28 人齐了，一起行动—CyclicBarrier详解

更新时间：2019-12-03 09:55:08

> 青年是学习智慧的时期，中年是付诸实践的时期。
>
> —— 卢梭

上一节我们讲解了 CountDownLatch，它的作用是让多个线程完成后，再促使主线程继续向下执行。不过它有一定的局限性，无法被重复使用。本节我们学习的 CyclicBarrier 不会有这个问题。CyclicBarrier 从字面上理解为循环栅栏。栅栏自然起到的就是屏障的作用，阻止线程通过，而循环则是指其可以反复使用。下面我们就先看看如何使用 CyclicBarrier。

1、CyclicBarrier 的使用

几年前北京的黑车盛行，西二旗地铁口，大量黑车司机在出口招揽生意：“软件园、软件园！5 块一位！还差最后一位！”。等你上车，发现其实不是还差一位，而是只有你一位。而司机此时绝对不会发车，而是会等车上坐够 4 个人后才出发，然后下一辆黑车再次坐满 4 人后发车。下面我们就使用 CyclicBarrier 来模拟这个场景。

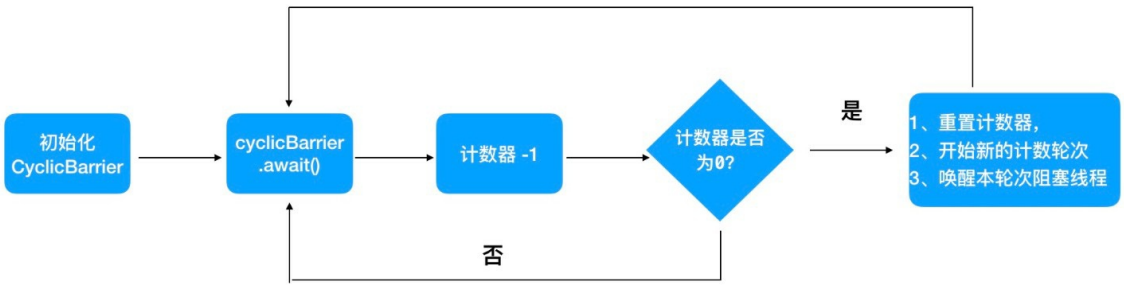
```
public class Client {
    public static void main(String[] args) {
        CyclicBarrier cyclicBarrier = new CyclicBarrier(4, () ->
                System.out.println("人满了发车")
        );

        IntStream.range(1, 11).forEach(number -> {
            try {
                Thread.currentThread().sleep(100);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }

            new Thread(() -> {
                try {
                    System.out.println("第 " + number + " 乘客上车了！ ");
                    cyclicBarrier.await();
                    System.out.println("第 " + number + " 乘客出发了！ ");
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }).start();
        });
    }
}
```

代码中首先声明 cyclicBarrier 对象，构造方法有两个参数，第一个参数是计数器初始值，每有一个线程达成则会减 1 。减到 0 时，触发执行第二个参数传入的 Runnable 实现的 run 方法。我这里使用 lambda 的方式简化代码。如果你不需要这个 Runnable 的任务，那么只需要传入第一个参数即可。

接下来的代码中，模拟 10 位乘客上车，每次上车后调用 cyclicBarrier.await() 。这里就是屏障点，此时当前线程会阻塞在此处，并且计数器被减 1 。为了输出的效果便于观看，每次新线程启动前先 sleep 一会。

每当四个乘客完成上车操作，cyclicBarrier 就会触发“人满了发车”的操作。而最后两位乘客上车后，由于没有新的乘客上车，计数器不会被减到 0，导致无法越过屏障，所以永远不会发车。

cyclicBarrier运行的示意图如下：



注意：cyclicBarrier.await()被多个线程调用。
每个线程调用时，计数器会-1，并且被阻塞。

代码运行输出如下：

```
第 1 乘客上车了！
第 2 乘客上车了！
第 3 乘客上车了！
第 4 乘客上车了！
人满了发车
第 4 乘客出发了！
第 1 乘客出发了！
第 2 乘客出发了！
第 3 乘客出发了！
第 5 乘客上车了！
第 6 乘客上车了！
第 7 乘客上车了！
第 8 乘客上车了！
人满了发车
第 8 乘客出发了！
第 5 乘客出发了！
第 7 乘客出发了！
第 6 乘客出发了！
第 9 乘客上车了！
第 10 乘客上车了！
```

可以看到每上车 4 人，才会触发发车，同时每个人的线程才会继续 cyclicBarrier.await() 后面的代码，输出 “第 n 乘客出发了！”

这个例子也验证了 CyclicBarrier 可以重复使用，每次满 4 人上车，都会触发发车。然后重新开始计数。

通过这个例子我们了解了 CyclicBarrier 的使用。在这里我们总结下 CyclicBarrier 涉及的几个概念：

1、计数器。初始值为构造 CyclicBarrier 传入的第一个参数，每当一个线程到达屏障点，计数器减1；

2、屏障点，线程中调用 cyclicBarrier.await() 后，该线程到达屏障点，等待 CyclicBarrier 打开，也就是计数器到 0；

3、冲出屏障后的任务。首先这个任务可选。不需要的话，在构造 CyclicBarrier 时只需要传入计数器初始值即可。这个任务在计数器到 0时被触发。

2、CyclicBarrier 原理解析

2.1、 CyclicBarrier 中的属性

```
/** CyclicBarrier使用的拍他锁*/
private final ReentrantLock lock = new ReentrantLock();
/** barrier被冲破前，线程等待的condition*/
private final Condition trip = lock.newCondition();
/** barrier被冲破时，需要满足的参与线程数。*/
private final int parties;
/* barrier被冲破后执行的方法。*/
private final Runnable barrierCommand;
/** 当其轮次 */
private Generation generation = new Generation();

/**
 *目前等待剩余的参与者数量。从 parties倒数到0。每个轮次该值会被重置回parties
 */
private int count;
```

