

主讲老师：Fox

有道笔记地址：<https://note.youdao.com/s/UODIYjGS>

跳出来，看全局：<https://www.processon.com/view/link/615d4a610e3e74663e97fa0e>
钻进去，看本质

思考：并发编程是为了解决什么问题的？

性能+线程安全（正确性）

分工 同步 互斥

常用并发同步工具类的真实应用场景

jdk提供了比synchronized更加高级的各种同步工具，包括ReentrantLock、Semaphore、CountDownLatch、CyclicBarrier等，可以实现更加丰富的多线程操作。



1. ReentrantLock

ReentrantLock是一种可重入的独占锁，它允许同一个线程多次获取同一个锁而不会被阻塞。它的功能类似于synchronized是一种互斥锁，可以保证线程安全。相对于 synchronized，ReentrantLock具备如下特点：

- 可中断
- 可以设置超时时间
- 可以设置为公平锁
- 支持多个条件变量
- 与 synchronized 一样，都支持可重入

它的主要应用场景是在多线程环境下对共享资源进行独占式访问，以保证数据的一致性和安全性。

1.1 常用API

Lock接口

ReentrantLock实现了Lock接口规范，常见API如下：

void lock()	获取锁，调用该方法当前线程会获取锁，当锁获得后，该方法返回
void lockInterruptibly() throws InterruptedException	可中断的获取锁，和lock()方法不同之处在于该方法会响应中断，即在锁的获取中可以中断当前线程
boolean tryLock()	尝试非阻塞的获取锁，调用该方法后立即返回。如果能够获取到返回true，否则返回false
boolean tryLock(long time, TimeUnit unit) throws InterruptedException	超时获取锁，当前线程在以下三种情况下会被返回： 当前线程在超时时间内获取了锁 当前线程在超时时间内被中断 超时时间结束，返回false
void unlock()	释放锁
Condition newCondition()	获取等待通知组件，该组件和当前的锁绑定，当前线程只有获取了锁，才能调用该组件的await()方法，而调用后，当前线程将释放锁

基本语法

```
1 //加锁 阻塞
2 lock.lock();
```

```

3  try {
4      ...
5  } finally {
6      // 解锁
7      lock.unlock();
8  }
9
10
11 //尝试加锁    非阻塞
12 if (lock.tryLock(1, TimeUnit.SECONDS)) {
13     try {
14         ...
15     } finally {
16         lock.unlock();
17     }
18 }

```

在使用时要注意 4 个问题：

1. 默认情况下 ReentrantLock 为非公平锁而非公平锁;
2. 加锁次数和释放锁次数一定要保持一致，否则会导致线程阻塞或程序异常;
3. 加锁操作一定要放在 try 代码之前，这样可以避免未加锁成功又释放锁的异常;
4. 释放锁一定要放在 finally 中，否则会导致线程阻塞。

1.2 ReentrantLock使用

独占锁：模拟抢票场景

8张票，10个人抢，如果不加锁，会出现什么问题？

```

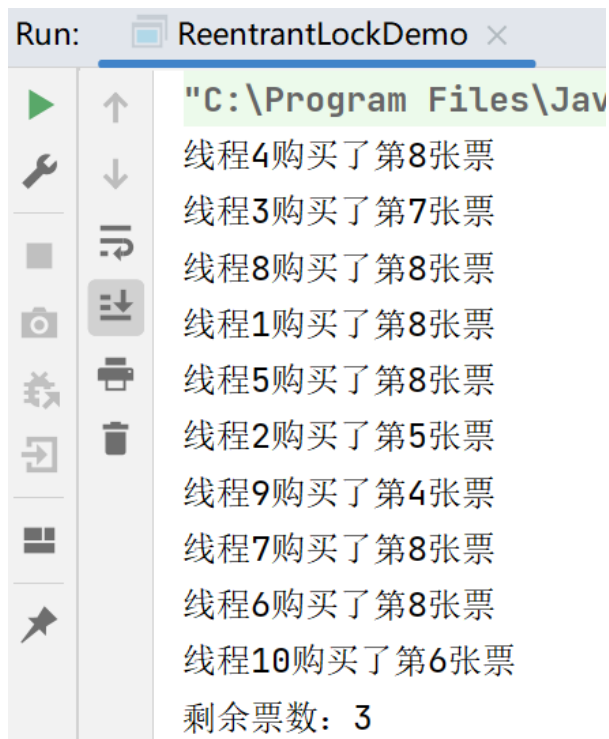
1  /**
2   * 模拟抢票场景
3   */
4  public class ReentrantLockDemo {
5
6      private final ReentrantLock lock = new ReentrantLock();//默认非公平
7      private static int tickets = 8; // 总票数
8
9      public void buyTicket() {
10         lock.lock(); // 获取锁

```

```
11     try {
12         if (tickets > 0) { // 还有票
13             try {
14                 Thread.sleep(10); // 休眠10ms,模拟出并发效果
15             } catch (InterruptedException e) {
16                 e.printStackTrace();
17             }
18             System.out.println(Thread.currentThread().getName() + "购买了第" +
tickets-- + "张票");
19         } else {
20             System.out.println("票已经卖完了, " + Thread.currentThread().getName() +
"抢票失败");
21         }
22     } finally {
23         lock.unlock(); // 释放锁
24     }
25 }
26
27
28
29 public static void main(String[] args) {
30     ReentrantLockDemo ticketSystem = new ReentrantLockDemo();
31     for (int i = 1; i <= 10; i++) {
32         Thread thread = new Thread(() -> {
33
34             ticketSystem.buyTicket(); // 抢票
35
36             }, "线程" + i);
37         // 启动线程
38         thread.start();
39     }
40 }
41
42
43 try {
44     Thread.sleep(3000);
45 } catch (InterruptedException e) {
46     throw new RuntimeException(e);
47 }
48 System.out.println("剩余票数: " + tickets);
```

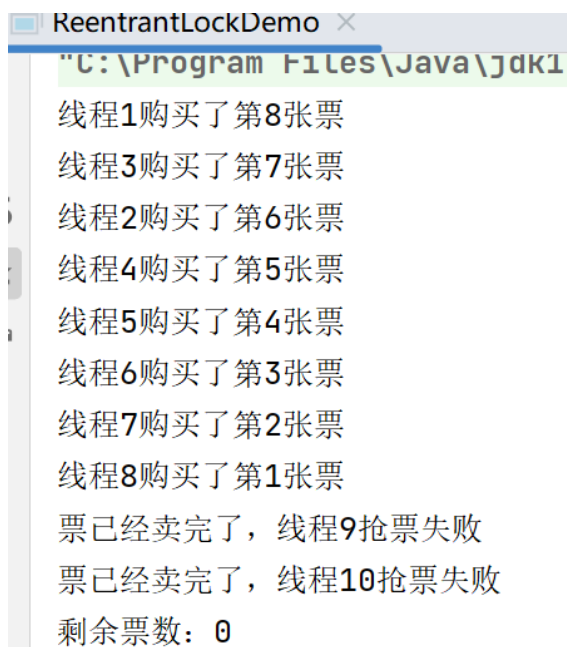
```
49     }  
50 }
```

不加锁的效果： 出现超卖的问题



```
Run: ReentrantLockDemo x  
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe"  
线程4购买了第8张票  
线程3购买了第7张票  
线程8购买了第8张票  
线程1购买了第8张票  
线程5购买了第8张票  
线程2购买了第5张票  
线程9购买了第4张票  
线程7购买了第8张票  
线程6购买了第8张票  
线程10购买了第6张票  
剩余票数: 3
```

