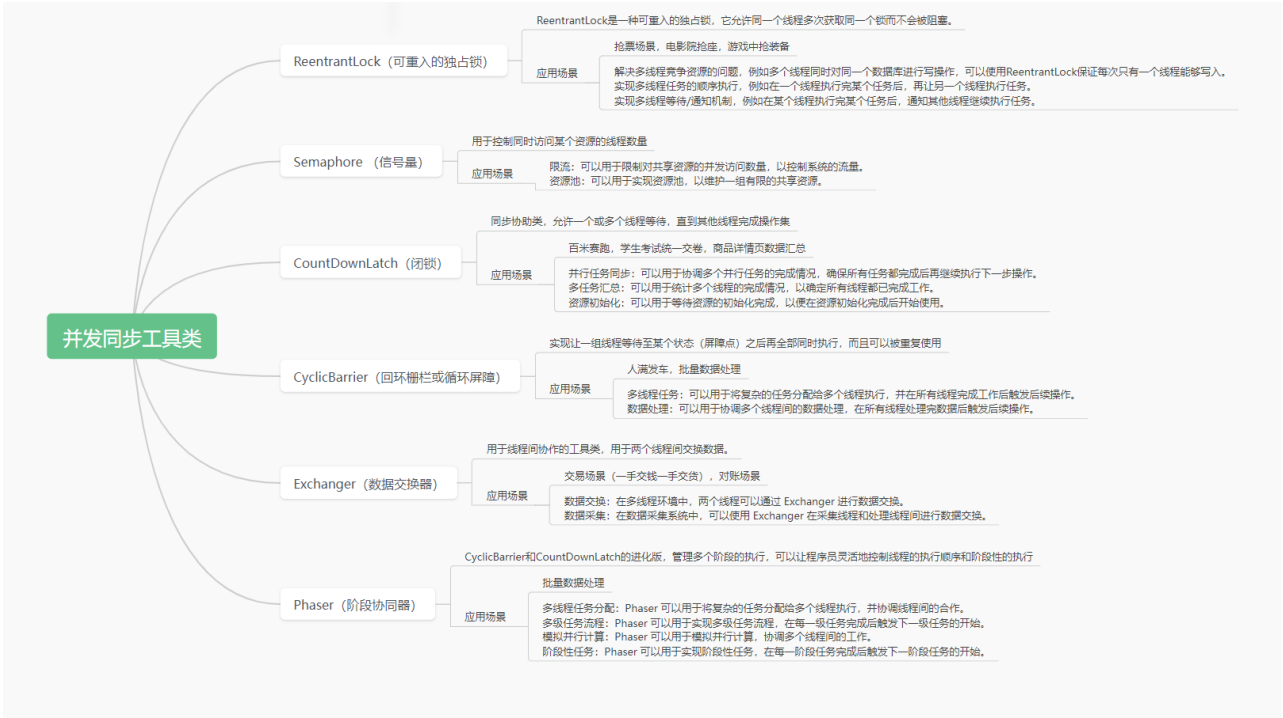
主讲老师：Fox

> 有道笔记地址：https://note.youdao.com/s/dbBcYl4a

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。

> https://www.processon.com/view/link/6620b9d763dc8148f6486eda?cid=63f364586e3252660403d78c



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

```java
private final Lock lock = new ReentrantLock();
public void foo()
{
    // 获取锁
    lock.lock();
    try
    {
        // 程序执行逻辑
    } finally
    {
        // finally语句块可以确保lock被正确释放
        lock.unlock();
    }
}


// 尝试获取锁，最多等待 100 毫秒
if (lock.tryLock(100, TimeUnit.MILLISECONDS)) {
    try {
        // 成功获取到锁，执行需要同步的代码块
        // ... 执行一些操作 ...
    } finally {
        // 释放锁
        lock.unlock();
    }
} else {
    // 超时后仍未获取到锁，执行备选逻辑
    // ... 执行一些不需要同步的操作 ...
}
```

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。

工作原理

当有线程调用lock方法的时候： 如果线程获取到锁了，那么就会通过CAS的方式把AQS内部的state设置成为1。这个时候，当前线程就获取到锁了。只有首部的节点(head节点封装的线程)可以获取到锁。其他线程都会加入到这一个阻塞队列当中。如果是公平锁的话，当head节点释放锁之后，会优先唤醒head.next这一个节点对应的线程。如果是非公平锁，允许新来的线程和head之后唤醒的线程通过cas竞争锁。

