Java 并发编程之美：并发编程高级篇之五

一、前言

Java 并发编程实践中的话：编写正确的程序并不容易，而编写正常的并发程序就更难了。相比于顺序执行的情况，多线程的线程安全问题是微妙而且出乎意料的，因为在没有进行适当同步的情况下多线程中各个操作的顺序是不可预期的。

并发编程相比 Java 中其他知识点学习起来门槛相对较高，学习起来比较费劲，从而导致很多人望而却步；而无论是职场面试和高并发高流量的系统的实现却都还离不开并发编程，从而导致能够真正掌握并发编程的人才成为市场比较迫切需求的。

本 Chat 作为 Java 并发编程之美系列的高级篇之五，讲解 JUC 包中提供的三种线程同步器的使用与原理分析内容如下：（建议先阅读 并发编程高级篇之三 - 锁 ）

- JUC 中倒数计数器 CountDownLatch 的使用与原理分析，当需要等待多个线程执行完毕后在做一件事情时候 CountDownLatch 是比调用线程的 join 方法更好的选择，CountDownLatch 与 线程的 join 方法区别是什么?
- JUC 中 回环屏障 CyclicBarrier 的使用与分析，它也可以实现像 CountDownLatch 一样让一组线程全部到达一个状态后再全部同时执行，但是 CyclicBarrier 可以被复用。那么 CyclicBarrier 内部的实现与 CountDownLatch 有何不同那?
- JUC 中 Semaphore 的使用与原理分析，Semaphore 也是 Java 中的一个同步器，与 CountDownLatch 和 CycleBarrier 不同在于它内部的计数器是递增的，那么，Semaphore 的内部实现是怎样的那?
- 最后对上面三种同步器实现进行简单对比。

二、CountDownLatch 原理分析

2.1 案例介绍

日常开发中经常会遇到需要在主线程中开启多线程去并行执行任务，并且主线程需要等待所有子线程执行完毕后在进行汇总的场景，在 CountDownLatch 出现之前一般都是使用线程的 join() 方法来实现，但是 join 不够灵活，不能够满足不同场景的需要，下面看一个使用 CountDownLatch 的例子:

```
public class JoinCountDownLatch {
```

```
    // 创建一个CountDownLatch实例，管理计数为ThreadNum
    private static volatile CountDownLatch countDownLatch = new
CountDownLatch(2);

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

        Thread threadOne = new Thread(new Runnable() {

            @Override
            public void run() {

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                System.out.println("child threadOne over!");
                countDownLatch.countDown();

            }
        });

        Thread threadTwo = new Thread(new Runnable() {

            @Override
            public void run() {

                try {
                    Thread.sleep(1000);
                } catch (InterruptedException e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                System.out.println("child threadTwo over!");
                countDownLatch.countDown();

            }
        });

        // 启动子线程
        threadOne.start();
        threadTwo.start();

        System.out.println("wait all child thread over!");

        // 等待子线程执行完毕，返回
        countDownLatch.await();
```

```
            System.out.println("all child thread over!");

        }
    }
```

输出结果：

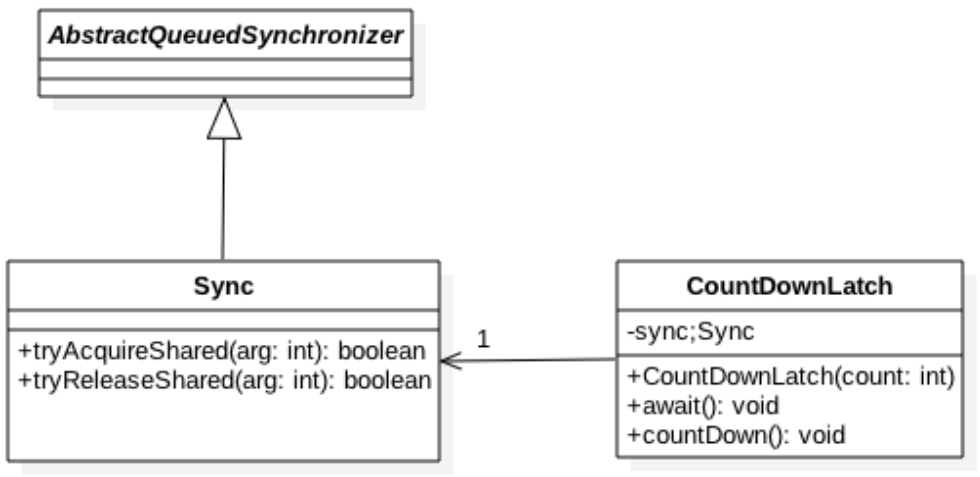
```
<terminated> JoinCountDownLatch [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
wait all child thread over!
child threadOne over!
child threadTwo over!
all child thread over!
```