加锁效果： 正常，两个人抢票失败



```
Run: ReentrantLockDemo x  
"C:\Program Files\Java\jdk1.8.0_101\bin\java.exe"  
线程1购买了第8张票  
线程3购买了第7张票  
线程2购买了第6张票  
线程4购买了第5张票  
线程5购买了第4张票  
线程6购买了第3张票  
线程7购买了第2张票  
线程8购买了第1张票  
票已经卖完了，线程9抢票失败  
票已经卖完了，线程10抢票失败  
剩余票数: 0
```

公平锁和非公平锁

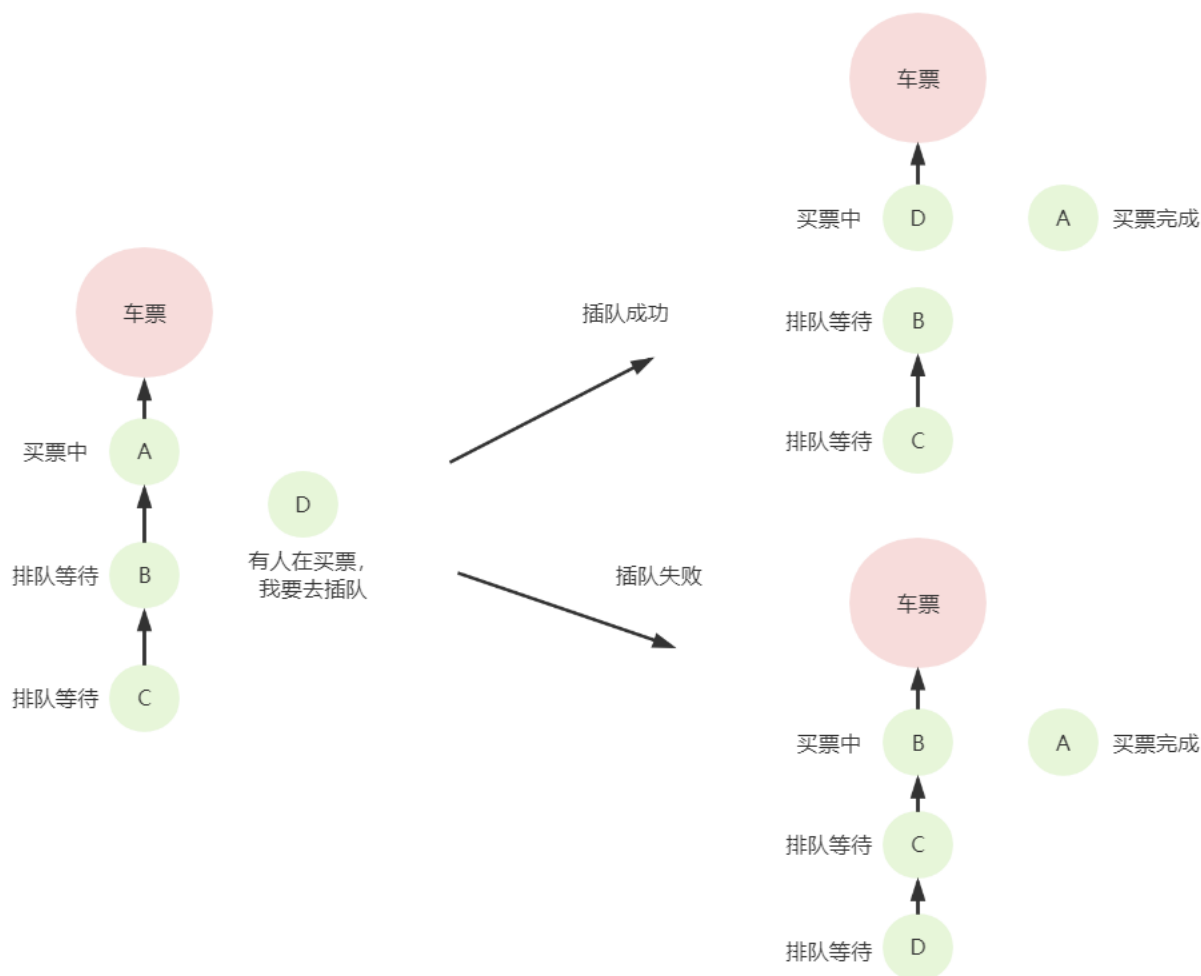
ReentrantLock支持公平锁和非公平锁两种模式：

- 公平锁：线程在获取锁时，按照等待的先后顺序获取锁。
- 非公平锁：线程在获取锁时，不按照等待的先后顺序获取锁，而是随机获取锁。ReentrantLock默认是非公平锁

```
1 ReentrantLock lock = new ReentrantLock(); //参数默认false，不公平锁
```

```
2 ReentrantLock lock = new ReentrantLock(true); //公平锁
```

比如买票的时候就有可能出现插队的场景，允许插队就是非公平锁，如下图：



可重入锁

可重入锁又名递归锁，是指在同一个线程在外层方法获取锁的时候，再进入该线程的内层方法会自动获取锁（前提锁对象得是同一个对象），不会因为之前已经获取过还没释放而阻塞。Java中`ReentrantLock`和`synchronized`都是可重入锁，可重入锁的一个优点是可一定程度避免死锁。在实际开发中，可重入锁常常应用于递归操作、调用同一个类中的其他方法、锁嵌套等场景中。

```
1 class Counter {
2     private final ReentrantLock lock = new ReentrantLock(); // 创建 ReentrantLock 对象
3
4     public void recursiveCall(int num) {
5         lock.lock(); // 获取锁
6         try {
7             if (num == 0) {
8                 return;
9             }
10            System.out.println("执行递归, num = " + num);
11        } finally {
12            lock.unlock();
13        }
14    }
15 }
```

```

11         recursiveCall(num - 1);
12     } finally {
13         lock.unlock(); // 释放锁
14     }
15 }
16
17 public static void main(String[] args) throws InterruptedException {
18     Counter counter = new Counter(); // 创建计数器对象
19
20     // 测试递归调用
21     counter.recursiveCall(10);
22 }
23 }

```

结合Condition实现生产者消费者模式

java.util.concurrent类库中提供Condition类来实现线程之间的协调。调用Condition.await() 方法使线程等待，其他线程调用Condition.signal() 或 Condition.signalAll() 方法唤醒等待的线程。

注意：调用Condition的await()和signal()方法，都必须在lock保护之内。

案例：基于ReentrantLock和Condition实现一个简单队列

```

1  public class ReentrantLockDemo3 {
2
3      public static void main(String[] args) {
4          // 创建队列
5          Queue queue = new Queue(5);
6          //启动生产者线程
7          new Thread(new Producer(queue)).start();
8          //启动消费者线程
9          new Thread(new Customer(queue)).start();
10
11     }
12 }
13
14 /**
15  * 队列封装类
16  */
17 class Queue {