可以看到 CyclicBarrier 内部通过 ReentrantLock 来实现的，而 ReentrantLock 的底层实现还是 AQS。

parties 在构造函数中被赋值，它的值永远不会变，因为 CyclicBarrier 会被重置复用。而每个轮次真正用来计数的变量是 count。每个轮次结束，count 会被重置为 parties 的值。

2.2、 await() 方法解析

await 方法的调用，代表调用线程到达了屏障点，这个方法其实调用了 dowait 方法，我们直接分析 dowait 方法，它实现了 CyclicBarrier 的核心功能。

```
/**
 * Main barrier code, covering the various policies.
 */
private int dowait(boolean timed, long nanos)
    throws InterruptedException, BrokenBarrierException,
           TimeoutException {
    final ReentrantLock lock = this.lock;
    //对共享资源count，generation操作前，需要先上锁保证线程安全
    lock.lock();
    try {
      //拿到当前轮次对象的引用
       final Generation g = generation;
    //如果已经broken，那么抛出异常
       if (g.broken)
          throw new BrokenBarrierException();
    //如果被打断，通过breakBarrier方法设置当前轮次为broken状态，通知当前轮次所有等待的线程线程
      //并且抛出InterruptedException
       if (Thread.interrupted()) {
          breakBarrier();
          throw new InterruptedException();
       }
    //count减1
       int index = --count;
      //如果index为0，那么冲破屏障点
       if (index == 0) {  // tripped
          boolean ranAction = false;
          //冲破屏障点后，如果CyclicBarrier构造时传入Runnable，则被调用。
          try {
             final Runnable command = barrierCommand;
             if (command != null)
                command.run();
             ranAction = true;
             //这个方法中会进行重置，并且通知所有在屏障点阻塞的线程继续执行。
             nextGeneration();
             return 0;
          } finally {
            //正常情况由于运行了command后ranAction被置为true，并不会执行如下逻辑
            //在command执行期间出了异常才会进入下面的逻辑，认为当前轮次被破坏了
             if (!ranAction)
                breakBarrier();
          }
       }

      //开始自旋，直到屏障被冲破，或者interrupted或者超时
       for (;;) {
          try {
             if (!timed)
               //阻塞，此时会释放锁，以让其他线程进入await方法中。等待屏障被冲破后，向后执行
                trip.await();
             else if (nanos > 0L)
                nanos = trip.awaitNanos(nanos);
          } catch (InterruptedException ie) {
            //如果当前线程阻塞被interrupt了，并且本轮次还没有被break，那么修改本轮次状态为broken
             if (g == generation && ! g.broken) {
                breakBarrier();
                throw ie;
             } else {
                Thread.currentThread().interrupt();
```

```
            }
        }
    //如果本轮次被破坏，那么抛出异常
        if (g.broken)
            throw new BrokenBarrierException();

        //如果已经成功进入下一轮次，那么返回index
        if (g != generation)
            return index;
    //如果已经超时，那么本轮次被打破
        if (timed && nanos <= 0L) {
            breakBarrier();
            throw new TimeoutException();
        }
      }
    } finally {
     //释放锁
      lock.unlock();
    }
}
```

以上代码分为两大段逻辑，分别是自旋前，和自旋。

A、自旋前的逻辑，核心逻辑如下：

1. 计数器 -1；
2. 判断是否计数器到 0；
3. 如果到了，则冲破屏障点，执行传入的 Runnable；
4. 调用 nextGeneration() 来更新 Generation，重置计数器，并且通知本轮次等待的线程。

B、如果计数器没有到 0，则进入自旋的逻辑：

1. 开始等待，此时会释放锁，以让其它线程进入 lock 的代码块执行以上逻辑；
2. 当被唤醒时，可能因为当前 generation 被 break 了，或者计数器到 0，屏障被冲破；
3. 对比边刚进入 dowait 方法时获取的 generation 对象和最新 generation 是否一致。不一致说明已经换代了，也就是屏障被冲破，可以 return 了；
4. 如果等待超时或者 generation 被 break，分别抛出异常。

不同线程在 A 部分的逻辑会影响已经进入 B 部分逻辑的线程中止自旋。这些自旋的线程或者冲破屏障点，继续向下执行，也可能抛出异常。

我们再看下用于更新轮次的方法 nextGeneration()：

```
private void nextGeneration() {
    // signal completion of last generation
    trip.signalAll();
    // set up next generation
    count = parties;
    generation = new Generation();
}
```

三行代码做了三件事：

1、通知所有被阻塞在本轮次屏障点的线程。屏障点被冲破，可以继续向下执行了；

2、重置计数器为初始值；

3、更新轮次对象。这样自旋中的线程才会跳出自旋。

3、总结

CyclicBarrier 和 CountDownLatch 相比，更为灵活，可以被重复使用。前者可以用来分段任务，假如有个任务需要分三个阶段来完成，每个阶段可以多线程并发执行，但是进入下一个阶段的时候，必须所有线程都完成了第一阶段的执行。那么通过 CyclicBarrier，在每个线程的每个阶段开始前都设置屏障点，可以很轻松地实现。

CyclicBarrier 的实现是通过 ReentrantLock 控制计数器的原子更新，通过条件变量来实现线程同步。

}

27 倒数计时开始，三、二、一——CountDownLatch详解

29 一手交钱，一手交货——Exchanger详解