如果存在就立即以BrokenBarrierException异常返回。此异常描述的是线程进入屏障被破坏的等待状态。否则进行2。
2. 检查当前线程是否被中断，如果是那么就设置中断位（使其它将要进入等待的线程知道），另外唤醒已经等待的线程，同时以InterruptedException异常返回，表示线程要处理中断。否则进行3。
3. 将剩余任务数减1，如果此时剩下的任务数为0，也就是达到了公共屏障点，那么就执行屏障点任务（如果有的话），同时创建新的Generation（在这个过程中会唤醒其它所有线程，因此当前线程是屏障点线程，那么其它线程就都应该在等待状态）。否则进行4。
4. 到这里说明还没有到达屏障点，那么此时线程就应该park()。很显然在下面的for循环中就是要park线程。这里park线程采用的是Condition.await()方法。也就是trip.await*()。为什么需要Condition？因为所有的await*()其实等待的都是一个条件，一旦条件满足就应该都被唤醒，所以Condition整好满足这个特点。所以到这里就会明白为什么在步骤3中到达屏障点时创建新的Generation的时候是一定要唤醒其它线程的原因了。

上面4个步骤其实只是描述主体结构，事实上整个过程中有非常多的逻辑来处理异常引发的问题，比如执行屏障点任务引发的异常，park线程超时引发的中断异常和超时异常等等。所以对于await()而言，异常的处理比业务逻辑的处理更复杂，这就解释了为什么await()的时候可能引发 *InterruptedException,BrokenBarrierException,TimeoutException* 三种异常。

清单3 生成下一个循环周期并唤醒其它线程

```
    private void nextGeneration() {
     trip.signalAll();
     count = parties;
     generation = new Generation();
  }
```

清单3 描述了如何生成下一个循环周期的过程，在这个过程中当然需要使用Condition.signalAll()唤醒所有已经执行完成并且正在等待的线程。另外这里count描述的是还有多少线程需要执行，是为了线程执行完毕索引计数。

isBroken() 方法描述的就是generation.broken，也即线程组是否发生了异常。这里再一次解释下为什么要有这个状态的存在。

如果一个将要位于屏障点或者已经位于屏障点的而执行屏障点任务的线程发生了异常，那么即使唤醒了其它等待的线程，其它等待的线程也会因为循环等待而"死去"，因为再也没有一个线程来唤醒这些第二次进行park的线程了。还有一个意图是，如果屏障点都已经损坏了，那么其它将要等待屏障点的再线程挂起就没有意义了。

写到这里的时候非常不幸，用了4年多了台灯终于"寿终正寝了"。

其实CyclicBarrier 还有一个reset方法，描述的是手动立即将所有线程中断，恢复屏障点，进行下一组任务的执行。也就是与重新创建一个新的屏障点相比，可能维护的代价要小一些（减少同步，减少上一个CyclicBarrier 的管理等等）。

本来是想和Semaphore 一起将的，最后发现铺开后就有点长了，而且也不利于理解和吸收，所以放到下一篇吧。

参考资料:

1. 使用 CyclicBarrier 做线程间同步
2. CyclicBarrier And CountDownLatch Tutorial
3. 线程—CyclicBarrier
4. Java线程学习笔记（十）CountDownLatch 和CyclicBarrier
5. 关于多线程同步的初步教程－－Barrier的设计及使用
6. Thread coordination with CountDownLatch and CyclicBarrier
7. 如何充分利用多核CPU，计算很大的List中所有整数的和

锁机制 part 5 闭锁 (CountDownLatch)　　目　录

锁机制 part 7 信号量 (Semaphore)

posted on 2010-07-12 23:33 imxylz 阅读(20020) 评论(2) 编辑 收藏 所属分类: J2EE

评论

 # re: 深入浅出 Java Concurrency (11): 锁机制 part 6 CyclicBarrier 2010-07-13 16:12 Mercy

和CountDownLatch类似，不过由于它的reset方法让他可以重用。 回复 更多评论

 # re: 深入浅出 Java Concurrency (11): 锁机制 part 6 CyclicBarrier 2011-03-04 17:02 cxb

上面的CyclicBarrier例子，效果并不理想：
当一组任务执行完后才一次性按下面的顺序打印出：
run over Left 50% Left 30%
而并不是(非一次性)
Left 50% Left 30% run over