```

```
18     private Object[] items ;
19     int size = 0;
20     int takeIndex;
21     int putIndex;
22     private ReentrantLock lock;
23     public Condition notEmpty; //消费者线程阻塞唤醒条件，队列为空阻塞，生产者生产完唤醒
24     public Condition notFull; //生产者线程阻塞唤醒条件，队列满了阻塞，消费者消费完唤醒
25
26     public Queue(int capacity){
27         this.items = new Object[capacity];
28         lock = new ReentrantLock();
29         notEmpty = lock.newCondition();
30         notFull = lock.newCondition();
31     }
32
33
34     public void put(Object value) throws Exception {
35         //加锁
36         lock.lock();
37         try {
38             while (size == items.length)
39                 // 队列满了让生产者等待
40                 notFull.await();
41
42             items[putIndex] = value;
43             if (++putIndex == items.length)
44                 putIndex = 0;
45             size++;
46             notEmpty.signal(); // 生产完唤醒消费者
47
48         } finally {
49             System.out.println("producer生产: " + value);
50             //解锁
51             lock.unlock();
52         }
53     }
54
55     public Object take() throws Exception {
56         lock.lock();
57         try {
```



```

58         // 队列空了就让消费者等待
59         while (size == 0)
60             notEmpty.await();
61
62         Object value = items[takeIndex];
63         items[takeIndex] = null;
64         if (++takeIndex == items.length)
65             takeIndex = 0;
66         size--;
67         notFull.signal(); //消费完唤醒生产者生产
68         return value;
69     } finally {
70         lock.unlock();
71     }
72 }
73 }
74
75 /**
76  * 生产者
77  */
78 class Producer implements Runnable {
79
80     private Queue queue;
81
82     public Producer(Queue queue) {
83         this.queue = queue;
84     }
85
86     @Override
87     public void run() {
88         try {
89             // 隔1秒轮询生产一次
90             while (true) {
91                 Thread.sleep(1000);
92                 queue.put(new Random().nextInt(1000));
93             }
94         } catch (Exception e) {
95             e.printStackTrace();
96         }
97     }

```

```
98  }
99
100 /**
101  * 消费者
102  */
103 class Customer implements Runnable {
104
105     private Queue queue;
106
107     public Customer(Queue queue) {
108         this.queue = queue;
109     }
110
111     @Override
112     public void run() {
113         try {
114             // 隔2秒轮询消费一次
115             while (true) {
116                 Thread.sleep(2000);
117                 System.out.println("consumer消费: " + queue.take());
118             }
119         } catch (Exception e) {
120             e.printStackTrace();
121         }
122     }
123 }
```

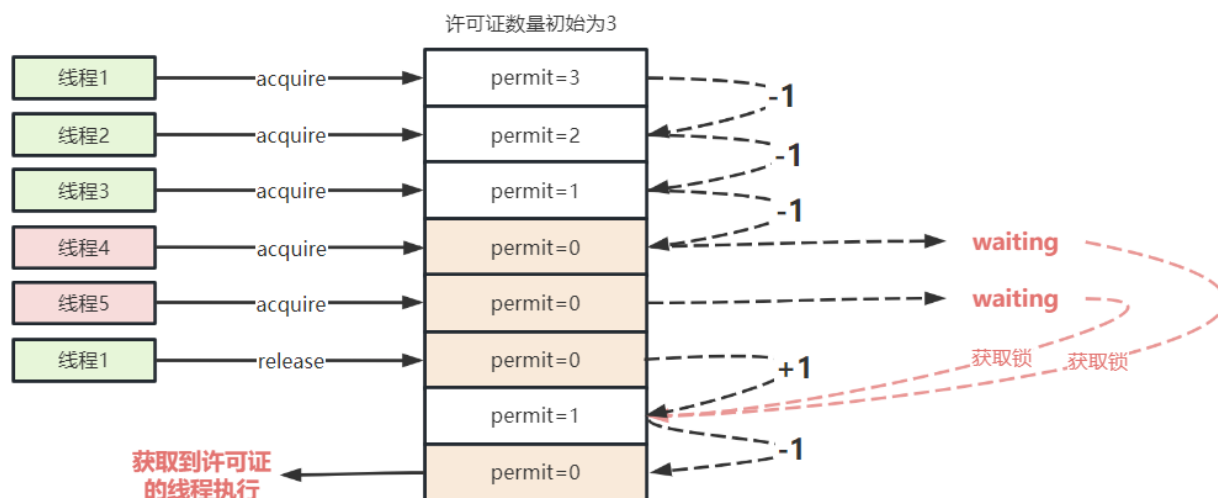
1.3 应用场景总结

ReentrantLock具体应用场景如下：

1. 解决多线程竞争资源的问题，例如多个线程同时对同一个数据库进行写操作，可以使用ReentrantLock保证每次只有一个线程能够写入。
2. 实现多线程任务的顺序执行，例如在一个线程执行完某个任务后，再让另一个线程执行任务。
3. 实现多线程等待/通知机制，例如在某个线程执行完某个任务后，通知其他线程继续执行任务。

2. Semaphore

Semaphore（信号量）是一种用于多线程编程的同步工具，**用于控制同时访问某个资源的线程数量。**



Semaphore维护了一个计数器，线程可以通过调用acquire()方法来获取Semaphore中的许可证，当计数器为0时，调用acquire()的线程将被阻塞，直到有其他线程释放许可证；线程可以通过调用release()方法来释放Semaphore中的许可证，这会使Semaphore中的计数器增加，从而允许更多的线程访问共享资源。

2.1 常用API

构造器

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

- permits 表示许可证的数量（资源数）
- fair 表示公平性，如果这个设为 true 的话，下次执行的线程会是等待最久的线程

常用方法

- acquire() 表示阻塞并获取许可
- tryAcquire() 方法在没有许可的情况下会立即返回 false，要获取许可的线程不会阻塞
- release() 表示释放许可

2.2 Semaphore使用

Semaphore实现服务接口限流

```
1  @Slf4j
2  public class SemaphoreDemo {
3
4      /**
5       * 同一时刻最多只允许有两个并发
6       */
7      private static Semaphore semaphore = new Semaphore(2);
8
9      private static Executor executor = Executors.newFixedThreadPool(10);
10
11     public static void main(String[] args) {
12         for(int i=0;i<10;i++){
13             executor.execute(()->getProductInfo2());
14         }
15     }
16
17     public static String getProductInfo() {
18         try {
19             semaphore.acquire();
20             log.info("请求服务");
21             Thread.sleep(2000);
22         } catch (InterruptedException e) {
23             throw new RuntimeException(e);
24         }finally {
25             semaphore.release();
26         }
27         return "返回商品详情信息";
28     }
29
30     public static String getProductInfo2() {
31
32         if(!semaphore.tryAcquire()){
33             log.error("请求被流控了");
34             return "请求被流控了";
35         }
36         try {
37             log.info("请求服务");
```

```

38         Thread.sleep(2000);
39     } catch (InterruptedException e) {
40         throw new RuntimeException(e);
41     }finally {
42         semaphore.release();
43     }
44     return "返回商品详情信息";
45 }
46 }

```

Semaphore实现数据库连接池

```

1  public class SemaphoreDemo2 {
2
3      public static void main(String[] args) {
4          final ConnectPool pool = new ConnectPool(2);
5          ExecutorService executorService = Executors.newCachedThreadPool();
6
7          //5个线程并发来争抢连接资源
8          for (int i = 0; i < 5; i++) {
9              final int id = i + 1;
10             executorService.execute(new Runnable() {
11                 @Override
12                 public void run() {
13                     Connect connect = null;
14                     try {
15                         System.out.println("线程" + id + "等待获取数据库连接");
16                         connect = pool.openConnect();
17                         System.out.println("线程" + id + "已拿到数据库连接:" + connect);
18                         //进行数据库操作2秒...然后释放连接
19                         Thread.sleep(2000);
20                         System.out.println("线程" + id + "释放数据库连接:" + connect);
21
22                     } catch (InterruptedException e) {
23                         e.printStackTrace();
24                     } finally {
25                         pool.releaseConnect(connect);
26                     }

