## Java 并发包中的高级同步工具 + 面试题

Java 中的并发包指的是 java.util.concurrent（简称 JUC）包和其子包下的类和接口，它为 Java 的并发提供了各种功能支持，比如：

* 提供了线程池的创建类 ThreadPoolExecutor、Executors 等；
* 提供了各种锁，如 Lock、ReentrantLock 等；
* 提供了各种线程安全的数据结构，如 ConcurrentHashMap、LinkedBlockingQueue、DelayQueue 等；
* 提供了更加高级的线程同步结构，如 CountDownLatch、CyclicBarrier、Semaphore 等。

在前面的章节中我们已经详细地介绍了线程池的使用、线程安全的数据结构等，本文我们就重点学习一下 Java 并发包中更高级的线程同步类：CountDownLatch、CyclicBarrier、Semaphore 和 Phaser 等。

### CountDownLatch 介绍和使用

CountDownLatch（闭锁）可以看作一个只能做减法的计数器，可以让一个或多个线程等待执行。  
CountDownLatch 有两个重要的方法：

* countDown()：使计数器减 1；
* await()：当计数器不为 0 时，则调用该方法的线程阻塞，当计数器为 0 时，可以唤醒等待的一个或者全部线程。

CountDownLatch 使用场景：  
以生活中的情景为例，比如去医院体检，通常人们会提前去医院排队，但只有等到医生开始上班，才能正式开始体检，医生也要给所有人体检完才能下班，这种情况就要使用 CountDownLatch，流程为：患者排队 → 医生上班 → 体检完成 → 医生下班。

CountDownLatch 示例代码如下：

// 医院闭锁  
CountDownLatch hospitalLatch = new CountDownLatch(1);  
// 患者闭锁  
CountDownLatch patientLatch = new CountDownLatch(5);  
System.out.println("患者排队");  
ExecutorService executorService = Executors.newCachedThreadPool();  
for (int i = 0; i < 5; i++) {  
 final int j = i;  
 executorService.execute(() -> {  
 try {  
 hospitalLatch.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.out.println("体检：" + j);  
 patientLatch.countDown();  
 });  
}  
System.out.println("医生上班");  
hospitalLatch.countDown();  
patientLatch.await();  
System.out.println("医生下班");  
executorService.shutdown();

以上程序执行结果如下：

患者排队

医生上班

体检：4

体检：0

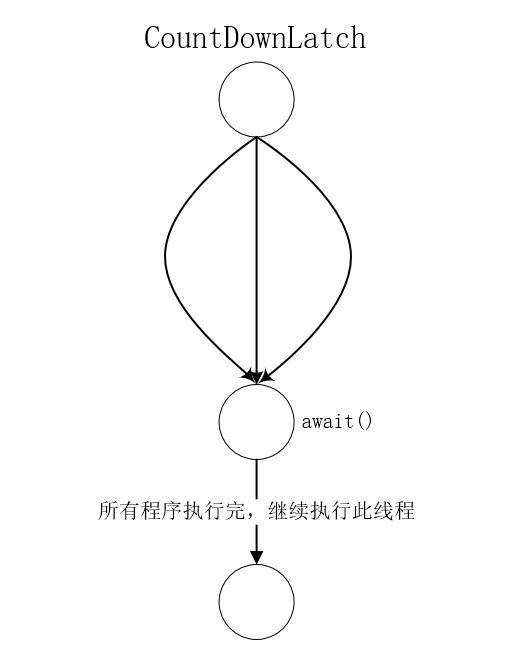
体检：1

体检：3

体检：2

医生下班

执行流程如下图：



### CyclicBarrier 介绍和使用

CyclicBarrier（循环屏障）通过它可以实现让一组线程等待满足某个条件后同时执行。

CyclicBarrier 经典使用场景是公交发车，为了简化理解我们这里定义，每辆公交车只要上满 4 个人就发车，后面来的人都会排队依次遵循相应的标准。

它的构造方法为 CyclicBarrier(int parties,Runnable barrierAction) 其中，parties 表示有几个线程来参与等待，barrierAction 表示满足条件之后触发的方法。CyclicBarrier 使用 await() 方法来标识当前线程已到达屏障点，然后被阻塞。