如上代码，创建了一个 CountDownLatch 实例，因为有两个子线程所以构造函数参数传递为 2，主线程调用 countDownLatch.await（）方法后会被阻塞。子线程执行完毕后调用 countDownLatch.countDown() 方法让 countDownLatch 内部的计数器减一，等所有子线程执行完毕调用 countDown（）后计数器会变为 0，这时候主线程的 await（）才会返回。

注：CountDownLatch 与 join 方法的区别，一个区别是调用一个子线程的 join（）方法后，该线程会一直被阻塞直到该线程运行完毕，而 CountDownLatch 则使用计数器允许子线程运行完毕或者运行中时候递减计数，也就是 CountDownLatch 可以在子线程运行任何时候让 await 方法返回而不一定必须等到线程结束；另外使用线程池来管理线程时候一般都是直接添加 Runable 到线程池这时候就没有办法在调用线程的 join 方法了，countDownLatch 相比 Join 方法让我们对线程同步有更灵活的控制。

2.2 实现原理探究

从 CountDownLatch 的名字可以猜测内部应该有个计数器，并且这个计数器是递减的，下面就通过源码看看 JDK 开发组是何时初始化计数器，何时递减的，计数器变为 0 时候做了什么操作，多个线程是如何通过计时器值实现同步的，为了一览 CountDownLatch 内部结构，先看下类图：



从类图可知 CountDownLatch 内部还是使用 AQS 实现的，通过下面构造函数初始化了计数器的值，可知实际上是把计数器的值赋值给了 AQS 的状态值 state，也就是这里 AQS 的状态值来表示计数器值。

```
    public CountDownLatch(int count) {
        if (count < 0) throw new IllegalArgumentException("count
< 0");
        this.sync = new Sync(count);
    }

   Sync(int count) {
       setState(count);
   }
```

下面主要看下 CountDownLatch 中几个重要的方法内部是如何调用 AQS 来实现功能的

- void await() 方法
  当线程调用了 CountDownLatch 对象的 await 方法后，当前线程会被阻塞，直到下面的情况之一发生才会返回：
  （1）当所有线程都调用了 CountDownLatch 对象的 countDown 方法后，也就是计时器值为 0 的时候；
  （2）其它线程调用了当前线程的 interrupt（）方法中断了当前线程，当前线程会抛出 InterruptedException 异常后返回.

下面看下 await() 方法内部是如何调用 AQS 的方法的：

```
//CountDownLatch的await（）方法
public void await() throws InterruptedException {
   sync.acquireSharedInterruptibly(1);
}
```

```
    //AQS的获取共享资源时候可被中断的方法
    public final void acquireSharedInterruptibly(int arg)
            throws InterruptedException {
        //如果线程被中断则抛异常
        if (Thread.interrupted())
            throw new InterruptedException();
        //尝试看当前是否计数值为0，为0则直接返回，否者进入AQS的队列等待
        if (tryAcquireShared(arg) < 0)
            doAcquireSharedInterruptibly(arg);
    }

 //sync类实现的AQS的接口
 protected int tryAcquireShared(int acquires) {
            return (getState() == 0) ? 1 : -1;
        }
```

从代码可知 await() 方法委托 sync 调用了 AQS 的 acquireSharedInterruptibly 方法，该方法的特点是线程获取资源的时候可以被中断，并且获取的资源是共享资源，这里为何要用 AQS 的这个方法，而不是调用独占资源的 acquireInterruptibly 方法那？是因为这里状态值需要的并不是非 0 即 1 的效果，而是和初始化时候指定的计数器值有关系，比如你初始化时候计数器值为 8，那么 state 的值应该就有 0 到 8 的状态，而不是只有 0 和 1 的独占效果。

这里 await() 方法调用 acquireSharedInterruptibly 时候传递的是 1 就是说明要获取一个资源，而这里计数器值是资源总个数，也就意味着是让总的资源数减去 1，acquireSharedInterruptibly 内部首先判断如果当前线程被中断了则抛出异常，否者调用 sync 实现的 tryAcquireShared 方法看当前状态值（计数器值）是否为 0，是则当前线程的 await() 方法直接返回，否者调用 AQS 的 doAcquireSharedInterruptibly 让当前线程阻塞。

另外调用 tryAcquireShared 的方法仅仅是检查当前状态值是不是为 0 并没有调用 CAS 让当前状态值减去 1。

- boolean await(long timeout, TimeUnit unit)

当线程调用了 CountDownLatch 对象的该方法后，当前线程会被阻塞，直到下面的情况之一发生才会返回：
（1）当所有线程都调用了 CountDownLatch 对象的 countDown 方法后，也就是计时器值为 0 的时候，这时候返回 true；
(2) 设置的 timeout 时间到了，因为超时而返回 false；
（3）其它线程调用了当前线程的 interrupt（）方法中断了当前线程，当前线程会抛出 InterruptedException 异常后返回。

```
    public boolean await(long timeout, TimeUnit unit)
          throws InterruptedException {
          return sync.tryAcquireSharedNanos(1,
unit.toNanos(timeout));
      }
```

