

33 分阶段执行你的任务-学习使用Phaser运行多阶段任务

更新时间：2019-12-19 09:43:24



“青年是学习智慧的时期，中年是付诸实践的时期。

—— 卢梭”

本节我们学习的 **Phaser**，自 **Java 7** 出现，在功能上和 **CyclicBarrier** 及 **CountDownLatch** 很相似，不过更为灵活。我们回想 **CyclicBarrier** 的使用，在初始化时需要指定参与者数量，并且无法更改。而 **Phaser** 可以灵活的添加参与者，以及动态注销参与者，从而更加灵活地协同线程工作。

1、Phaser API 介绍

Phaser 从名称可以看出，它对线程协同的重点是任务阶段。**phaser** 中维护了所处阶段的数值。其实 **CyclicBarrier** 也可以实现类似的功能，但无法应对更为复杂的场景。**Phaser** 会更为的灵活，这体现着它对参与者的动态增减。并且参与者可以选择到达屏障点后是否阻塞。我们先看 **Phaser** 中涉及到的两个重要概念：

1. **phase**。**Phaser** 对阶段进行管理，而 **phase** 就是阶段，可以是阶段 1、阶段 2、阶段 3..... 当所有的参与者到达某个阶段屏障点时，**phaser** 会进入下一个阶段；
2. **party**。参与者，**Phase r** 中会记录参与者的数量，可以通过 **register** 方法来添加，或者通过 **arriveAndDeregister** 来注销。

接下来我们看一下 **Phaser** 的主要 API：

1. **register ()**：参与者数量加一；
2. **arrive ()**：参与者到达屏障点，到达数量加一。但是不会阻塞调用此方法的线程；
3. **arriveAndAwaitAdvance ()**：参与者到达屏障点，到达数量加一。阻塞线程直到所有的参与者到达该 **phase** 轮次；

4. `arriveAndDeregister ()`: 参与者到达屏障点，到达数量加一。然后从 `Phase` 注销掉一个参与者，参与者减一；
5. `awaitAdvance (int phase)`: 阻塞所有的参与者到达该 `phaser` 的指定轮次。如果当前轮次和 `phase` 值不同或者 `phase` 已被终止时，会立即返回；
6. `awaitAdvanceInterruptibly (int phase)`: 功能同上，但是可以被打断；
7. `awaitAdvanceInterruptibly (int phase, long timeout, TimeUnit unit)`: 功能同上，但是只阻塞指定的时长；
8. `bulkRegister (int parties)`: 批量注册参与者；
9. `forceTermination ()`: 终止当前 `phaser`，改变其状态为 `termination`；
10. `onAdvance ()`: 阶段达成时被调用，子类可以对其重写。。

2、Phaser 使用示例

网上有很多 `Phaser` 的使用范例，但其实绝大多数并没有体现出 `Phaser` 的优势来，看完之后反而觉得用 `CyclicBarrier` 也是能直接实现。其实 `Phaser` 的优势体现在对参与者数量动态管理上。下面我们写一个简单的例子，来看看 `Phaser` 如何使用。

我们设想如下场景：期末考试到了，软件学院三个班共有 60 个学生一起参加考试，全部交卷后，有 3 个老师做判卷的工作，再由 3 位辅导员公布成绩。

这个过程中分为三个阶段：

1. 学生考试，参与者 50
2. 老师判卷，参与者 3
3. 辅导员公布成绩，参与者 3

这个过程使用一个 `CyclicBarrier` 是无法实现的，因为 `CyclicBarrier` 的参与者数量无法变化。为了日志的简洁，下面的代码只模拟 10 个学生考试：

```

public class Client {
    public static void main(String[] args) {
        Phaser phaser = new Phaser();
        //主线程注册
        phaser.register();
        //10个学生线程分别启动开始考试，然后交卷，交卷后通知phaser已到达并且注销
        IntStream.range(1,10).forEach(number->{
            phaser.register();
            Thread student= new Thread(()->{
                try {
                    TimeUnit.SECONDS.sleep(5);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("学生"+number+"交卷");
                phaser.arriveAndDeregister();
            });
            student.start();
        });
        //学生并行考试时，主线程会先执行到此行代码，但由于本phase还没有达成，所以阻塞在此
        phaser.arriveAndAwaitAdvance();
        //所有学生达成后，开始新的phase，下面启动三个老师线程，开始判卷
        IntStream.range(1,3).forEach(number->{
            phaser.register();
            Thread teacher= new Thread(()->{
                try {
                    TimeUnit.SECONDS.sleep(3);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println("老师"+number+"判卷完成");
                phaser.arriveAndDeregister();
            });
            teacher.start();
        });
        //老师判卷时，主线程会先执行到此行代码，但由于本phase还没有达成，所以阻塞在此
        phaser.arriveAndAwaitAdvance();
        //所有老师都达成后，开始新的phase，下面启动三个辅导员线程，公布成绩
        IntStream.range(1,3).forEach(number->{
            phaser.register();
            Thread counsellor= new Thread(()->{
                System.out.println("辅导员"+number+"公布成绩完成");
                phaser.arriveAndDeregister();
            });
            counsellor.start();
        });
    }
}