1.2 ReentrantLock使用

独占锁：模拟抢票场景

思考：8张票，10个人抢，如果不加锁，会出现什么问题？

```java
/**
 *  模拟抢票场景
 */
public class ReentrantLockDemo {

    private final ReentrantLock lock = new ReentrantLock();//默认非公平
    private static int tickets = 8; // 总票数

    public void buyTicket() {
        lock.lock(); // 获取锁
        try {
            if (tickets > 0) { // 还有票
                try {
                    Thread.sleep(10); // 休眠10ms,模拟出并发效果
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
                System.out.println(Thread.currentThread().getName() + "购买了第" + tickets-- + "张票");
            } else {
                System.out.println("票已经卖完了，" + Thread.currentThread().getName() + "抢票失败");
            }

        } finally {
            lock.unlock(); // 释放锁
        }
    }


    public static void main(String[] args) {
        ReentrantLockDemo ticketSystem = new ReentrantLockDemo();
        for (int i = 1; i <= 10; i++) {
            Thread thread = new Thread(() -> {

                ticketSystem.buyTicket(); // 抢票

            }, "线程" + i);
            // 启动线程
```

```
            thread.start();

        }

        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println("剩余票数：" + tickets);
    }
}
```

不加锁的效果： 出现超卖的问题

加锁效果： 正常，两个人抢票失败

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

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

```java
class Counter {
    private final ReentrantLock lock = new ReentrantLock(); // 创建 ReentrantLock 对象

    public void recursiveCall(int num) {
        lock.lock(); // 获取锁
        try {
            if (num == 0) {
                return;
            }
            System.out.println("执行递归，num = " + num);
            recursiveCall(num - 1);
        } finally {
            lock.unlock(); // 释放锁
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Counter counter = new Counter(); // 创建计数器对象

        // 测试递归调用
        counter.recursiveCall(10);
    }
}
```

Condition详解

在Java中，Condition是一个接口，它提供了线程之间的协调机制，可以将它看作是一个更加灵活、更加强大的wait()和notify()机制，通常与Lock接口（比如ReentrantLock）一起使用。它的核心作用体现在两个方面，如下：

- **等待/通知机制**：它允许线程等待某个条件成立，或者通知其他线程某个条件已经满足，这与使用Object的wait()和notify()方法相似，但Condition提供了更高的灵活性和更多的控制。
- **多条件协调**：与每个Object只有一个内置的等待/通知机制不同，一个Lock可以对应多个Condition对象，这意味着可以为不同的等待条件创建不同的Condition，从而实现对多个等待线程集合的独立控制。

核心方法2

void	await()	使当前线程等待，直到被其他线程通过 signal() 或 signalAll() 方法唤醒，或者线程被中断，或者发生了其他不可预知的情况（如假唤醒）。该方法会在等待之前释放当前线程所持有的锁，在被唤醒后会再次尝试获取锁。
boolean	await(long time, TimeUnit unit)	使当前线程等待指定的时间，或者直到被其他线程通过 signal() 或 signalAll() 方法唤醒，或者线程被中断。如果在指定的时间内没有被唤醒，该方法将返回。在等待之前会释放当前线程所持有的锁，在被唤醒或超时后会再次尝试获取锁。
void	signal()	唤醒等待在此 Condition 上的一个线程。如果有多个线程正在等待，则选择其中的一个进行唤醒。被唤醒的线程将从其 await() 调用中返回，并重新尝试获取与此 Condition 关联的锁。
void	signalAll()	唤醒等待在此 Condition 上的所有线程。每个被唤醒的线程都将从其 await() 调用中返回，并重新尝试获取与此 Condition 关联的锁。