- void countDown()
  当线程调用了该方法后，会递减计数器的值，递减后如果计数器为 0 则会唤醒所有调用 await 方法而被阻塞的线程，否者什么都不做，下面看下 countDown() 方法内部是如何调用 AQS 的方法的：

```
    //CountDownLatch的countDown（）方法
    public void countDown() {
       //委托sync调用AQS的方法
        sync.releaseShared(1);
    }
```

```
    //AQS的方法
     public final boolean releaseShared(int arg) {
         //调用sync实现的tryReleaseShared
         if (tryReleaseShared(arg)) {
             //AQS的释放资源方法
             doReleaseShared();
             return true;
         }
         return false;
     }
```

如上代码可知 CountDownLatch 的 countDown（）方法是委托 sync 调用了 AQS 的 releaseShared 方法，后者调用了 sync 实现的 AQS 的 tryReleaseShared 代码如下：

```
//syn的方法
protected boolean tryReleaseShared(int releases) {
  //循环进行cas，直到当前线程成功完成cas使计数值（状态值state）减一并更新到
state
  for (;;) {
      int c = getState();

      //如果当前状态值为0则直接返回（1）
      if (c == 0)
          return false;

      //CAS设置计数值减一（2）
      int nextc = c-1;
      if (compareAndSetState(c, nextc))
          return nextc == 0;
  }
}
```

如上代码可知首先获取当前状态值（计数器值），代码（1）如果当前状态值为 0 则直接返回 false，则 countDown（）方法直接返回；否者执行代码（2）使用 CAS 设置计数器减一，CAS 失败则循环重试，否者如果当前计数器为 0 则返回 true。

返回 true 后说明当前线程是最后一个调用的 countdown 方法的线程，那么该线程除了让计数器值减一外，还需要唤醒调用 CountDownLatch 的 await 方法而被阻塞的线程。

这里代码（1）貌似是多余的，其实不然，之所以添加代码（1）是为了防止当计数器值为 0 后，其它线程又调用了 countDown 方法，如果没有代码（1），状态值就会变成负数了。

- long getCount()
  获取当前计数器的值，也就是 AQS 的 state 的值，一般在 debug 测试时候使用，下面看下代码：

```
    public long getCount() {
        return sync.getCount();
    }

    int getCount() {
     return getState();
    }
```

如上代码可知内部还是调用了 AQS 的 getState 方法来获取 state 的值（计数器当前值）。

三、CyclicBarrier 原理分析

上面介绍的 CountDownLatch 在解决多个线程同步方面相对于调用线程的 join 已经提供了不少改进，但是 CountDownLatch 的计数器是一次性的，也就是等到计数器变为 0 后，在调用 CountDownLatch 的 await 和 countdown 方法都会立刻返回，这就起不到线程同步的效果了。CyclicBarrier 类的功能不限于 CountDownLatch 所提供的功能，从字面意思理解 CyclicBarrier 是回环屏障的意思，它可以实现让一组线程全部到达一个状态后再全部同时执行。这里之所以叫做回环是因为当所有等待线程执行完毕之后，重置 CyclicBarrier 的状态后可以被重用。

3.1 案例介绍

在介绍原理前先介绍几个使用实例，下面例子我们要实现的是使用两个线程去执行一个被分解的任务 A，当两个线程把自己的任务都执行完毕后在对它们的结果进行汇总处理。

```
public class CycleBarrierTest1 {

    // 创建一个CyclicBarrier实例,添加一个所有子线程全部到达屏障后执行的一个任务
    private static volatile CyclicBarrier cyclicBarrier = new CyclicBarrier(2, new Runnable() {
        public void run() {
            System.out.println(Thread.currentThread() + " task1 merge result");
        }
    });

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

        //创建一个线程个数固定为2的线程池
        ExecutorService executorService = Executors.newFixedThreadPool(2);
```

```
            // 加入线程A到线程池
            executorService.submit(new Runnable() {
                public void run() {
                    try {

                        System.out.println(Thread.currentThread() + "
    task1-1");

                        System.out.println(Thread.currentThread() + "
    enter in barrier");
                        cyclicBarrier.await();
                        System.out.println(Thread.currentThread() + "
    enter out barrier");

                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });

            // 加入线程B到线程池
            executorService.submit(new Runnable() {
                public void run() {
                    try {
                        System.out.println(Thread.currentThread() + "
    task1-2");

                        System.out.println(Thread.currentThread() + "
    enter in barrier");
                        cyclicBarrier.await();
                        System.out.println(Thread.currentThread() + "
    enter out barrier");

                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });

            // 关闭线程池
            executorService.shutdown();
        }
    }
```