```

- 1、首先主线程进行 `register`，因为主线程要使用 `phaser` 来控制流程，它也是参与者之一；
- 2、然后起 10 个学生线程考试、交卷。注意起线程前需要通过 `phaser.register ()` 来注册参与者；
- 3、接下来主线程 `phaser.arriveAndAwaitAdvance ()`；这个方法会阻塞，直到所有的子线程执行了 `phaser.arriveAndDeregister ()`，此时进入下一个 phase；
- 4、创建三个 `teacher` 线程进行判卷，和 `student` 线程一样，需要先注册自己，输出判卷完成后，调用 `phaser.arriveAndAwaitAdvance ()`，通知 `phaser` 自己已经到达并且要注销；
- 5、主线程还是调用 `phaser.arriveAndAwaitAdvance ()`；阻塞，等待所有老师线程 `arrive`。然后继续执行；
- 6、创建 3 个 `counsellor` 线程。先注册自己，公布成绩后，调用 `phaser.arriveAndAwaitAdvance ()`，通知 `phaser` 自己已经到达并且要注销。

主流程通过 `phaser.arriveAndAwaitAdvance ()` 来阻塞，控制主流程在上一 `phase` 完成后才进入下个 `phase`。在每个 `phase` 中会有多个线程同时执行。

程序输出如下：

```
学生8交卷
学生1交卷
学生9交卷
学生2交卷
学生3交卷
学生6交卷
学生7交卷
学生4交卷
学生5交卷
学生10交卷
老师3判卷完成
老师1判卷完成
老师2判卷完成
辅导员1公布成绩完成
辅导员2公布成绩完成
辅导员3公布成绩完成
```

可以看到和我们预想的一模一样。阶段间串行，阶段内并行。

下面总结一下我们使用到的 `phaser` 的方法：

- 1、`new Phaser ()`。创建新的 `Phaser`，并且参与者为 0；
- 2、`phaser.register ()`；增加参与者；
- 3、`phaser.arriveAndDeregister ()`；参与者到达，并且注销掉参与者。这个方法不会阻塞；
- 4、`phaser.arriveAndAwaitAdvance ()`；阻塞等待阶段达成。

3、Phaser 实现原理解析

下面我们分析一下 `Phaser` 的实现原理。我们先来理解 `Phaser` 中有一个很关键的属性 `status`。

```
private volatile long state;
```

这个 `long` 类型的 `status` 在不同 `bit` 位保存了 `Phaser` 状态相关的四种属性，具体如下：

0-15 位：还未到达屏障的参与者数量

16-31 位：参与者数量

32-62 位：`phase` 的轮次

63 位：标识是否被终止

可以看到与 `Phaser` 状态相关的数据都包含在 `state` 之中。不分开保存的原因是多个属性不能通过 `CAS` 的方式做原子操作。把这些属性组合起来，可以通过 `CAS` 方式更新确保线程安全，并且变相做到了多个属性更新的原子操作。

`Phaser` 中有两个链表保存等待的线程：

```
private final AtomicReference<QNode> evenQ;  
private final AtomicReference<QNode> oddQ;
```

这是为了消除添加和释放线程等待的争抢。所以根据 **phaser** 轮次的奇偶，保存在不同的链表中。

这里就不再展开将 **Phaser** 的源代码了，简单讲一下源代码中的实现原理。首先我们知道 **state** 保存了 4 种状态，所以更新状态的时候要把 **status** 中相应属性增减的数值换算为相应位数对应的整数，然后通过 **CAS** 的方式进行修改。

比如通过调用 **arrive** 方法，需要减少一个未到达屏障的参与者，也就是要对 **state** 的 0-15 位 - 1。由于为低位 - 1，所以直接对 **state** 减一即可。如下，**adjust** 的值为 1：

```
(UNSAFE.compareAndSwapLong(this, stateOffset, s, s-=adjust))
```

如果调用 **arriveAndDeregister** 方法，减少一个未到达屏障的参与者，并且还要减少一个参与者。相当于对 0-15 位减 1，并且对 16-31 位减 1，对应的二进制数值就是 10000000000000001，转化为 10 进制为 65537。那么需要对 **status** 减掉 65537。我们看一下 **arriveAndDeregister** 方法：

```
public int arriveAndDeregister() {  
    return doArrive(ONE_DEREGISTER);  
}
```

我们看到调用 **doArrive** 时传入的参数是 **ONE_DEREGISTER**，它的值如下：

```
private static final int ONE_DEREGISTER = ONE_ARRIVAL|ONE_PARTY;
```

ONE_ARRIVAL= 1， **ONE_PARTY**=1 << **PARTIES_SHIFT**, **PARTIES_SHIFT**=16。也就是 **ONE_PARTY** 的值是 1 向左移 16 位。那么 **ONE_ARRIVAL|ONE_PARTY** 得出的二进制就是 10000000000000001。正是我们按照需求推断出的二进制数值。

此外为了取出 **state** 中相应 **bit** 位数区间的状态值，**Phaser** 是通过位移或者 **&**、**|** 操作来实现，例如取得 **phase** 轮次值的代码如下：

```
int phase = (int)(s >>> PHASE_SHIFT);
```

PHASE_SHIFT 为 32。将 **state** 值右移 32 位，这意味着把代表 **phase** 轮次的 32-63 位 **bit** 数值移到了 0-32 位。然后强转 **int** 类型，消除掉高 32 位。这样就得到了 **phase** 的轮次真正数值。

Phaser 对 **state** 的操作方式都是这种二进制的方式，一开始看起来会比较费劲，但是理解了它的原理，也并不复杂。其他的关于等待线程的管理、线程阻塞和恢复，和之前我们分析的源代码大同小异。大家感兴趣的话，也可以看一看。

4、总结

Phaser 提供了分阶段执行任务的功能，并且能够动态的改变参与者的数量，和 **CyclicBarrier** 以及 **CountDownLatch** 比较起来更为灵活。这也是 **JDK** 的文档中所提到的。实际开发中按照需求选择使用。

}