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

```java
public class ReentrantLockDemo3 {

    public static void main(String[] args) {
        // 创建队列
        Queue queue = new Queue(5);
        //启动生产者线程
        new Thread(new Producer(queue)).start();
        //启动消费者线程
        new Thread(new Customer(queue)).start();

    }
}

/**
 * 队列封装类
 */
class Queue {
    private Object[] items ;
    int size = 0;
    int takeIndex;
    int putIndex;
    private ReentrantLock lock;
    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
    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

    public Queue(int capacity){
        this.items = new Object[capacity];
        lock = new ReentrantLock();
        notEmpty = lock.newCondition();
        notFull =  lock.newCondition();
    }


    public void put(Object value) throws Exception {
        //加锁
        lock.lock();
        try {
            while (size == items.length)
```

```
                // 队列满了让生产者等待
                notFull.await();

            items[putIndex] = value;
            if (++putIndex == items.length)
                putIndex = 0;
            size++;
            notEmpty.signal(); // 生产完唤醒消费者

        } finally {
            System.out.println("producer生产: " + value);
            //解锁
            lock.unlock();
        }
    }

    public Object take() throws Exception {
        lock.lock();
        try {
            // 队列空了就让消费者等待
            while (size == 0)
                notEmpty.await();

            Object value = items[takeIndex];
            items[takeIndex] = null;
            if (++takeIndex == items.length)
                takeIndex = 0;
            size--;
            notFull.signal(); //消费完唤醒生产者生产
            return value;
        } finally {
            lock.unlock();
        }
    }
}

/**
 * 生产者
 */
```

```
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);
```

```
                System.out.println("consumer消费: " + queue.take());
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

1.3 应用场景总结

ReentrantLock的应用场景主要体现在多线程环境下对共享资源的独占式访问，以保证数据的一致性和安全性。

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，主要用于在一个时刻允许多个线程对共享资源进行并行操作的场景。

通常情况下，使用Semaphore的过程实际上是多个线程获取访问共享资源许可证的过程。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。

Semaphore的基本流程如图：

Semaphore的应用场景主要涉及到需要限制资源访问数量或控制并发访问的场景，例如数据库连接、文件访问、网络请求等。在这些场景中，Semaphore能够有效地协调线程对资源的访问，保证系统的稳定性和性能。

2.1 常用API

构造器

- public Semaphore(int permits)：定义Semaphore指定许可证数量（资源数），并且指定非公平的同步器，因此new Semaphore(n)实际上是等价于new Semaphore(n，false)的。
- public Semaphore(int permits, boolean fair)：定义Semaphore指定许可证数量的同时给定非公平或是公平同步器。

常用方法

- **acquire方法**

acquire方法是向Semaphore获取许可证，但是该方法比较偏执一些，获取不到就会一直等（陷入阻塞状态），Semaphore为我们提供了acquire方法的两种重载形式。

  - void **acquire()** throws InterruptedException：该方法会向Semaphore获取一个许可证，如果获取不到就会一直等待，直到Semaphore有可用的许可证为止，或者被其他线程中断。当然，如果有可用的许可证则会立即返回。
  - void **acquire(int permits)** throws InterruptedException：该方法会向Semaphore获取指定数量的许可证，如果获取不到就会一直等待，直到Semaphore有可用的相应数量的许可证为止，或者被其他线程中断。同样，如果有可用的permits个许可证则会立即返回。

```java
// 定义permit=1的Semaphore
final Semaphore semaphore = new Semaphore(1, true);
// 主线程直接抢先申请成功
semaphore.acquire();
Thread t = new Thread(() -> {
    try {
        // 线程t会进入阻塞，等待当前有可用的permit
        System.out.println("子线程等待获取permit");
        semaphore.acquire();
        System.out.println("子线程获取到permit");
    } catch (InterruptedException e) {
        e.printStackTrace();
    }finally {
        //释放permit
        semaphore.release();
    }
});
t.start();
TimeUnit.SECONDS.sleep(5);
System.out.println("主线程释放permit");
// 主线程休眠5秒后释放permit，线程t才能获取到permit
semaphore.release();
```

- **tryAcquire方法**

tryAcquire方法尝试向Semaphore获取许可证，如果此时许可证的数量少于申请的数量，则对应的线程会立即返回，结果为false表示申请失败，tryAcquire包含如下四种重载方法。

  - **tryAcquire()**：尝试获取Semaphore的许可证，该方法只会向Semaphore申请一个许可证，在Semaphore内部的可用许可证数量大于等于1的情况下，许可证将会获取成功，反之获取许可证则会失败，并且返回结果为false。
  - boolean **tryAcquire(long timeout, TimeUnit unit)** throws InterruptedException：该方法与tryAcquire无参方法类似，同样也是尝试获取一个许可证，但是增加了超时参数。如果在超时时间内还是没有可用的许可证，那么线程就会进入阻塞状态，直到到达超时时间或者在超时时间内有可用的证书（被其他线程释放的证书），或者阻塞中的线程被其他线程执行了中断。