输出结果：

```
<terminated> CycleBarrierTest1 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main] task1-1
Thread[pool-1-thread-1,5,main] enter in barrier
Thread[pool-1-thread-2,5,main] task1-2
Thread[pool-1-thread-2,5,main] enter in barrier
Thread[pool-1-thread-2,5,main] task1 merge result
Thread[pool-1-thread-2,5,main] enter out barrier
Thread[pool-1-thread-1,5,main] enter out barrier
```

如上代码创建了一个 CyclicBarrier 对象，第一个参数为计数器初始值，第二个参数 Runable 是指当计数器为 0 时候需要执行的任务。main 函数里面首先创建了固定大小为 2 的线程池，然后添加两个子任务到线程池，每个子任务在执行完自己的逻辑后会调用 await 方法。

一开始计数器为 2，当第一个线程调用 await 方法时候，计数器会递减为 1，由于计数器不为 0，所以当前线程就到了屏障点会被阻塞，然后第二个线程调用 await 时候，会进入屏障，计数器也会递减现在计数器为 0，就会去执行在 CyclicBarrier 构造时候的任务，执行完毕后就会退出屏障点，并且会唤醒被阻塞的第一个线程，这时候第一个线程也会退出屏障点继续向下运行。

上面的例子说明了多个线程之间是相互等待的，假如计数器为 N，那么调用 await 方法的前面 N-1 的线程都会因为到达屏障点被阻塞，当第 N 个线程调用 await 后，计数器为 0 了，这时候第 N 个线程才会发出通知唤醒前面的 N-1 个线程。也就是全部线程达到屏障点时候才能一块继续向下执行，对与这个例子来说使用 CountDownLatch 也可以达到类似输出结果，下面在放个例子来说明 CyclicBarrier 的可复用性。

假设一个任务由阶段 1、阶段 2、阶段 3 组成，每个线程要串行的执行阶段 1 和 2 和 3，多个线程执行该任务时候，必须要保证所有线程的阶段 1 全部完成后才能进行阶段 2 执行，所有线程的阶段 2 全部完成后才能进行阶段 3 执行，下面使用 CyclicBarrier 来完成这个需求。

```
public class CycleBarrierTest2 {

    // 创建一个CyclicBarrier实例
    private static volatile CyclicBarrier cyclicBarrier = new
CyclicBarrier(2);

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

        ExecutorService executorService =
Executors.newFixedThreadPool(2);

        // 加入线程A到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() +
" step1");
```

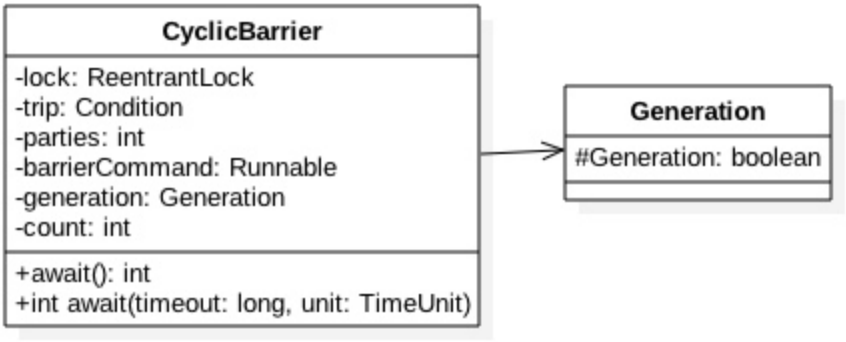
```
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +
" step2");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +
" step3");

                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });

        // 加入线程B到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {
                    System.out.println(Thread.currentThread() +
" step1");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +
" step2");
                    cyclicBarrier.await();

                    System.out.println(Thread.currentThread() +
" step3");

                } catch (Exception e) {
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }
            }
        });

        //关闭线程池
        executorService.shutdown();
    }
}
```

```
<terminated> CycleBarrierTest2 [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main] step1
Thread[pool-1-thread-2,5,main] step1
Thread[pool-1-thread-2,5,main] step2
Thread[pool-1-thread-1,5,main] step2
Thread[pool-1-thread-1,5,main] step3
Thread[pool-1-thread-2,5,main] step3
```

如上代码，在每个子线程执行完 step1 后都调用了 await 方法，所有线程都到达屏障点后才会一块往下执行，这就保证了所有线程完成了 step1 后才会开始执行 step2，然后在 step2 后面调用了 await 方法，这保证了所有线程的 step2 完成后，线程才能开始 step3 的执行，这个功能使用单个 CountDownLatch 是无法完成的。

3.2 实现原理探究

为了能够一览 CyclicBarrier 的架构设计，下面先看下 CyclicBarrier 的类图结构



如上类图可知 CyclicBarrier 内部并不是直接使用 AQS 实现，而是使用了独占锁 ReentrantLock 来实现的同步；parties 用来记录线程个数，用来表示需要多少线程先调用 await 后，所有线程才会冲破屏障继续往下运行；