```

```
27
28         }
29     });
30 }
31 }
32 }
33
34 //数据库连接池
35 class ConnectPool {
36     private int size;
37     private Connect[] connects;
38
39     //记录对应下标的Connect是否已被使用
40     private boolean[] connectFlag;
41     //信号量对象
42     private Semaphore semaphore;
43
44     /**
45      * size:初始化连接池大小
46      */
47     public ConnectPool(int size) {
48         this.size = size;
49         semaphore = new Semaphore(size, true);
50         connects = new Connect[size];
51         connectFlag = new boolean[size];
52         initConnects();//初始化连接池
53     }
54
55     private void initConnects() {
56         for (int i = 0; i < this.size; i++) {
57             connects[i] = new Connect();
58         }
59     }
60
61     /**
62      * 获取数据库连接
63      *
64      * @return
65      * @throws InterruptedException
66      */
```

```

67     public Connect openConnect() throws InterruptedException {
68         //得先获得使用许可证，如果信号量为0，则拿不到许可证，一直阻塞直到能获得
69         semaphore.acquire();
70         return getConnect();
71     }
72
73     private synchronized Connect getConnect() {
74         for (int i = 0; i < connectFlag.length; i++) {
75             if (!connectFlag[i]) {
76                 //标记该连接已被使用
77                 connectFlag[i] = true;
78                 return connects[i];
79             }
80         }
81         return null;
82     }
83
84     /**
85      * 释放某个数据库连接
86      */
87     public synchronized void releaseConnect(Connect connect) {
88         for (int i = 0; i < this.size; i++) {
89             if (connect == connects[i]) {
90                 connectFlag[i] = false;
91                 semaphore.release();
92             }
93         }
94     }
95 }
96
97 /**
98  * 数据库连接
99  */
100 class Connect {
101
102     private static int count = 1;
103     private int id = count++;
104
105     public Connect() {

```

```
106         //假设打开一个连接很耗费资源，需要等待1秒
107         try {
108             Thread.sleep(1000);
109         } catch (InterruptedException e) {
110             e.printStackTrace();
111         }
112         System.out.println("连接#" + id + "已与数据库建立通道！");
113     }
114
115     @Override
116     public String toString() {
117         return "#" + id + "#";
118     }
119 }
120
121 }
```

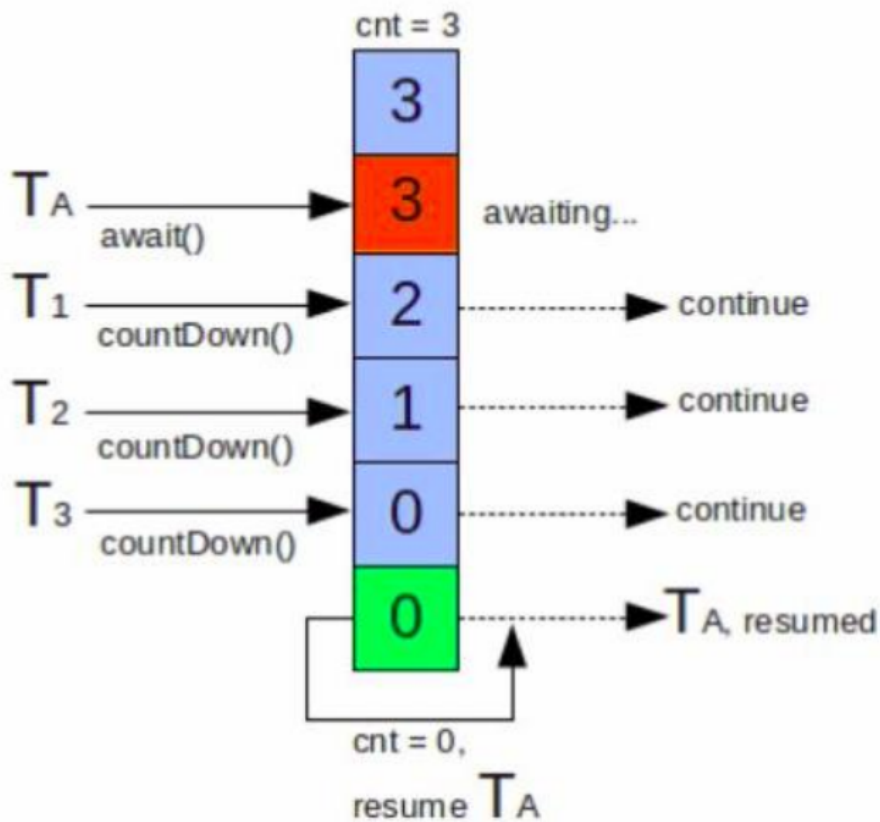
2.3 应用场景总结

以下是一些使用Semaphore的常见场景：

1. 限流：Semaphore可以用于限制对共享资源的并发访问数量，以控制系统的流量。
2. 资源池：Semaphore可以用于实现资源池，以维护一组有限的共享资源。

3. CountdownLatch

CountDownLatch（闭锁）是一个同步协助类，允许一个或多个线程等待，直到其他线程完成操作集。



`CountDownLatch`使用给定的计数值（count）初始化。`await`方法会阻塞直到当前的计数值（count），由于`countDown`方法的调用达到0，count为0之后所有等待的线程都会被释放，并且随后对`await`方法的调用都会立即返回。这是一个一次性现象——count不会被重置。

3.1 常用API

构造器

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

常用方法

```
1 // 调用 await() 方法的线程会被挂起，它会等待直到 count 值为 0 才继续执行  
2 public void await() throws InterruptedException { };  
3 // 和 await() 类似，若等待 timeout 时长后，count 值还是没有变为 0，不再等待，继续执行  
4 public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };  
5 // 会将 count 减 1，直至为 0  
6 public void countDown() { };
```

3.2 CountdownLatch使用

模拟实现百米赛跑

```
1 public class CountdownLatchDemo {
2     // begin 代表裁判 初始为 1
3     private static CountdownLatch begin = new CountdownLatch(1);
4
5     // end 代表玩家 初始为 8
6     private static CountdownLatch end = new CountdownLatch(8);
7
8     public static void main(String[] args) throws InterruptedException {
9
10        for (int i = 1; i <= 8; i++) {
11            new Thread(new Runnable() {
12                @SneakyThrows
13                @Override
14                public void run() {
15                    // 预备状态
16                    System.out.println("参赛者"+Thread.currentThread().getName()+ "已经
准备好了");
17                    // 等待裁判吹哨
18                    begin.await();
19                    // 开始跑步
20                    System.out.println("参赛者"+Thread.currentThread().getName()+ "开始
跑步");
21                    Thread.sleep(1000);
22                    // 跑步结束，跑完了
23                    System.out.println("参赛者"+Thread.currentThread().getName()+ "到达
终点");
24                    // 跑到终点，计数器就减一
25                    end.countDown();
26                }
27            }).start();
28        }
29        // 等待 5s 就开始吹哨
30        Thread.sleep(5000);
31        System.out.println("开始比赛");
32        // 裁判吹哨，计数器减一
33        begin.countDown();
```

```
34         // 等待所有玩家到达终点
35         end.await();
36         System.out.println("比赛结束");
37
38     }
```