CyclicBarrier 示例代码如下：

import java.util.concurrent.\*;  
public class CyclicBarrierTest {  
 public static void main(String[] args) throws InterruptedException {  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(4, new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("发车了");  
 }  
 });  
 for (int i = 0; i < 4; i++) {  
 new Thread(new CyclicWorker(cyclicBarrier)).start();  
 }  
 }  
 static class CyclicWorker implements Runnable {  
 private CyclicBarrier cyclicBarrier;  
 CyclicWorker(CyclicBarrier cyclicBarrier) {  
 this.cyclicBarrier = cyclicBarrier;  
 }  
 @Override  
 public void run() {  
 for (int i = 0; i < 2; i++) {  
 System.out.println("乘客：" + i);  
 try {  
 cyclicBarrier.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

以上程序执行结果如下：

乘客：0

乘客：0

乘客：0

乘客：0

发车了

乘客：1

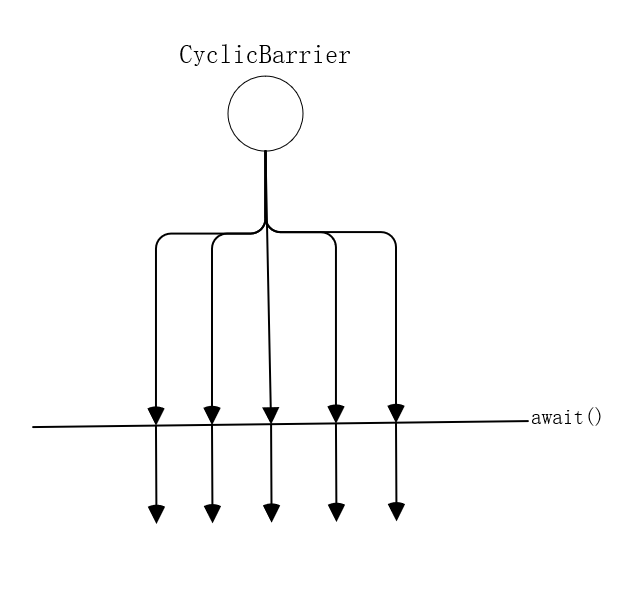
乘客：1

乘客：1

乘客：1

发车了

执行流程如下图：



### Semaphore 介绍和使用

Semaphore（信号量）用于管理多线程中控制资源的访问与使用。Semaphore 就好比停车场的门卫，可以控制车位的使用资源。比如来了 5 辆车，只有 2 个车位，门卫可以先放两辆车进去，等有车出来之后，再让后面的车进入。

Semaphore 示例代码如下：

Semaphore semaphore = new Semaphore(2);  
ThreadPoolExecutor semaphoreThread = new ThreadPoolExecutor(10, 50, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<>());  
for (int i = 0; i < 5; i++) {  
 semaphoreThread.execute(() -> {  
 try {  
 // 堵塞获取许可  
 semaphore.acquire();  
 System.out.println("Thread：" + Thread.currentThread().getName() + " 时间：" + LocalDateTime.now());  
 TimeUnit.SECONDS.sleep(2);  
 // 释放许可  
 semaphore.release();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
}

以上程序执行结果如下：

Thread：pool-1-thread-1 时间：2019-07-10 21:18:42

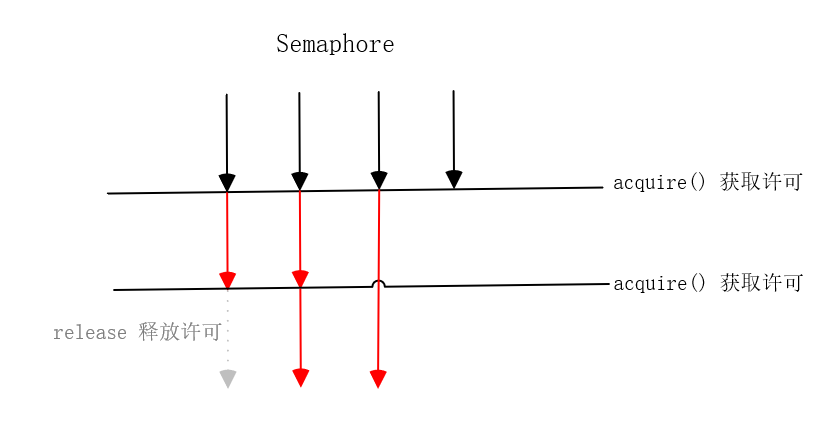
Thread：pool-1-thread-2 时间：2019-07-10 21:18:42