而 count 一开始等于 parties，每当线程调用 await 方法后就递减 1，当为 0 时候就表示所有线程都到了屏障点，另外你可能疑惑为何维护 parties 和 count 这两个变量，只有 count 不就可以了？

别忘了 cycleBarier 是可以被复用的，使用两个变量原因是用 parties 始终来记录总的线程个数，当 count 计数器变为 0 后，会使用 parties 赋值给 count，已达到复用的作用。这两个变量是在构造 CyclicBarrier 对象时候传递的，如下：

```
public CyclicBarrier(int parties, Runnable barrierAction) {
    if (parties <= 0) throw new IllegalArgumentException();
    this.parties = parties;
    this.count = parties;
    this.barrierCommand = barrierAction;
}
```

这里还有一个变量 barrierCommand 也通过构造函数传递而来，这是一个任务，这个任务的执行时机是当所有线程都都到达屏障点后。另外 CyclicBarrier 内部使用独占锁 Lock 来保证同时只有一个线程调用 await 方法时候才可以返回，使用 lock 首先保证了更新计数器 count 的原子性，另外使用 lock 的条件变量 trip 支持了线程间使用 notify，wait 操作进行同步。

最后变量 generation 内部就一个变量 broken 用来记录当前屏障是否被打破，另外注意这里 broken 并没有被声明为 volatile，是因为锁内使用变量不需要。

```
    private static class Generation {
        boolean broken = false;
    }
```

下面来看下中 CyclicBarrier 几个重要的函数：

- int await()
  当前线程调用 CyclicBarrier 的该方法时候当前线程会被阻塞，直到满足下面条件之一才会返回：
  （1） parties 个线程都调用了 await() 函数，也就是线程都到了屏障点
  (2) 其它线程调用了当前线程的 interrupt（）方法中断了当前线程，则当前线程会抛出 InterruptedException 异常返回
  （3） 当前屏障点关联的 Generation 对象的 broken 标志被设置为 true 时候，会抛出 BrokenBarrierException 异常

如下代码可知内部调用了 dowait 方法，第一个参数 false 说明不设置超时时间，这时候第二个参数没有意义

```
public int await() throws InterruptedException,
BrokenBarrierException {
    try {
        return dowait(false, 0L);
    } catch (TimeoutException toe) {
        throw new Error(toe); // cannot happen
    }
}
```

- boolean await(long timeout, TimeUnit unit)
  当前线程调用 CyclicBarrier 的该方法时候当前线程会被阻塞，直到满足下面条件之一才会返回：
  （1） parties 个线程都调用了 await() 函数，也就是线程都到了屏障点，这时候返回 true
  (2) 当设置的超时时间到了后返回 false
  (3) 其它线程调用了当前线程的 interrupt（）方法中断了当前线程，则当前线程会抛出 InterruptedException 异常返回
  （4） 当前屏障点关联的 Generation 对象的 broken 标志被设置为 true 时候，会抛出 BrokenBarrierException 异常

如下代码可知内部调用了 dowait 方法，第一个参数 true 说明设置了超时时间，这时候第二个参数是超时时间

```
    public int await(long timeout, TimeUnit unit)
        throws InterruptedException,
               BrokenBarrierException,
```

```
                TimeoutException {
        return dowait(true, unit.toNanos(timeout));
    }
```

- int dowait(boolean timed, long nanos)
  该方法是实现 CyclicBarrier 的核心功能，代码如下：

```
private int dowait(boolean timed, long nanos)
        throws InterruptedException, BrokenBarrierException,
               TimeoutException {
        final ReentrantLock lock = this.lock;
        lock.lock();
        try {
             ...

            //(1)如果index==0说明所有线程都到到了屏障点，则执行初始化时候传递的任务
            int index = --count;
            if (index == 0) {  // tripped
                boolean ranAction = false;
                try {
                    //(2)执行任务
                    if (command != null)
                        command.run();
                    ranAction = true;
                    //（3）激活其它调用await而被阻塞的线程，并重置CyclicBarrier
                    nextGeneration();
                    //返回
                    return 0;
                } finally {
                    if (!ranAction)
                        breakBarrier();
                }
            }

            // (4)如果index!=0
            for (;;) {
                try {
                     //（5）没有设置超时时间，
                     if (!timed)
                        trip.await();
                    //（6）设置了超时时间
                    else if (nanos > 0L)
                        nanos = trip.awaitNanos(nanos);
                } catch (InterruptedException ie) {
                    ...
                }
                    ...
            }
```

```
            } finally {
                lock.unlock();
            }
        }

        private void nextGeneration() {
            //（7）唤醒条件队列里面阻塞线程
            trip.signalAll();
            //（8） 重置CyclicBarrier
            count = parties;
            generation = new Generation();
        }
```

