二

53 CountDownLatch 是如何安排线程执行顺序的?

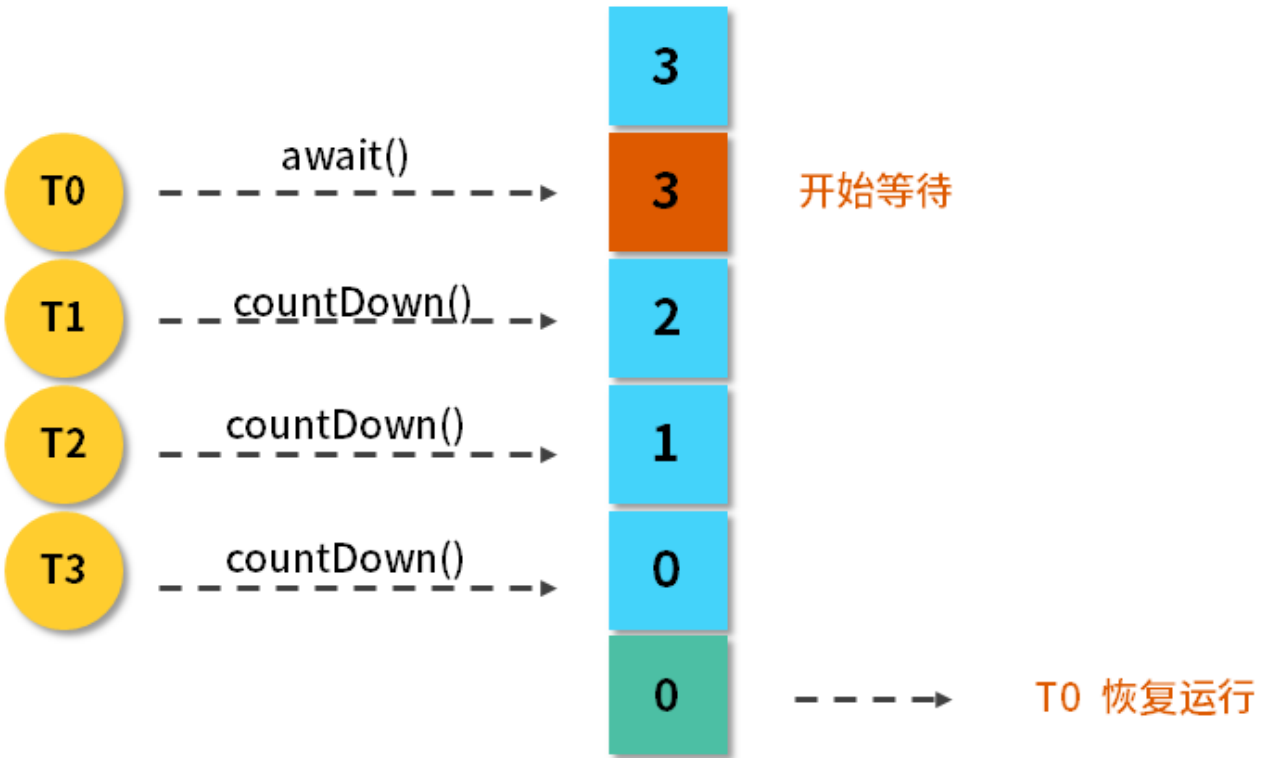
本课时我们主要介绍 CountDownLatch 是如何安排线程执行顺序的。

我们先来介绍一下 CountDownLatch，它是 JDK 提供的**并发流程控制**的工具类，它是在 java.util.concurrent 包下，在 JDK1.5 以后加入的。下面举个例子来说明它主要在什么场景下使用。

比如我们去游乐园坐激流勇进，有的时候游乐园里人不是那么多，这时，管理员会让你稍等一下，等人坐满了再开船，这样的话可以在一定程度上节约游乐园的成本。座位有多少，就需要等多少人，这就是 **CountDownLatch** 的核心思想，等到一个设定的数值达到之后，才能出发。

流程图

我们把激流勇进的例子用流程图的方式来表示：



可以看到，最开始 CountDownLatch 设置的初始值为 3，然后 T0 线程上来就调用 await 方法，它的作用是让这个线程开始等待，等待后面的 T1、T2、T3，它们每一次调用 countDown 方法，3 这个数值就会减 1，也就是从 3 减到 2，从 2 减到 1，从 1 减到 0，一旦减到 0 之后，这个 T0 就相当于达到了自己触发继续运行的条件，于是它就恢复运行了。

主要方法介绍

下面介绍一下 CountDownLatch 的主要方法。

（1）构造函数：public CountDownLatch(int count) { };

它的构造函数是传入一个参数，该参数 count 是需要倒数的数值。

（2）await()：调用 await() 方法的线程开始等待，直到倒数结束，也就是 count 值为 0 的时候才会继续执行。

（3）await(long timeout, TimeUnit unit)：await() 有一个重载的方法，里面会传入超时参数，这个方法的作用和 await() 类似，但是这里可以设置超时时间，如果超时就不再等待了。

（4）countDown()：把数值倒数 1，也就是将 count 值减 1，直到减为 0 时，之前等待的线程会被唤起。

用法

接着来介绍一下 CountDownLatch 的两个典型用法。

用法一：一个线程等待其他多个线程都执行完毕，再继续自己的工作

在实际场景中，很多情况下需要我们初始化一系列的前置条件（比如建立连接、准备数据），在这些准备条件都完成之前，是不能进行下一步工作的，所以这就是利用 CountDownLatch 的一个很好场景，我们可以让应用程序的主线程在其他线程都准备完毕之后再继续执行。