```
final Semaphore semaphore = new Semaphore(1, true);
// 定义一个线程
new Thread(() -> {
    // 获取许可证
    boolean gotPermit = semaphore.tryAcquire();
    // 如果获取成功就休眠5秒的时间
    if (gotPermit) {
        try {
            System.out.println(Thread.currentThread() + " get one permit.");
            TimeUnit.SECONDS.sleep(5);
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            // 释放Semaphore的许可证
            semaphore.release();
        }
    }
}).start();
// 短暂休眠1秒的时间，确保上面的线程能够启动，并且顺利获取许可证
TimeUnit.SECONDS.sleep(1);
// 主线程在3秒之内肯定是无法获取许可证的，那么主线程将在阻塞3秒之后返回获取许可证失败
if(semaphore.tryAcquire(3, TimeUnit.SECONDS)){
    System.out.println("get the permit");
}else {
    System.out.println("get the permit failure.");
}
```

- boolean **tryAcquire(int permits)**：在使用无参的tryAcquire时只会向Semaphore尝试获取一个许可证，但是该方法会向Semaphore尝试获取指定数目的许可证。

```
// 定义许可证数量为5的Semaphore
final Semaphore semaphore = new Semaphore(5, true);
// 尝试获取5个许可证，成功
assert semaphore.tryAcquire(5) : "acquire permit successfully.";
// 此时Semaphore中已经没有可用的许可证了，尝试获取将会失败
assert !semaphore.tryAcquire() : "acquire permit failure.";
```

- boolean **tryAcquire(int permits, long timeout, TimeUnit unit)** throws InterruptedException：该方法与第二个方法类似，只不过其可以指定尝试获取许可证数量的参数。

- **正确使用release**

在一个Semaphore中，许可证的数量可用于控制在同一时间允许多少个线程对共享资源进行访问，所以许可证的数量是非常珍贵的。因此当每一个线程结束对Semaphore许可证的使用之后应该立即将其释放，允许其他线程有机会争抢许可证，下面是Semaphore提供的许可证释放方法。

- void **release()**：释放一个许可证，并且在Semaphore的内部，可用许可证的计数器会随之加一，表明当前有一个新的许可证可被使用。
- void **release(int permits)**：释放指定数量（permits）的许可证，并且在Semaphore内部，可用许可证的计数器会随之增加permits个，表明当前又有permits个许可证可被使用。

release方法非常简单，是吧？但是该方法往往是很多程序员容易出错的地方，而且一旦出现错误在系统运行起来之后，排查是比较困难的，为了确保能够释放已经获取到的许可证，我们的第一反应是将其放到try...finally...语句块中，这样无论在任何情况下都能确保将已获得的许可证释放，但是恰恰是这样的操作会导致对Semaphore的使用不当，我们一起来看一下下面的例子。

```
// 定义只有一个许可证的Semaphore
final Semaphore semaphore = new Semaphore(1, true);
// 创建线程t1
Thread t1 = new Thread(() -> {
    try {
        // 获取Semaphore的许可证
        semaphore.acquire();
        System.out.println("The thread t1 acquired permit from semaphore.");
        // 霸占许可证一个小时
        TimeUnit.HOURS.sleep(1);
    } catch (InterruptedException e) {
        System.out.println("The thread t1 is interrupted");
    } finally {
        // 在finally语句块中释放许可证
        semaphore.release();
    }
});
// 启动线程t1
t1.start();
// 为确保线程t1已经启动，在主线程中休眠1秒稍作等待
TimeUnit.SECONDS.sleep(1);
// 创建线程t2
Thread t2 = new Thread(() -> {
    try {
        // 阻塞式地获取一个许可证
        semaphore.acquire();
        System.out.println("The thread t2 acquired permit from semaphore.");
    } catch (InterruptedException e) {
        System.out.println("The thread t2 is interrupted");
    } finally {
        // 同样在finally语句块中释放已经获取的许可证
        semaphore.release();
    }
});
// 启动线程t2
t2.start();
// 休眠2秒后
TimeUnit.SECONDS.sleep(2);
```