上如是 dowait 方法的主干代码，当一个线程调用了 dowait 方法后首先会获取独占锁 lock，如果创建 CycleBarrier 时候传递的参数为 10，那么后面 9 个调用线程会被阻塞；

然后当前获取线程对计数器 count 进行递减操作，递减后的 Count=index=9，因为 index!=0 所以当前线程会执行（4）处代码。如果是无参数的当前线程调用的是无参数的 await 方法，则这里 timed=false，所以当前线程会被放入条件变量 trip 的阻塞队列，当前线程会被挂起并释放获取的 Lock 锁；

如果是调用的有参数的 await 方法则 timed=true，则当前线程线程也会被放入条件变量阻塞队列并释放锁资源，但是不同的是当前线程会在指定时间超时后自动被激活。

当第一个获取锁的线程由于被阻塞释放锁后，被阻塞的 9 个线程中有一个会竞争到 lock 锁，然后执行第一个线程同样的操作，直到最后一个线程获取到 lock 时候已经有 9 个线程被放入了 Lock 的条件队列里面，最后这一个线程 count 递减后 Count=index 等于 0，所以执行（2）处代码，如果创建 CyclicBarrier 时候传递了任务，则在其它线程被唤醒前先执行任务，任务执行完毕后在执行（3）处代码，唤醒其它 9 个线程，并重置 CyclicBarrier，然后这 10 个线程就可以继续向下运行了。

四、 信号量 Semaphore 原理探究

Semaphore 信号量也是 Java 中一个同步器，与 CountDownLatch 和 CycleBarrier 不同在于它内部的计数器是递增的。

4.1 案例介绍

类似于 CountDownLatch，下面我们的例子也是在主线程中开启两个子线程进行执行，等所有子线程执行完毕后主线程在继续向下运行。

```
public class SemaphoreTest {

    // 创建一个Semaphore实例
```

```
    private static volatile Semaphore semaphore = new
Semaphore(0);

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

        ExecutorService executorService =
Executors.newFixedThreadPool(2);

        // 加入线程A到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() +
" over");
                    semaphore.release();

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

        // 加入线程B到线程池
        executorService.submit(new Runnable() {
            public void run() {
                try {

                    System.out.println(Thread.currentThread() +
" over");
                    semaphore.release();

                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        });

        // 等待子线程执行完毕，返回
        semaphore.acquire(2);
        System.out.println("all child thread over!");

        //关闭线程池
        executorService.shutdown();
    }
}
```

输出结果:

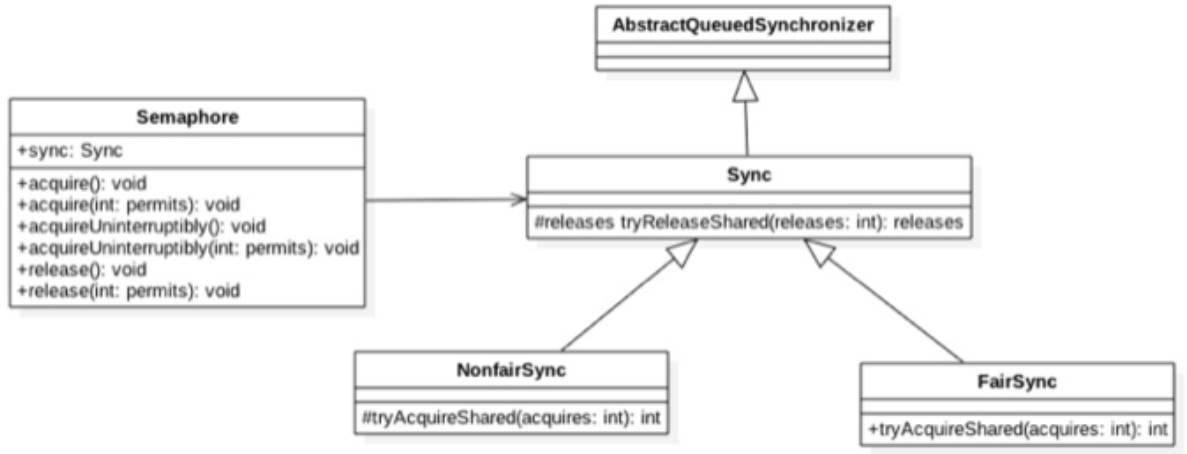
```
<terminated> SemaphoreTest [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_101.jdk/Contents/Home/bin/java
Thread[pool-1-thread-1,5,main] over
Thread[pool-1-thread-2,5,main] over
all child thread over!
```

如上代码首先首先创建了一个信号量实例，构造函数的入参为 0，说明当前信号量计数器为 0，然后 main 函数添加两个线程任务到线程池，每个线程内部调用了信号量的 release 方法，相当于计数值递增一，最后在 main 线程里面调用信号量的 acquire 方法，参数传递为 2 说明调用 acquire 方法的线程会一直阻塞，直到信号量的计数变为 2 时才会返回。