Thread：pool-1-thread-3 时间：2019-07-10 21:18:44

Thread：pool-1-thread-4 时间：2019-07-10 21:18:44

Thread：pool-1-thread-5 时间：2019-07-10 21:18:46

执行流程如下图：



enter image description here

### Phaser 介绍和使用

Phaser（移相器）是 JDK 7 提供的，它的功能是等待所有线程到达之后，才继续或者开始进行新的一组任务。

比如有一个旅行团，我们规定所有成员必须都到达指定地点之后，才能发车去往景点一，到达景点之后可以各自游玩，之后必须全部到达指定地点之后，才能继续发车去往下一个景点，类似这种场景就非常适合使用 Phaser。

Phaser 示例代码如下：

public class Lesson5\_6 {  
 public static void main(String[] args) throws InterruptedException {  
 Phaser phaser = new MyPhaser();  
 PhaserWorker[] phaserWorkers = new PhaserWorker[5];  
 for (int i = 0; i < phaserWorkers.length; i++) {  
 phaserWorkers[i] = new PhaserWorker(phaser);  
 // 注册 Phaser 等待的线程数，执行一次等待线程数 +1  
 phaser.register();  
 }  
 for (int i = 0; i < phaserWorkers.length; i++) {  
 // 执行任务  
 new Thread(new PhaserWorker(phaser)).start();  
 }  
 }  
 static class PhaserWorker implements Runnable {  
 private final Phaser phaser;  
 public PhaserWorker(Phaser phaser) {  
 this.phaser = phaser;  
 }  
 @Override  
 public void run() {  
 System.out.println(Thread.currentThread().getName() + " | 到达" );  
 phaser.arriveAndAwaitAdvance(); // 集合完毕发车  
 try {  
 Thread.sleep(new Random().nextInt(5) \* 1000);  
 System.out.println(Thread.currentThread().getName() + " | 到达" );  
 phaser.arriveAndAwaitAdvance(); // 景点 1 集合完毕发车  
 Thread.sleep(new Random().nextInt(5) \* 1000);  
 System.out.println(Thread.currentThread().getName() + " | 到达" );  
 phaser.arriveAndAwaitAdvance(); // 景点 2 集合完毕发车  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 // Phaser 每个阶段完成之后的事件通知  
 static class MyPhaser extends Phaser{  
 @Override  
 protected boolean onAdvance(int phase, int registeredParties) { // 每个阶段执行完之后的回调  
 switch (phase) {  
 case 0:  
 System.out.println("==== 集合完毕发车 ====");  
 return false;  
 case 1:  
 System.out.println("==== 景点1集合完毕，发车去下一个景点 ====");  
 return false;  
 case 2:  
 System.out.println("==== 景点2集合完毕，发车回家 ====");  
 return false;  
 default:  
 return true;  
 }  
 }  
 }  
}

以上程序执行结果如下：

Thread-0 | 到达

Thread-4 | 到达

Thread-3 | 到达

Thread-1 | 到达

Thread-2 | 到达

==== 集合完毕发车 ====

Thread-0 | 到达

Thread-4 | 到达

Thread-1 | 到达

Thread-3 | 到达

Thread-2 | 到达

==== 景点1集合完毕，发车去下一个景点 ====

Thread-4 | 到达

Thread-3 | 到达

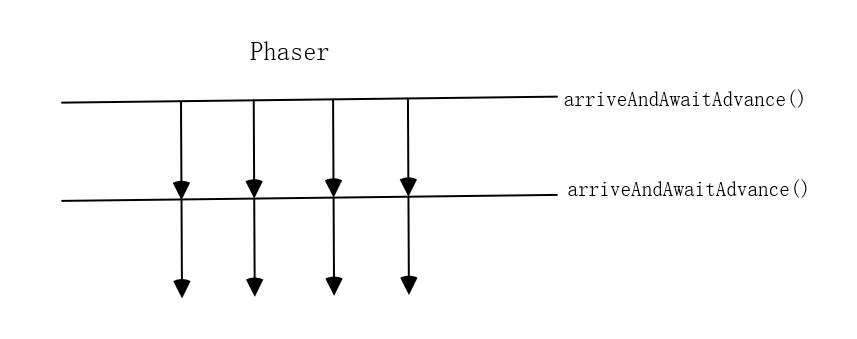
Thread-2 | 到达

Thread-1 | 到达

Thread-0 | 到达

==== 景点2集合完毕，发车回家 ====

执行流程如下图：



enter image description here

### 相关面试题

#### 1.以下哪个类用于控制某组资源的访问权限？

A：Phaser  
B：Semaphore  
C：CountDownLatch  
D：CyclicBarrier

答：B

#### 2.以下哪个类不能被重用？

A：Phaser  
B：Semaphore  
C：CountDownLatch  
D：CyclicBarrier

答：C

#### 3.以下哪个方法不属于 CountDownLatch 类？

A：await()  
B：countDown()  
C：getCount()  
D：release()