```
39 // 对线程t2执行中断操作
40 t2.interrupt();
41 // 主线程获取许可证
42 semaphore.acquire();
43 System.out.println("The main thread acquired permit.");
```

根据我们的期望，无论线程t2是被中断还是在阻塞中，主线程都不应该成功获取到许可证，但是由于我们对release方法的错误使用，导致了主线程成功获取了许可证，运行上述代码会看到如下的输出结果：

为什么会这样？就是finally语句块导致的问题，当线程t2被其他线程中断或者因自身原因出现异常的时候，它释放了原本不属于自己的许可证，导致在Semaphore内部的可用许可证计数器增多，其他线程才有机会获取到原本不该属于它的许可证。
这难道是Semaphore的设计缺陷？其实并不是，打开Semaphore的官方文档，其中对release方法的描述如下：“There is no requirement that a thread that releases a permit must have acquired that permit by calling acquire(). Correct usage of a semaphore is established by programming convention in the application.”由此可以看出，设计并未强制要求执行release操作的线程必须是执行了acquire的线程才可以，而是需要开发人员自身具有相应的编程约束来确保Semaphore的正确使用，不管怎样，我们对上面的代码稍作修改，具体如下。

```
...省略
Thread t2 = new Thread(() ->
{
    try{
        // 获取许可证
        semaphore.acquire();
    } catch (InterruptedException e){
        System.out.println("The thread t2 is interrupted");
        // 若出现异常则不再往下进行
        return;
    }
    // 程序运行到此处，说明已经成功获取了许可证，因此在finally语句块中对其进行释放就是理所当然的了
    try
    {
        System.out.println("The thread t2 acquired permit from semaphore.");
    } finally{
        semaphore.release();
    }
});
t2.start();
...省略
```

程序修改之后再次运行，当线程t2被中断之后，它就无法再进行许可证的释放操作了，因此主线程也将不会再意外获取到许可证，这种方式是确保能够解决许可证被正确释放的思路之一。

2.2 Semaphore使用

Semaphore实现商品服务接口限流

Semaphore可以用于实现限流功能，即限制某个操作或资源在一定时间内的访问次数。

```java
@Slf4j
public class SemaphoreDemo {

    /**
     * 同一时刻最多只允许有两个并发
     */
    private static Semaphore semaphore = new Semaphore(2);

    private static Executor executor = Executors.newFixedThreadPool(10);

    public static void main(String[] args) {
        for(int i=0;i<10;i++){
            executor.execute(()->getProductInfo2());
        }
    }

    public static String getProductInfo() {
        try {
            semaphore.acquire();
            log.info("请求服务");
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }finally {
            semaphore.release();
        }
        return "返回商品详情信息";
    }

    public static String getProductInfo2() {

        if(!semaphore.tryAcquire()){
            log.error("请求被流控了");
            return "请求被流控了";
        }
        try {
            log.info("请求服务");
            Thread.sleep(2000);
```

```
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }finally {
            semaphore.release();
        }
        return "返回商品详情信息";
    }
}
```

Semaphore限制同时在线的用户数量

我们模拟某个登录系统，最多限制给定数量的人员同时在线，如果所能申请的许可证不足，那么将告诉用户无法登录，稍后重试。

```
public class SemaphoreDemo7 {
    public static void main(String[] args) {
        // 定义许可证数量，最多同时只能有10个用户登录成功并且在线
        final int MAX_PERMIT_LOGIN_ACCOUNT = 10;

        final LoginService loginService = new LoginService(MAX_PERMIT_LOGIN_ACCOUNT);

        // 启动20个线程
        IntStream.range(0, 20).forEach(i -> new Thread(() -> {
            // 登录系统，实际上是一次许可证的获取操作
            boolean login = loginService.login();
            // 如果登录失败，则不再进行其他操作
            if (!login) {
                //超过最大在线用户数就会拒绝
                System.out.println(currentThread() + " is refused due to exceed max online account.");
                return;
            }

            try {
                // 简单模拟登录成功后的系统操作
                simulateWork();
            } finally {
                // 退出系统，实际上是对许可证资源的释放
                loginService.logout();
            }
        }, "User-" + i).start());
    }