多任务完成后合并汇总

很多时候，我们的并发任务，存在前后依赖关系；比如数据详情页需要同时调用多个接口获取数据，并发请求获取到数据后、需要进行结果合并；或者多个数据操作完成后，需要数据check。

```
1  public class CountdownLatchDemo2 {
2      public static void main(String[] args) throws Exception {
3
4          CountdownLatch countDownLatch = new CountdownLatch(5);
5          for (int i = 0; i < 5; i++) {
6              final int index = i;
7              new Thread(() -> {
8                  try {
9                      Thread.sleep(1000 + ThreadLocalRandom.current().nextInt(2000));
10                     System.out.println("任务" + index + "执行完成");
11                     countDownLatch.countDown();
12                 } catch (InterruptedException e) {
13                     e.printStackTrace();
14                 }
15             }).start();
16          }
17
18          // 主线程在阻塞，当计数器为0，就唤醒主线程往下执行
19          countDownLatch.await();
20          System.out.println("主线程:在所有任务运行完成后，进行结果汇总");
21      }
22  }
```

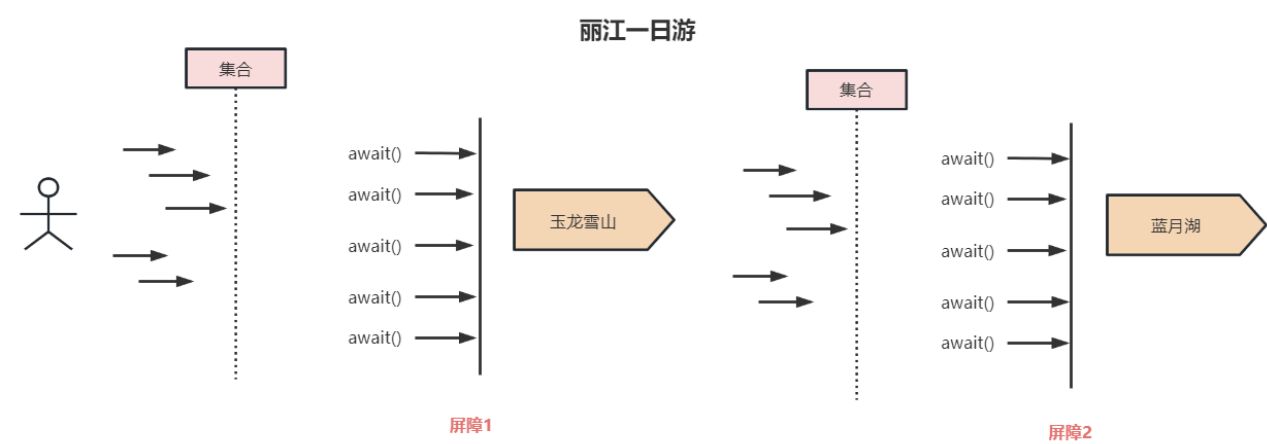
3.3 应用场景总结

以下是使用CountDownLatch的常见场景：

1. 并行任务同步：CountDownLatch可以用于协调多个并行任务的完成情况，确保所有任务都完成后再继续执行下一步操作。
2. 多任务汇总：CountDownLatch可以用于统计多个线程的完成情况，以确定所有线程都已完成工作。
3. 资源初始化：CountDownLatch可以用于等待资源的初始化完成，以便在资源初始化完成后开始使用。

4. CyclicBarrier

CyclicBarrier（回环栅栏或循环屏障），是 Java 并发库中的一个同步工具，通过它可以实现让一组线程等待至某个状态（屏障点）之后再全部同时执行。叫做回环是因为当所有等待线程都被释放以后，CyclicBarrier可以被重用。

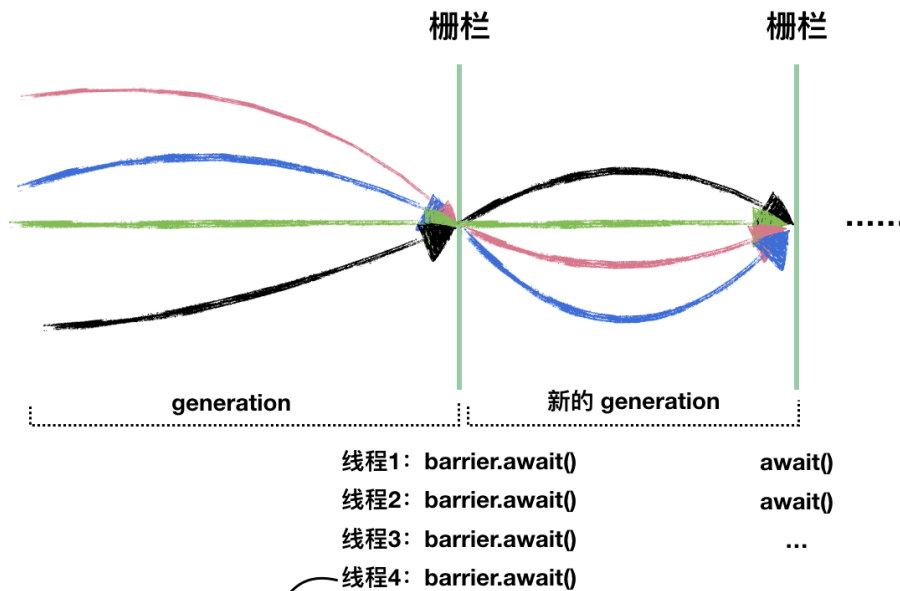


4.1 常用API

构造器

```
1 // parties表示屏障拦截的线程数量，每个线程调用 await 方法告诉 CyclicBarrier 我已经到达了屏障，然后当前线程被阻塞。
2 public CyclicBarrier(int parties)
3 // 用于在线程到达屏障时，优先执行 barrierAction，方便处理更复杂的业务场景(该线程的执行时机是在到达屏障之后再执行)
4 public CyclicBarrier(int parties, Runnable barrierAction)
```

```
CyclicBarrier barrier = new CyclicBarrier(4, new Runnable(){...});
```



第 4 个线程也 `await` 的时候, 那么就是来齐了, 大家一起通过栅栏

1. `parties = 4`, 代表有 4 个线程参与;
2. `count` 初始值是 4, 随着每个线程 `await` 一次, `count` 减 1, 穿过栅栏后重置为 4
3. 构造方法的第二个参数是 `Runnable` 的实例, 代表在大家都到达栅栏, 在通过之前需要执行的动作, 由最后一个到达栅栏的线程执行。如果没有需要执行的, 传 `null`

常用方法

```
1 //指定数量的线程全部调用await()方法时, 这些线程不再阻塞
2 // BrokenBarrierException 表示栅栏已经被破坏, 破坏的原因可能是其中一个线程 await() 时被中断
  或者超时
3 public int await() throws InterruptedException, BrokenBarrierException
4 public int await(long timeout, TimeUnit unit) throws InterruptedException,
  BrokenBarrierException, TimeoutException
5
6 //循环 通过reset()方法可以进行重置
7 public void reset()
```