看到这里也就明白了，如果构造 Semaphore 时候传递的参数为 N，在 M 个线程中调用了该信号量的 release 方法，那么在调用 acquire 对 M 个线程进行同步时候传递的参数应该是 M+N;

4.2 实现原理探究

为了能够一览 Semaphore 的内部结构，首先看下 Semaphore 的类图：



如上类图可知 Semaphore 内部还是使用 AQS 来实现，Sync 只是对 AQS 的一个修饰，并且 sync 有俩个实现类分别代表获取信号量时候是否采用公平策略。如下代码创建 Semaphore 时候会有一个变量标示是否使用公平策略：

```
public Semaphore(int permits) {
    sync = new NonfairSync(permits);
}

public Semaphore(int permits, boolean fair) {
    sync = fair ? new FairSync(permits) : new
    NonfairSync(permits);
}

Sync(int permits) {
    setState(permits);
}
```

如上代码 Semaphore 默认采用的非公平策略，如果你需要公平策略则可以使用带两个参数的构造函数来构造 Semaphore 对象，另外和 CountDownLatch 一构造函数里面传递的初始化信号量个数 permits 被赋值给了 AQS 的 state 状态变量，也就是这里 AQS 的 state 值表示当前持有的信号量个数。

下面来看下 Semaphore 实现的主要函数：

- void acquire()
  当前线程调用该方法时候目的是希望获取一个信号量资源，如果当前信号量计数个数大于 0，并且当前线程获取到了一个信号量则该方法直接返回，当前信号量的计数会减少 1. 否者会被放入 AQS 的阻塞队列，当前线程被挂起，直到其它线程调用了 release 方法释放了信号量，并且当前线程通过竞争获取到了该信号量。当当前线程被其它线程调用了 interrupt（）方法中断后，当前线程会抛出 InterruptedException 异常然后返回。下面看下代码实现：

```
public void acquire() throws InterruptedException {
    //传递参数为1，说明要获取1个信号量资源
    sync.acquireSharedInterruptibly(1);
}
```

```
public final void acquireSharedInterruptibly(int arg)
        throws InterruptedException {

    //（1）如果线程被中断，则抛出中断异常
    if (Thread.interrupted())
        throw new InterruptedException();

    //（2）否者调用sync子类方法尝试获取,这里根据构造函数确定使用公平策略
    if (tryAcquireShared(arg) < 0)
        //如果获取失败则放入阻塞队列,然后再次尝试如果失败则调用park方法挂起当前线程
        doAcquireSharedInterruptibly(arg);
}
```

如上代码可知 acquire() 内部调用了 sync 的 acquireSharedInterruptibly 方法，后者是对中断响应的（如果当前线程被中断，则抛出中断异常），尝试获取信号量资源的 AQS 的方法 tryAcquireShared 是由 sync 的子类实现，所以这里就要分公平性了，这里讨论非公平策略 NonfairSync 类的 tryAcquireShared 方法，代码如下：

```
protected int tryAcquireShared(int acquires) {
    return nonfairTryAcquireShared(acquires);
}
```

```
final int nonfairTryAcquireShared(int acquires) {
    for (;;) {
     //获取当前信号量值
     int available = getState();
     //计算当前剩余值
     int remaining = available - acquires;
     //如果当前剩余小于0或者CAS设置成功则返回
     if (remaining < 0 ||
         compareAndSetState(available, remaining))
         return remaining;
    }
}
```

如上代码先计算当前信号量值（available）减去需要获取的值（acquires）得到剩余的信号量个数（remaining），如果剩余值小于 0 说明当前信号量个数满足不了需求则直接返回负数，然后当前线程会被放入 AQS 的阻塞队列，当前线程被挂起。

如果剩余值大于 0 则使用 CAS 操作设置当前信号量值为剩余值，然后返回剩余值。另外可知 NonFairSync 是非公平性获取的，是说先调用 aquire 方法获取信号量的线程不一定比后来者先获取到。

下面看下公平性的 FairSync 类是如何保证公平性的：

```
        protected int tryAcquireShared(int acquires) {
            for (;;) {
                if (hasQueuedPredecessors())
                    return -1;
                int available = getState();
                int remaining = available - acquires;
                if (remaining < 0 ||
                    compareAndSetState(available, remaining))
                    return remaining;
            }
        }
```

可知公平新还是靠 hasQueuedPredecessors 这个函数来做的，前面锁章节已经讲过公平性是看当前线程节点的是否有前驱节点也在等待获取该资源，如果是则自己放弃获取的权利，然后当前线程会被放入 AQS 阻塞队列，否者就去获取。

```
    public final boolean hasQueuedPredecessors() {
        Node t = tail;
        Node h = head;
        Node s;
        return h != t &&
            ((s = h.next) == null || s.thread !=
Thread.currentThread());
    }
```