答：D  
题目解析：release() 是 Semaphore 的释放许可的方法，CountDownLatch 类并不包含此方法。

#### 4.CyclicBarrier 与 CountDownLatch 有什么区别？

答：CyclicBarrier 与 CountDownLatch 本质上都是依赖 volatile 和 CAS 实现的，它们区别如下：

* CountDownLatch 只能使用一次，而 CyclicBarrier 可以使用多次。
* CountDownLatch 是手动指定等待一个或多个线程执行完成再执行，而 CyclicBarrier 是 n 个线程相互等待，任何一个线程完成之前，所有的线程都必须等待。

#### 5.以下哪个类不包含 await() 方法？

A：Semaphore  
B：CountDownLatch  
C：CyclicBarrier

答：A

#### 6.以下程序执行花费了多长时间？

Semaphore semaphore = new Semaphore(2);  
ThreadPoolExecutor semaphoreThread = new ThreadPoolExecutor(10, 50, 60, TimeUnit.SECONDS, new LinkedBlockingQueue<>());  
for (int i = 0; i < 3; i++) {  
 semaphoreThread.execute(() -> {  
 try {  
 semaphore.release();  
 System.out.println("Hello");  
 TimeUnit.SECONDS.sleep(2);  
 semaphore.acquire();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
}

A：1s 以内  
B：2s 以上

答：A  
题目解析：循环先执行了 release() 也就是释放许可的方法，因此程序可以一次性执行 3 个线程，同时会在 1s 以内执行完。

#### 7.Semaphore 有哪些常用的方法？

答：常用方法如下：

* acquire()：获取一个许可。
* release()：释放一个许可。
* availablePermits()：当前可用的许可数。
* acquire(int n)：获取并使用 n 个许可。
* release(int n)：释放 n 个许可。

#### 8.Phaser 常用方法有哪些？

答：常用方法如下：

* register()：注册新的参与者到 Phaser
* arriveAndAwaitAdvance()：等待其他线程执行
* arriveAndDeregister()：注销此线程
* forceTermination()：强制 Phaser 进入终止态
* isTerminated()：判断 Phaser 是否终止

#### 9.以下程序是否可以正常执行？“发车了”打印了多少次？

import java.util.concurrent.\*;  
public class TestMain {  
 public static void main(String[] args) {  
 CyclicBarrier cyclicBarrier = new CyclicBarrier(4, new Runnable() {  
 @Override  
 public void run() {  
 System.out.println("发车了");  
 }  
 });  
 for (int i = 0; i < 4; i++) {  
 new Thread(new CyclicWorker(cyclicBarrier)).start();  
 }  
 }  
 static class CyclicWorker implements Runnable {  
 private CyclicBarrier cyclicBarrier;  
  
 CyclicWorker(CyclicBarrier cyclicBarrier) {  
 this.cyclicBarrier = cyclicBarrier;  
 }  
 @Override  
 public void run() {  
 for (int i = 0; i < 2; i++) {  
 System.out.println("乘客：" + i);  
 try {  
 cyclicBarrier.await();  
 System.out.println("乘客 II：" + i);  
 cyclicBarrier.await();  
 } catch (InterruptedException e) {  
 e.printStackTrace();  
 } catch (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

答：可以正常执行，因为执行了两次 await()，所以“发车了”打印了 4 次。

### 总结

本文我们介绍了四种比 synchronized 更高级的线程同步类，其中 CountDownLatch、CyclicBarrier、Phaser 功能比较类似都是实现线程间的等待，只是它们的侧重点有所不同，其中 CountDownLatch 一般用于等待一个或多个线程执行完，才执行当前线程，并且 CountDownLatch 不能重复使用；CyclicBarrier 用于等待一组线程资源都进入屏障点再共同执行；Phaser 是 JDK 7 提供的功能更加强大和更加灵活的线程辅助工具，等待所有线程达到之后，继续或开始新的一组任务，Phaser 提供了动态增加和消除线程同步个数功能。而 Semaphore 提供的功能更像锁，用于控制一组资源的访问权限。

[点击此处下载本文源码](https://github.com/vipstone/java-%20interview/tree/master/interview-code/src/main/java/com/interview)

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持