4.2 CyclicBarrier使用

模拟人满发车

利用CyclicBarrier的计数器能够重置，屏障可以重复使用的特性，可以支持类似“人满发车”的场景

```
1 public class CyclicBarrierDemo {
2
3     public static void main(String[] args) {
4
5         ExecutorService executorService = Executors.newFixedThreadPool(5);
6
7         CyclicBarrier cyclicBarrier = new CyclicBarrier(5,
8             () -> System.out.println("人齐了，准备发车"));
9
10        for (int i = 0; i < 10; i++) {
11            final int id = i+1;
12            executorService.submit(new Runnable() {
13                @Override
14                public void run() {
15                    try {
16                        System.out.println(id+"号马上就到");
17                        int sleepMills = ThreadLocalRandom.current().nextInt(2000);
18                        Thread.sleep(sleepMills);
19                        System.out.println(id + "号到了，上车");
20                        cyclicBarrier.await();
21
22                    } catch (InterruptedException e) {
23                        e.printStackTrace();
24                    } catch (BrokenBarrierException e){
25                        e.printStackTrace();
26                    }
27                }
28            });
29        }
30
31    }
32
33
34 }
```

多线程批量处理数据

```
1 public class CyclicBarrierBatchProcessorDemo {
2
3     public static void main(String[] args) {
4         //生成数据
5         List<Integer> data = new ArrayList<>();
6         for (int i = 1; i <= 50; i++) {
7             data.add(i);
8         }
9
10        //指定数据处理大小
11        int batchSize = 5;
12        CyclicBarrierBatchProcessor processor = new CyclicBarrierBatchProcessor(data,
13        batchSize);
14        //处理数据
15        processor.process(batchData -> {
16            for (Integer i : batchData) {
17                System.out.println(Thread.currentThread().getName() + "处理数据" + i);
18            }
19        });
20    }
21
22
23    class CyclicBarrierBatchProcessor {
24        private List<Integer> data;
25        private int batchSize;
26        private CyclicBarrier barrier;
27        private List<Thread> threads;
28
29        public CyclicBarrierBatchProcessor(List<Integer> data, int batchSize) {
30            this.data = data;
31            this.batchSize = batchSize;
32            this.barrier = new CyclicBarrier(batchSize);
33            this.threads = new ArrayList<>();
34        }
35
36        public void process(BatchTask task) {
```

```

37         // 对任务分批，获取线程数
38         int threadCount = (data.size() + batchSize - 1) / batchSize;
39         for (int i = 0; i < threadCount; i++) {
40             int start = i * batchSize;
41             int end = Math.min(start + batchSize, data.size());
42             //获取每个线程处理的任务数
43             List<Integer> batchData = data.subList(start, end);
44             Thread thread = new Thread(() -> {
45                 task.process(batchData);
46                 try {
47                     barrier.await();
48                 } catch (InterruptedException | BrokenBarrierException e) {
49                     e.printStackTrace();
50                 }
51             });
52             threads.add(thread);
53             thread.start();
54         }
55
56     }
57
58     public interface BatchTask {
59         void process(List<Integer> batchData);
60     }
61 }

```

4.3 应用场景总结

以下是一些常见的 CyclicBarrier 应用场景：

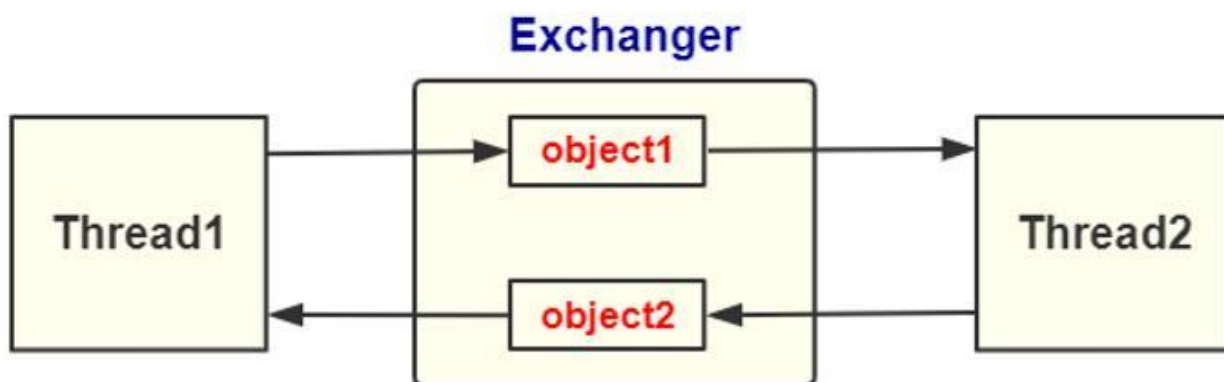
1. 多线程任务：CyclicBarrier 可以用于将复杂的任务分配给多个线程执行，并在所有线程完成工作后触发后续操作。
2. 数据处理：CyclicBarrier 可以用于协调多个线程间的数据处理，在所有线程处理完数据后触发后续操作。

4.4 CyclicBarrier 与 CountdownLatch 区别

- CountdownLatch 是一次性的，CyclicBarrier 是可循环利用的
- CountdownLatch 参与的线程的职责是不一样的，有的在倒计时，有的在等待倒计时结束。CyclicBarrier 参与的线程职责是一样的。

5. Exchanger

Exchanger是一个用于线程间协作的工具类，**用于两个线程间交换数据**。具体交换数据是通过exchange方法来实现的，如果一个线程先执行exchange方法，那么它会同步等待另一个线程也执行exchange方法，这个时候两个线程就都达到了同步点，两个线程就可以交换数据。



5.1 常用API

```
1 public V exchange(V x) throws InterruptedException
2 public V exchange(V x, long timeout, TimeUnit unit) throws InterruptedException,
   TimeoutException
```

- `V exchange(V v)`: 等待另一个线程到达此交换点（除非当前线程被中断），然后将给定的对象传送给该线程，并接收该线程的对象。
- `V exchange(V v, long timeout, TimeUnit unit)`: 等待另一个线程到达此交换点，或者当前线程被中断——抛出中断异常；又或者是等候超时——抛出超时异常，然后将给定的对象传送给该线程，并接收该线程的对象。

5.2 Exchanger使用

模拟交易场景

用一个简单的例子来看下Exchanger的具体使用。两方做交易，如果一方先到要等另一方也到了才能交易，交易就是执行exchange方法交换数据。

```
1 public class ExchangerDemo {
2     private static Exchanger exchanger = new Exchanger();
3     static String goods = "电脑";
4     static String money = "$4000";
```

```

5     public static void main(String[] args) throws InterruptedException {
6
7         System.out.println("准备交易，一手交钱一手交货...");
8         // 卖家
9         new Thread(new Runnable() {
10             @Override
11             public void run() {
12                 System.out.println("卖家到了，已经准备好货：" + goods);
13                 try {
14                     String money = (String) exchanger.exchange(goods);
15                     System.out.println("卖家收到钱：" + money);
16                 } catch (Exception e) {
17                     e.printStackTrace();
18                 }
19             }
20         }).start();
21
22         Thread.sleep(3000);
23
24         // 买家
25         new Thread(new Runnable() {
26             @Override
27             public void run() {
28                 try {
29                     System.out.println("买家到了，已经准备好钱：" + money);
30                     String goods = (String) exchanger.exchange(money);
31                     System.out.println("买家收到货：" + goods);
32                 } catch (Exception e) {
33                     e.printStackTrace();
34                 }
35             }
36         }).start();
37
38     }
39 }

