

CyclicBarrier的源码分析

与CountDownLatch、Semaphore直接基于AQS实现不同，CyclicBarrier 是基于 ReentrantLock + ConditionObject 实现的，间接基于AQS实现的。

CyclicBarrier内部结构

- Generation，静态内部类，持有布尔类型的属性broken，默认为false，只有在重置方法reset()、执行出现异常或中断调用breakBarrier()，属性会被设置为true。
- nextGenerate() 重置 CyclicBarrier 的计数器和generation属性。
- breakBarrier() 任务执行中断、异常、被重置，将Generation中的布尔类型属性设置为true，将Waiter队列中的线程转移到AQS队列中，待执行完unlock方法后，唤醒AQS队列中的挂起线程。
- await()：CyclicBarrier的核心方法，计数器递减处理。

The image shows a screenshot of the CyclicBarrier source code in an IDE. Red arrows point from Chinese annotations to specific code elements. The annotations are as follows:

- 持有布尔型属性，默认为false** (Holds a boolean attribute, default is false) - points to the `broken` field in the `Generation` class.
- 表示当前任务是否被中断、执行异常、被重置** (Indicates whether the current task is interrupted, execution exception, or reset) - points to the `broken` field.
- 有参构造函数** (Parameterized constructor) - points to the `CyclicBarrier(int, Runnable)` constructor.
- 执行结束，重置CyclicBarrier的计数器，generation属性** (After execution ends, reset the counter and generation attribute of CyclicBarrier) - points to the `nextGeneration()` method.
- CyclicBarrier的核心方法，计数器递减操作就在await方法中** (CyclicBarrier's core method, the counter decrement operation is in the `await` method) - points to the `await()` method.
- ReentrantLock + Condition，控制线程的挂起与唤醒** (ReentrantLock + Condition, control thread suspension and wakeup) - points to the `lock` and `trip` fields.
- 自实现的Runnable** (Self-implemented Runnable) - points to the `barrierCommand` field.
- 计数值** (Count value) - points to the `count` field.

构造函数

构造参数重载，最终调用的是CyclicBarrier(int, Runnable)，详情如下：

```
1 public CyclicBarrier(int parties) {
2     this(parties, null);
3 }
4
5 public CyclicBarrier(int parties, Runnable barrierAction) {
6     // 参数合法性校验
7     if (parties <= 0) throw new IllegalArgumentException();
8     // final修饰，所有线程执行完成归为或重置时 使用
9     this.parties = parties;
10    // 在await方法中计数值，表示还有多少线程待执行await
11    this.count = parties;
12    // 当计数count为0时 ，执行此Runnable，再唤醒被阻塞的线程
13    this.barrierCommand = barrierAction;
14 }
```

CyclicBarrier属性

核心方法源码分析

await()

在CyclicBarrier中，await有重载方法。await()表示会一直等待指定数量的线程未准备就绪(执行await方法)；await(timeout, unit)表示等待timeout时间后，指定数量的线程未准备就绪，抛出TimeoutException超时异常。

CyclicBarrier#await 详情如下：

```
1 // 执行没有超时时间的await
2 public int await() throws InterruptedException, BrokenBarrierException {
3     try {
4         // 执行dowait()
5         return dowait(false, 0L);
6     } catch (TimeoutException toe) {
7         throw new Error(toe);
8     }
9 }
10
11 // 执行有超时时间的await
12 public int await(long timeout, TimeUnit unit)
13     throws InterruptedException,
14         BrokenBarrierException,
15         TimeoutException {
16     return dowait(true, unit.toNanos(timeout));
17 }
```