    // 随机休眠
    private static void simulateWork() {
        try {
            TimeUnit.SECONDS.sleep(current().nextInt(10));
        } catch (InterruptedException e) {
            // ignore
        }
    }

    private static class LoginService {
```

```
39          private final Semaphore semaphore;
40
41          public LoginService(int maxPermitLoginAccount) {
42              // 初始化Semaphore
43              this.semaphore = new Semaphore(maxPermitLoginAccount, true);
44          }
45
46          public boolean login() {
47              // 获取许可证，如果获取失败该方法会返回false，tryAcquire不是一个阻塞方法
48              boolean login = semaphore.tryAcquire();
49              if (login) {
50                  System.out.println(currentThread() + " login success.");
51              }
52              return login;
53          }
54
55          // 释放许可证
56          public void logout() {
57              semaphore.release();
58              System.out.println(currentThread() + " logout success.");
59          }
60      }
61  }
```

在上面的代码中，我们定义了Semaphore的许可证数量为10，这就意味着当前的系统最多只能有10个用户同时在线，如果其他线程在Semaphore许可证数量为0的时候尝试申请，就将会出现申请不成功的情况。

如果将tryAcquire方法修改为阻塞方法acquire，那么我们会看到所有的未登录成功的用户在其他用户退出系统后会陆陆续续登录成功（修改后的login方法）。

```java
public boolean login()
{
    try
    {
        // acquire为阻塞方法，会一直等待有可用的许可证并且获取之后才会退出阻塞
        semaphore.acquire();
        System.out.println(currentThread() + " login success.");
    } catch (InterruptedException e)
    {
        // 在阻塞过程中有可能被其他线程中断
        return false;
    }
    return true;
}
```

2.3 应用场景总结

Semaphore（信号量）是一个非常好的高并发工具类，它允许最多可以有多少个线程同时对共享数据进行访问。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，可以用于控制一个或多个线程等待多个任务完成后再执行。当某项工作需要由若干项子任务并行地完成，并且只有在所有的子任务结束之后（正常结束或者异常结束），当前主任务才能进入下一阶段，CountDownLatch工具将是非常好用的工具。

CountDownLatch 内部维护了一个计数器，该计数器初始值为 N，代表需要等待的线程数目，当一个线程完成了需要等待的任务后，就会调用 countDown() 方法将计数器减 1，当计数器的值为 0 时，等待的线程就会开始执行。

3.1 常用API

构造器

常用方法

```
 // 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
public void await() throws InterruptedException { };
// 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
public boolean await(long timeout, TimeUnit unit) throws InterruptedException { };
// 会将 count 减 1，直至为 0
public void countDown() { };
```

CountDownLatch的其他方法及总结：

- CountDownLatch的构造非常简单，需要给定一个不能小于0的int数字。
- countDown()方法，该方法的主要作用是使得构造CountDownLatch指定的count计数器减一。如果此时CountDownLatch中的计数器已经是0，这种情况下如果再次调用countDown()方法，则会被忽略，也就是说count的值最小只能为0。
- await()方法会使得当前的调用线程进入阻塞状态，直到count为0，当然其他线程可以将当前线程中断。同样，当count的值为0的时候，调用await方法将会立即返回，当前线程将不再被阻塞。
- await（long timeout, TimeUnit unit）是一个具备超时能力的阻塞方法，当时间达到给定的值以后，计数器count的值若还大于0，则当前线程会退出阻塞。
- getCount()方法，该方法将返回CountDownLatch当前的计数器数值，该返回值的最小值为0。

示例：

```
// 定义一个计数器为2的Latch
CountDownLatch latch = new CountDownLatch(2);
// 调用countDown方法，此时count=1
latch.countDown();
// 调用countDown方法，此时count=0
latch.countDown();
// 调用countDown方法，此时count仍然为0
latch.countDown();
// count已经为0，那么执行await将会被直接返回，不再进入阻塞
latch.await();
```

3.2 CountDownLatch使用

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。

```
public class CountDownLatchDemo2 {
    public static void main(String[] args) throws Exception {