```

模拟对账场景

```
1 public class ExchangerDemo2 {
2
3     private static final Exchanger<String> exchanger = new Exchanger();
4     private static ExecutorService threadPool = Executors.newFixedThreadPool(2);
5
6     public static void main(String[] args) {
7
8         threadPool.execute(new Runnable() {
9             @Override
10             public void run() {
11                 try {
12                     String A = "12379871924sfkhfksdhfks";
13                     exchanger.exchange(A);
14                 } catch (InterruptedException e) {
15                 }
16             }
17         });
18
19         threadPool.execute(new Runnable() {
20             @Override
21             public void run() {
22                 try {
23                     String B = "32423423jknjkfsbfj";
24                     String A = exchanger.exchange(B);
25                     System.out.println("A和B数据是否一致: " + A.equals(B));
26                     System.out.println("A= "+A);
27                     System.out.println("B= "+B);
28                 } catch (InterruptedException e) {
29                 }
30             }
31         });
32
33         threadPool.shutdown();
34
35     }
36 }
```

模拟队列中交换数据场景

```
1 public class ExchangerDemo3 {
2
3     private static ArrayBlockingQueue<String> fullQueue
4         = new ArrayBlockingQueue<>(5);
5     private static ArrayBlockingQueue<String> emptyQueue
6         = new ArrayBlockingQueue<>(5);
7     private static Exchanger<ArrayBlockingQueue<String>> exchanger
8         = new Exchanger<>();
9
10
11     public static void main(String[] args) {
12         new Thread(new Producer()).start();
13         new Thread(new Consumer()).start();
14
15     }
16
17     /**
18      * 生产者
19      */
20     static class Producer implements Runnable {
21         @Override
22         public void run() {
23             ArrayBlockingQueue<String> current = emptyQueue;
24             try {
25                 while (current != null) {
26                     String str = UUID.randomUUID().toString();
27                     try {
28                         current.add(str);
29                         System.out.println("producer: 生产了一个序列: " + str + ">>>>加入到交换区");
30                         Thread.sleep(2000);
31                     } catch (IllegalStateException e) {
32                         System.out.println("producer: 队列已满, 换一个空的");
33                         current = exchanger.exchange(current);
34                     }
35                 }
36             } catch (Exception e) {
```

```

37         e.printStackTrace();
38     }
39 }
40 }
41
42 /**
43  * 消费者
44  */
45 static class Consumer implements Runnable {
46     @Override
47     public void run() {
48         ArrayBlockingQueue<String> current = fullQueue;
49         try {
50             while (current != null) {
51                 if (!current.isEmpty()) {
52                     String str = current.poll();
53                     System.out.println("consumer: 消耗一个序列: " + str);
54                     Thread.sleep(1000);
55                 } else {
56                     System.out.println("consumer: 队列空了, 换个满的");
57                     current = exchanger.exchange(current);
58                     System.out.println("consumer: 换满的成功
59 ~~~~~~");
60                 }
61             } catch (Exception e) {
62                 e.printStackTrace();
63             }
64         }
65     }
66
67
68 }
69

```

5.3 应用场景总结

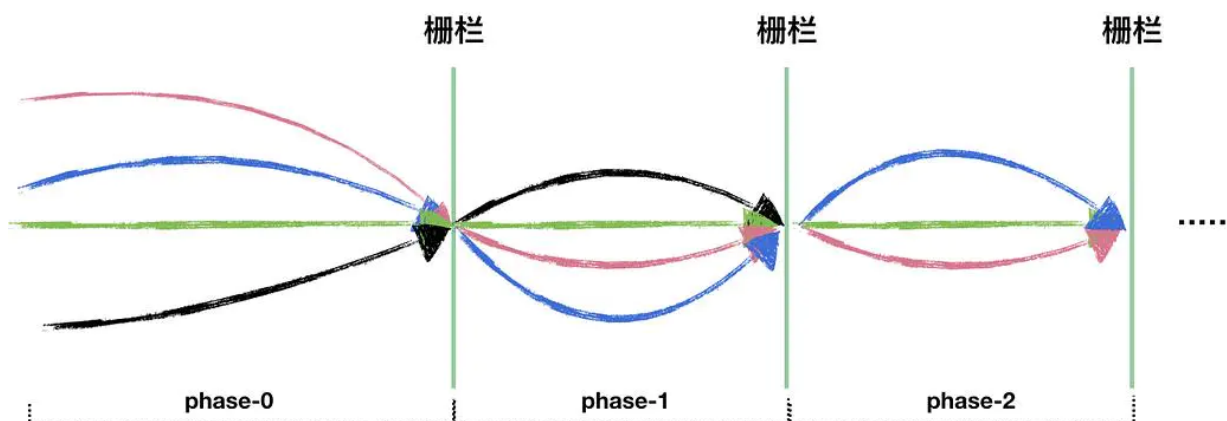
Exchanger 可以用于各种应用场景，具体取决于具体的 Exchanger 实现。常见的场景包括：

1. 数据交换：在多线程环境中，两个线程可以通过 Exchanger 进行数据交换。

2. 数据采集：在数据采集系统中，可以使用 Exchanger 在采集线程和处理线程间进行数据交换。

6. Phaser

Phaser（阶段协同器）是一个Java实现的并发工具类，用于协调多个线程的执行。它提供了一些方便的方法来管理多个阶段的执行，可以让程序员灵活地控制线程的执行顺序和阶段性的执行。Phaser可以被视为CyclicBarrier和CountDownLatch的进化版，它能够自适应地调整并发线程数，可以动态地增加或减少参与线程的数量。所以Phaser特别适合使用在重复执行或者重用的情况。



6.1 常用API

构造方法

- `Phaser()`: 参与任务数0
- `Phaser(int parties)`: 指定初始参与任务数
- `Phaser(Phaser parent)`: 指定parent阶段器，子对象作为一个整体加入parent对象，当子对象中没有参与者时，会自动从parent对象解除注册
- `Phaser(Phaser parent, int parties)`: 集合上面两个方法

增减参与任务数方法

- `int register()` 增加一个任务数，返回当前阶段号。
- `int bulkRegister(int parties)` 增加指定任务个数，返回当前阶段号。
- `int arriveAndDeregister()` 减少一个任务数，返回当前阶段号。

到达、等待方法

- `int arrive()` 到达(任务完成)，返回当前阶段号。
- `int arriveAndAwaitAdvance()` 到达后等待其他任务到达，返回到达阶段号。
- `int awaitAdvance(int phase)` 在指定阶段等待(必须是当前阶段才有效)
- `int awaitAdvanceInterruptibly(int phase)` 阶段到达触发动作
- `int awaitAdvanceInterruptibly(int phase, long timeout, TimeUnit unit)`
- `protected boolean onAdvance(int phase, int registeredParties)`类似CyclicBarrier的触发命令，通过重写该