await最终调用dowait()方法, CyclicBarrier#dowait 详情如下:

```

1 private int dowait(boolean timed, long nanos) throws InterruptedException,
  BrokenBarrierException, TimeoutException {
2     // 获取锁对象
3     final ReentrantLock lock = this.lock;
4     // 加锁
5     lock.lock();
6     try {
7         // 获取generation对象
8         final Generation g = generation;
9
10        // 这组线程中在执行过程中是否异常、超时、中断、重置
11        if (g.broken)
12            throw new BrokenBarrierException();
13
14        // 这组线程被中断，重置标识与计数值，
15        // 将Waiter队列中的线程转移到AQS队列，抛出InterruptedException
16        if (Thread.interrupted()) {
17            breakBarrier();
18            throw new InterruptedException();
19        }
20
21        // 计数值 - 1
22        int index = --count;
23        // 这组线程都已准备就绪
24        if (index == 0) {
25            // 执行结果标识
26            boolean ranAction = false;
27            try {
28                // 若使用2个参数的有参构造，就传入了自实现任务，index == 0，先执行
                // CyclicBarrier有参的任务
29                // 此处设计与 FutureTask 构造参数设计类似
30                final Runnable command = barrierCommand;
31                if (command != null)
32                    // 执行任务
33                    command.run();
34                // 执行完成，设置为true
35                ranAction = true;
36                // CyclicBarrier属性归位
37                nextGeneration();

```

```

38         return 0;
39     } finally {
40         // 执行过程中出现问题
41         if (!ranAction)
42             // 重置标识与计数值，将Waiter队列中的线程转移到AQS队列
43             breakBarrier();
44     }
45 }
46
47 // -- 之后，count不为0，表示还有线程在等待
48 // 自旋 直到被中断、超时、异常、count = 0
49 for (;;) {
50     try {
51         // 未设置超时时间
52         if (!timed)
53             // 挂起线程，将线程转移到 Condition 队列
54             trip.await();
55         // 未达到等待时间
56         else if (nanos > 0L)
57             // 挂起线程，并返回剩余等待时间
58             nanos = trip.awaitNanos(nanos);
59     } catch (InterruptedException ie) {
60         // 中断异常
61         if (g == generation && ! g.broken) {
62             breakBarrier();
63             throw ie;
64         } else {
65             // 线程中断
66             Thread.currentThread().interrupt();
67         }
68     }
69
70     // 该组线程被中断、执行异常、超时，抛出BrokenBarrierException异常
71     if (g.broken)
72         throw new BrokenBarrierException();
73
74     if (g != generation)
75         return index;
76

```

```
77         // 超时，抛出异常TimeoutException
78         if (timed && nanos <= 0L) {
79             breakBarrier();
80             throw new TimeoutException();
81         }
82     }
83 } finally {
84     // 释放锁资源
85     lock.unlock();
86 }
87 }
```

breakBarrier() - 结束CyclicBarrier的执行

```
1 // 结束CyclicBarrier的执行
2 private void breakBarrier() {
3     // 设置线程执行过程中是否异常、中断、重置标识
4     generation.broken = true;
5     // 重置计数值
6     count = parties;
7     // 将Condition队列中的Node转移到AQS队列中，等到执行完unlock，AQS队列中的挂起线程会被唤醒
8     // 有后继节点的，设置ws = -1;
9     // 无后继节点的，设置ws = 0
10    trip.signalAll();
11 }
```

reset() - 重置CyclicBarrier

```

1 // 重置CyclicBarrier
2 public void reset() {
3     // 获取锁对象
4     final ReentrantLock lock = this.lock;
5     // 加锁
6     lock.lock();
7     try {
8         // 设置当前generation属性，并将Waiter队列中线程转移到AQS队列
9         breakBarrier();
10        // 重置generation 属性、计数值
11        nextGeneration();
12    } finally {
13        // 释放锁
14        lock.unlock();
15    }
16 }

```

nextGeneration() - CyclicBarrier归位

```

1 private void nextGeneration() {
2     // 将Waiter队列中线程转移到AQS队列
3     trip.signalAll();
4     // 计数值、generation 归位
5     count = parties;
6     generation = new Generation();
7 }

```

总结

CyclicBarrier基于 ReentrantLock + ConditionObject实现，CyclicBarrier的构造函数中必须指定 parties，同时对象generation，内部持有布尔型属性表示当前CyclicBarrier执行过程中是否有超时、异常、中断的情况。

parties是初始待执行线程数，在构造函数中会将parties赋给计数值count，每当一个线程执行 await()，count就会减1。

当count被减为0时，代表所有线程都准备就绪，此时判断构造函数是否初始化了barrierCommand属性，若对barrierCommand属性做了赋值，优先执行barrierCommand任务；

barrierCommand任务执行完成，再将Waiter队列中的线程转移到AQS队列中，执行完unlock，唤醒AQS队列中的线程；计数值count、generation归位。