举个生活中的例子，那就是运动员跑步的场景，比如在比赛跑步时有 5 个运动员参赛，终点有一个裁判员，什么时候比赛结束呢？那就是当所有人都跑到终点之后，这相当于裁判员等待 5 个运动员都跑到终点，宣布比赛结束。我们用代码的形式来写出运动员跑步的场景，代码如下：

```
public class RunDemo1 {

    public static void main(String[] args) throws InterruptedException {

        CountDownLatch latch = new CountDownLatch(5);

        ExecutorService service = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 5; i++) {

            final int no = i + 1;

            Runnable runnable = new Runnable() {

                @Override

                public void run() {

                    try {

                        Thread.sleep((long) (Math.random() * 10000));

                        System.out.println(no + "号运动员完成了比赛。");

                    } catch (InterruptedException e) {

                        e.printStackTrace();

                    } finally {

                        latch.countDown();

                    }

                }

            };

            service.submit(runnable);

        }

        System.out.println("等待5个运动员都跑完.....");

        latch.await();

        System.out.println("所有人都跑完了，比赛结束。");

    }

}
```

在这段代码中，我们新建了一个初始值为 5 的 CountDownLatch，然后建立了一个固定 5 线程的线程池，用一个 for 循环往这个线程池中提交 5 个任务，每个任务代表一个运动员，

这个运动员会首先随机等待一段时间，代表他在跑步，然后打印出他完成了比赛，在跑完了之后，同样会调用 countDown 方法来把计数减 1。

之后我们再回到主线程，主线程打印完“等待 5 个运动员都跑完”这句话后，会调用 await() 方法，代表让主线程开始等待，在等待之前的那几个子线程都执行完毕后，它才会认为所有人都跑完了比赛。这段程序的运行结果如下所示：

```
等待5个运动员都跑完.....
4号运动员完成了比赛。
3号运动员完成了比赛。
1号运动员完成了比赛。
5号运动员完成了比赛。
2号运动员完成了比赛。
所有人都跑完了，比赛结束。
```

可以看出，直到 5 个运动员都完成了比赛之后，主线程才会继续，而且由于子线程等待的时间是随机的，所以各个运动员完成比赛的次序也是随机的。

用法二：多个线程等待某一个线程的信号，同时开始执行

这和第一个用法有点相反，我们再列举一个实际的场景，比如在运动会上，刚才说的是裁判员等运动员，现在是运动员等裁判员。在运动员起跑之前都会等待裁判员发号施令，一声令下运动员统一起跑，我们用代码把这件事情描述出来，如下所示：

```
public class RunDemo2 {

    public static void main(String[] args) throws InterruptedException {

        System.out.println("运动员有5秒的准备时间");

        CountDownLatch countDownLatch = new CountDownLatch(1);

        ExecutorService service = Executors.newFixedThreadPool(5);

        for (int i = 0; i < 5; i++) {

            final int no = i + 1;

            Runnable runnable = new Runnable() {

                @Override
```

```
                public void run() {
                    System.out.println(no + "号运动员准备完毕，等待裁判员的发令枪");
                    try {
                        countDownLatch.await();
                        System.out.println(no + "号运动员开始跑步了");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    }
                }
            };
            service.submit(runnable);
        }
        Thread.sleep(5000);
        System.out.println("5秒准备时间已过，发令枪响，比赛开始！");
        countDownLatch.countDown();
    }
}
```

在这段代码中，首先打印出了运动员有 5 秒的准备时间，然后新建了一个 CountDownLatch，其倒数值只有 1；接着，同样是一个 5 线程的线程池，并且用 for 循环的方式往里提交 5 个任务，而这 5 个任务在一开始时就让它调用 await() 方法开始等待。

接下来我们再回到主线程。主线程会首先等待 5 秒钟，这意味着裁判员正在做准备工作，比如他会喊“各就各位，预备”这样的话语；然后 5 秒之后，主线程会打印出“5 秒钟准备时间已过，发令枪响，比赛开始”的信号，紧接着会调用 countDown 方法，一旦主线程调用了该方法，那么之前那 5 个已经调用了 await() 方法的线程都会被唤醒，所以这段程序的运行结果如下：

```
运动员有5秒的准备时间
2号运动员准备完毕，等待裁判员的发令枪
1号运动员准备完毕，等待裁判员的发令枪
3号运动员准备完毕，等待裁判员的发令枪
```

```
4号运动员准备完毕，等待裁判员的发令枪
5号运动员准备完毕，等待裁判员的发令枪
5秒准备时间已过，发令枪响，比赛开始！
2号运动员开始跑步了
1号运动员开始跑步了
5号运动员开始跑步了
4号运动员开始跑步了
3号运动员开始跑步了
```

可以看到，运动员首先会有 5 秒钟的准备时间，然后 5 个运动员分别都准备完毕了，等待发令枪响，紧接着 5 秒之后，发令枪响，比赛开始，于是 5 个子线程几乎同时开始跑步了。

注意点

下面来讲一下 CountDownLatch 的注意点：

- 刚才讲了两种用法，其实这两种用法并不是孤立的，甚至可以把这两种用法结合起来，比如利用两个 CountDownLatch，第一个初始值为多个，第二个初始值为 1，这样就可以应对更复杂的业务场景了；
- CountDownLatch 是不能够重用的，比如已经完成了倒数，那可不可以在下一次继续去重新倒数呢？这是做不到的，如果你有这个需求的话，可以考虑使用 CyclicBarrier 或者创建一个新的 CountDownLatch 实例。

总结

CountDownLatch 类在创建实例的时候，需要在构造函数中传入倒数次数，然后由需要等待的线程去调用 await 方法开始等待，而每一次其他线程调用了 countDown 方法之后，计数便会减 1，直到减为 0 时，之前等待的线程便会继续运行

上一页 下一页