方法来增加阶段到达动作，该方法返回true将终结Phaser对象。

6.2 Phaser使用

多线程批量处理数据

```
1 public class PhaserBatchProcessorDemo {
2
3     private final List<String> data;
4     private final int batchSize; //一次处理多少数据
5     private final int threadCount; //处理的线程数
6     private final Phaser phaser;
7     private final List<String> processedData;
8
9     public PhaserBatchProcessorDemo(List<String> data, int batchSize, int threadCount)
10    {
11        this.data = data;
12        this.batchSize = batchSize;
13        this.threadCount = threadCount;
14        this.phaser = new Phaser(1);
15        this.processedData = new ArrayList<>();
16    }
17
18    public void process() {
19        for (int i = 0; i < threadCount; i++) {
20
21            phaser.register();
22            new Thread(new BatchProcessor(i)).start();
23        }
24
25        phaser.arriveAndDeregister();
26    }
27
28    private class BatchProcessor implements Runnable {
29
30        private final int threadIndex;
31
32        public BatchProcessor(int threadIndex) {
```

```
32         this.threadIndex = threadIndex;
33     }
34
35     @Override
36     public void run() {
37         int index = 0;
38         while (true) {
39             // 所有线程都到达这个点之前会阻塞
40             phaser.arriveAndAwaitAdvance();
41
42             // 从未处理数据中找到一个可以处理的批次
43             List<String> batch = new ArrayList<>();
44             synchronized (data) {
45                 while (index < data.size() && batch.size() < batchSize) {
46                     String d = data.get(index);
47                     if (!processedData.contains(d)) {
48                         batch.add(d);
49                         processedData.add(d);
50                     }
51                     index++;
52                 }
53             }
54
55             // 处理数据
56             for (String d : batch) {
57                 System.out.println("线程" + threadIndex + "处理数据" + d);
58             }
59
60             // 所有线程都处理完当前批次之前会阻塞
61             phaser.arriveAndAwaitAdvance();
62
63             // 所有线程都处理完当前批次并且未处理数据已经处理完之前会阻塞
64             if (batch.isEmpty() || index >= data.size()) {
65                 phaser.arriveAndDeregister();
66                 break;
67             }
68         }
69     }
70 }
71
```



```

72     public static void main(String[] args) {
73         //数据准备
74         List<String> data = new ArrayList<>();
75         for (int i = 1; i <= 15; i++) {
76             data.add(String.valueOf(i));
77         }
78
79         int batchSize = 4;
80         int threadCount = 3;
81         PhaserBatchProcessorDemo processor = new PhaserBatchProcessorDemo(data,
            batchSize, threadCount);
82         //处理数据
83         processor.process();
84     }
85 }
86

```

阶段性任务：模拟公司团建

```

1  public class PhaserDemo {
2      public static void main(String[] args) {
3          final Phaser phaser = new Phaser() {
4              //重写该方法来增加阶段到达动作
5              @Override
6              protected boolean onAdvance(int phase, int registeredParties) {
7                  // 参与者数量，去除主线程
8                  int staffs = registeredParties - 1;
9                  switch (phase) {
10                     case 0:
11                         System.out.println("大家都到公司了，出发去公园，人数: " + staffs);
12                         break;
13                     case 1:
14                         System.out.println("大家都到公园门口了，出发去餐厅，人数: " +
15 staffs);
16                         break;
17                     case 2:
18                         System.out.println("大家都到餐厅了，开始用餐，人数: " + staffs);
19                         break;

```

```

20         }
21
22         // 判断是否只剩下主线程（一个参与者），如果是，则返回true，代表终止
23         return registeredParties == 1;
24     }
25 };
26
27 // 注册主线程 —— 让主线程全程参与
28 phaser.register();
29 final StaffTask staffTask = new StaffTask();
30
31 // 3个全程参与团建的员工
32 for (int i = 0; i < 3; i++) {
33     // 添加任务数
34     phaser.register();
35     new Thread(() -> {
36         try {
37             staffTask.step1Task();
38             //到达后等待其他任务到达
39             phaser.arriveAndAwaitAdvance();
40
41             staffTask.step2Task();
42             phaser.arriveAndAwaitAdvance();
43
44             staffTask.step3Task();
45             phaser.arriveAndAwaitAdvance();
46
47             staffTask.step4Task();
48             // 完成了，注销离开
49             phaser.arriveAndDeregister();
50         } catch (InterruptedException e) {
51             e.printStackTrace();
52         }
53     }).start();
54 }
55
56 // 两个不聚餐的员工加入
57 for (int i = 0; i < 2; i++) {
58     phaser.register();
59     new Thread(() -> {

```

```

60         try {
61             staffTask.step1Task();
62             phaser.arriveAndAwaitAdvance();
63
64             staffTask.step2Task();
65             System.out.println("员工【" + Thread.currentThread().getName() + "】
回家了");
66             // 完成了，注销离开
67             phaser.arriveAndDeregister();
68         } catch (InterruptedException e) {
69             e.printStackTrace();
70         }
71     }).start();
72 }
73
74 while (!phaser.isTerminated()) {
75     int phase = phaser.arriveAndAwaitAdvance();
76     if (phase == 2) {
77         // 到了去餐厅的阶段，又新增4人，参加晚上的聚餐
78         for (int i = 0; i < 4; i++) {
79             phaser.register();
80             new Thread(() -> {
81                 try {
82                     staffTask.step3Task();
83                     phaser.arriveAndAwaitAdvance();
84
85                     staffTask.step4Task();
86                     // 完成了，注销离开
87                     phaser.arriveAndDeregister();
88                 } catch (InterruptedException e) {
89                     e.printStackTrace();
90                 }
91             }).start();
92         }
93     }
94 }
95 }
96
97 static final Random random = new Random();
98

```

```
99     static class StaffTask {
100         public void step1Task() throws InterruptedException {
101             // 第一阶段：来公司集合
102             String staff = "员工【" + Thread.currentThread().getName() + "】";
103             System.out.println(staff + "从家出发了.....");
104             Thread.sleep(random.nextInt(5000));
105             System.out.println(staff + "到达公司");
106         }
107
108         public void step2Task() throws InterruptedException {
109             // 第二阶段：出发去公园
110             String staff = "员工【" + Thread.currentThread().getName() + "】";
111             System.out.println(staff + "出发去公园玩");
112             Thread.sleep(random.nextInt(5000));
113             System.out.println(staff + "到达公园门口集合");
114
115         }
116
117         public void step3Task() throws InterruptedException {
118             // 第三阶段：去餐厅
119             String staff = "员工【" + Thread.currentThread().getName() + "】";
120             System.out.println(staff + "出发去餐厅");
121             Thread.sleep(random.nextInt(5000));
122             System.out.println(staff + "到达餐厅");
123
124         }
125
126         public void step4Task() throws InterruptedException {
127             // 第四阶段：就餐
128             String staff = "员工【" + Thread.currentThread().getName() + "】";
129             System.out.println(staff + "开始用餐");
130             Thread.sleep(random.nextInt(5000));
131             System.out.println(staff + "用餐结束，回家");
132         }
133     }
134 }
```

6.3 应用场景总结

以下是一些常见的 Phaser 应用场景：

1. 多线程任务分配：Phaser 可以用于将复杂的任务分配给多个线程执行，并协调线程间的合作。
2. 多级任务流程：Phaser 可以用于实现多级任务流程，在每一级任务完成后触发下一级任务的开始。
3. 模拟并行计算：Phaser 可以用于模拟并行计算，协调多个线程间的工作。
4. 阶段性任务：Phaser 可以用于实现阶段性任务，在每一阶段任务完成后触发下一阶段任务的开始。