        CountDownLatch countDownLatch = new CountDownLatch(5);
        for (int i = 0; i < 5; i++) {
            final int index = i;
            new Thread(() -> {
                try {
                    Thread.sleep(1000 + ThreadLocalRandom.current().nextInt(2000));
                    System.out.println("任务" + index +"执行完成");
                    countDownLatch.countDown();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        // 主线程在阻塞，当计数器为0，就唤醒主线程往下执行
        countDownLatch.await();
        System.out.println("主线程:在所有任务运行完成后，进行结果汇总");
    }
}
```

电商场景中的应用——等待所有子任务结束

考虑一下这样一个场景，我们需要调用某个品类的商品，然后针对活动规则、会员等级、商品套餐等计算出陈列在页面的最终价格（这个计算过程可能会比较复杂、耗时较长，因为可能要调用其他系统的接口，比如ERP、CRM等），最后将计算结果统一返回给调用方，如图

假设根据商品品类ID获取到了10件商品，然后分别对这10件商品进行复杂的划价计算，最后统一将结果返回给调用者。想象一下，即使忽略网络调用的开销时间，整个结果最终将耗时T = M（M为获取品类下商品的时间）+10×N（N为计算每一件商品价格的平均时间开销），整个串行化的过程中，总体的耗时还会随着N的数量增多而持续增长。

那么，如果想要提高接口调用的响应速度应该如何操作呢？很明显，将某些串行化的任务并行化处理是一种非常不错的解决方案（这些串行化任务在整体的运行周期中彼此之间互相独立）。改进之后的设计方案将变成如图

经过改进之后，接口响应的最终耗时T = M（M为获取品类下商品的时间）+ Max（N）（N为计算每一件商品价格的开销时间），简单开发程序模拟一下这样的一个场景，代码如下

```
public class CountDownLatchDemo3 {

    /**
     * 根据品类ID获取商品列表
     *
     * @return
     */
    private static int[] getProductsByCategoryId() {
        // 商品列表编号为从1～10的数字
        return IntStream.rangeClosed(1, 10).toArray();
    }

    /*
     *  商品编号与所对应的价格，当然真实的电商系统中不可能仅存在这两个字段
     */
    private static class ProductPrice {
        private final int prodID;
        private double price;

        private ProductPrice(int prodID) {
            this(prodID, -1);
        }

        private ProductPrice(int prodID, double price) {
            this.prodID = prodID;
            this.price = price;
        }

        int getProdID() {
            return prodID;
        }

        void setPrice(double price) {
            this.price = price;
        }

        @Override
        public String toString() {
```

```
            return "ProductPrice{" + "prodID=" + prodID + ", price=" + price + '}';
        }
    }

    public static void main(String[] args) throws InterruptedException {
        // 首先获取商品编号的列表
        final int[] products = getProductsByCategoryId();

        // 通过stream的map运算将商品编号转换为ProductPrice
        List<ProductPrice> list =
Arrays.stream(products).mapToObj(ProductPrice::new).collect(toList());
        //1. 定义CountDownLatch，计数器数量为子任务的个数
        final CountDownLatch latch = new CountDownLatch(products.length);
        list.forEach(pp ->
                // 2. 为每一件商品的计算都开辟对应的线程
                new Thread(() -> {
                    System.out.println(pp.getProdID() + "-> 开始计算商品价格.");
                    try {
                        // 模拟其他的系统调用，比较耗时，这里用休眠替代
                        TimeUnit.SECONDS.sleep(current().nextInt(10));
                        // 计算商品价格
                        if (pp.prodID % 2 == 0) {
                            pp.setPrice(pp.prodID * 0.9D);
                        } else {
                            pp.setPrice(pp.prodID * 0.71D);
                        }
                        System.out.println(pp.getProdID() + "-> 价格计算完成.");
                    } catch (InterruptedException e) {
                        e.printStackTrace();
                    } finally {
                        // 3. 计数器count down，子任务执行完成
                        latch.countDown();
                    }
                }).start());

        // 4.主线程阻塞等待所有子任务结束，如果有一个子任务没有完成则会一直等待
        latch.await();
        System.out.println("所有价格计算完成.");
        list.forEach(System.out::println);
    }
```

```
78
79
80 }
```

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。

CountDownLatch的不足

CountDownLatch是一次性的，计算器的值只能在构造方法中初始化一次，之后没有任何机制再次对其设置值，当CountDownLatch使用完毕后，它不能再次被使用。

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。CyclicBarrier也非常适合用于某个串行化任务被分拆成若干个并行执行的子任务，当所有的子任务都执行结束之后再继续接下来的工作。

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)
```

常用方法

```
//指定数量的线程全部调用await()方法时，这些线程不再阻塞
// 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
public int await() throws InterruptedException, BrokenBarrierException
public int await(long timeout, TimeUnit unit) throws InterruptedException, BrokenBarrierException, TimeoutException

//循环  通过reset()方法可以进行重置
public void reset()
```