如上代码如果当前线程节点有前驱节点则返回 true，否者如果当前 AQS 队列为空或者当前线程节点是 AQS 的第一个节点则返回 false. 其中如果 h==t 则说明当前队列为空则直接返回 false，如果 h!=t 并且 s==null 说明有一个元素将要作为 AQS 的第一个节点入队列 (回顾下 enq 函数第一个元素入队列是两步操作，首先创建一个哨兵头节点，然后第一个元素插入到哨兵节点后面)，那么返回 true，如果 h!=t 并且 s!=null 并且 s.thread != Thread.currentThread() 则说明队列里面的第一个元素不是当前线程则返回 true.

- void acquire(int permits)
  该方法与 acquire() 不同在与后者只需要获取一个信号量值，而前者则获取指定 permits 个。

```
    public void acquire(int permits) throws InterruptedException
{
        if (permits < 0) throw new IllegalArgumentException();
        sync.acquireSharedInterruptibly(permits);
    }
```

- void acquireUninterruptibly()
  该方法与 acquire() 类似，不同之处在于该方法对中断不响应，也就是当当前线程调用了 acquireUninterruptibly 获取资源过程中（包含被阻塞后）其它线程调用了当前线程的 interrupt（）方法设置了当前线程的中断标志当前线程并不会抛出 InterruptedException 异常而返回。

```
    public void acquireUninterruptibly() {
        sync.acquireShared(1);
    }
```

- void acquireUninterruptibly(int permits)
  该方法与 acquire(int permits) 不同在于该方法对中断不响应。

```
    public void acquireUninterruptibly(int permits) {
        if (permits < 0) throw new IllegalArgumentException();
        sync.acquireShared(permits);
    }
```

- void release()
  该方法作用的把当前 semaphore 对象的信号量值增加 1，如果当前有线程因为调用 aquire 方法被阻塞放入了 AQS 的阻塞队列，则会根据公平策略选择一个线程进行激活，激活的线程会尝试获取刚增加的信号量，下面看下代码实现：

```
    public void release() {
        //(1)arg=1
```

```
        sync.releaseShared(1);
    }

    public final boolean releaseShared(int arg) {

        //(2)尝试释放资源
        if (tryReleaseShared(arg)) {

            //(3)资源释放成功则调用park唤醒AQS队列里面最先挂起的线程
            doReleaseShared();
            return true;
        }
        return false;
    }

    protected final boolean tryReleaseShared(int releases) {
        for (;;) {

            //(4)获取当前信号量值
            int current = getState();

            //(5)当前信号量值增加releases，这里为增加1
            int next = current + releases;
            if (next < current) // 移除处理
                throw new Error("Maximum permit count exceeded");

            //(6)使用cas保证更新信号量值的原子性
            if (compareAndSetState(current, next))
                return true;
        }
    }
```

如上代码 release()->sync.releaseShared(1)，可知 release 方法每次只会对信号量值增加 1，tryReleaseShared 方法是无限循环，使用 CAS 保证了 release 函数对信号量递增 1 的原子性操作。当 tryReleaseShared 函数增加信号量成功后会执行（3）处的代码，调用 AQS 的方法来激活因为调用 aquire 方法而被阻塞的线程。

- void release(int permits)
  该方法与不带参数的不同之处在于前者每次调用会在信号量值原来基础上增加 permits，而后者每次增加 1.

```
    public void release(int permits) {
        if (permits < 0) throw new IllegalArgumentException();
        sync.releaseShared(permits);
    }
```

另外注意到这里调用的是 sync.releaseShared 是共享方法，这说明该信号量是线程共享的，信号量没有和固定线程绑定，多个线程可以同时使用 CAS 去更新信号量的值而不会

被阻塞。

五、 总结

本节介绍了并发包中线程协作的一些重要类，首先 CountDownLatch 通过计数器提供了更灵活的控制，只要检测到计数器为 0，而不管当前线程是否结束调用 await 的线程就可以往下执行，相比使用 jion 必须等待线程执行完毕后主线程才会继续向下运行更灵活。

CyclicBarrier 也可以达到 CountDownLatch 的效果，但是后者当计数器变为 0 后，就不能在被复用，而前者则使用 reset 方法可以重置后复用，前者对同一个算法但是输入参数不同的类似场景下比较适用。

而 semaphore 采用了信号量递增的策略，一开始并不需要关心需要同步的线程个数，等调用 aquire 时候在指定需要同步个数，并且提供了获取信号量的公平性策略。

掌握本章的知识后，会大大减少你在 Java 中使用 wait,notify 等复杂的并发控制代码，在日常开发中当需要进行线程同步时候使用这些同步类会节省很多代码量并且可以保证正确性。