4.2 CyclicBarrier使用

等待所有子任务结束

前面CountDownLatch中调用某个品类的商品最终价格的场景同样也可以使用CyclicBarrier实现。

```java
public class CyclicBarrierDemo2 {

    /**
     * 根据品类ID获取商品列表
     *
     * @return
     */
    private static int[] getProductsByCategoryId() {
        // 商品列表编号为从1～10的数字
        return IntStream.rangeClosed(1, 10).toArray();
    }

    /*
     *  商品编号与所对应的价格，当然真实的电商系统中不可能仅存在这两个字段
     */
    private static class ProductPrice {
        private final int prodID;
        private double price;

        private ProductPrice(int prodID) {
            this(prodID, -1);
        }

        private ProductPrice(int prodID, double price) {
            this.prodID = prodID;
            this.price = price;
        }

        int getProdID() {
            return prodID;
        }

        void setPrice(double price) {
            this.price = price;
        }

        @Override
        public String toString() {
```

```
39              return "ProductPrice{" + "prodID=" + prodID + ", price=" + price + '}';
40          }
41      }
42
43
44      public static void main(String[] args) throws InterruptedException {
45          // 根据商品品类获取一组商品ID
46          final int[] products = getProductsByCategoryId();
47          // 通过转换将商品编号转换为ProductPrice
48          List<ProductPrice> list =
    Arrays.stream(products).mapToObj(ProductPrice::new).collect(toList());
49          // 1. 定义CyclicBarrier ， 指定parties为子任务数量
50          final CyclicBarrier barrier = new CyclicBarrier(list.size());
51          // 2.用于存放线程任务的list
52          final List<Thread> threadList = new ArrayList<>();
53          list.forEach(pp -> {
54              Thread thread = new Thread(() -> {
55                  System.out.println(pp.getProdID() + "开始计算商品价格.");
56                  try {
57                      TimeUnit.SECONDS.sleep(current().nextInt(10));
58                      if (pp.prodID % 2 == 0) {
59                          pp.setPrice(pp.prodID * 0.9D);
60                      } else {
61                          pp.setPrice(pp.prodID * 0.71D);
62                      }
63                      System.out.println(pp.getProdID() + "->价格计算完成.");
64                  } catch (InterruptedException e) {
65                      // ignore exception
66                  } finally {
67                      try {
68                          // 3.在此等待其他子线程到达barrier point
69                          barrier.await();
70                      } catch (InterruptedException | BrokenBarrierException e) {
71                      }
72                  }
73              });
74              threadList.add(thread);
75              thread.start();
76          });
77          // 4. 等待所有子任务线程结束
```

```
        threadList.forEach(t -> {
            try {
                t.join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        });
        System.out.println("所有价格计算完成.");
        list.forEach(System.out::println);
    }
}
```

CyclicBarrier的循环特性——模拟跟团旅游

只有在所有的旅客都上了大巴之后司机才能将车开到下一个旅游景点，当大巴到达旅游景点之后，导游还会进行人数清点以确认车上没有旅客由于睡觉而逗留，车才能开去停车场，进而旅客在该景点游玩。

```
public class CyclicBarrierDemo3 {
    public static void main(String[] args)
            throws BrokenBarrierException, InterruptedException {
        // 定义CyclicBarrier，注意这里的parties值为11
        final CyclicBarrier barrier = new CyclicBarrier(11);
        // 创建10个线程
        for (int i = 0; i < 10; i++) {
            // 定义游客线程，传入游客编号和barrier
            new Thread(new Tourist(i, barrier)).start();
        }
        // 主线程也进入阻塞，等待所有游客都上了旅游大巴
        barrier.await();
        System.out.println("导游:所有的游客都上了车.");
        // 主线程进入阻塞，等待所有游客都下了旅游大巴
        barrier.await();
        System.out.println("导游:所有的游客都下车了.");
    }

    private static class Tourist implements Runnable {
        private final int touristID;
        private final CyclicBarrier barrier;

        private Tourist(int touristID, CyclicBarrier barrier) {
            this.touristID = touristID;
            this.barrier = barrier;
        }

        @Override
        public void run() {
            System.out.printf("游客:%d 乘坐旅游大巴\n", touristID);
            // 模拟乘客上车的时间开销
            this.spendSeveralSeconds();
            // 上车后等待其他同伴上车
            this.waitAndPrint("游客:%d 上车，等别人上车.\n");
            System.out.printf("游客:%d 到达目的地\n", touristID);
            // 模拟乘客下车的时间开销
            this.spendSeveralSeconds();
            // 下车后稍作等待，等待其他同伴全部下车
```

```
39              this.waitAndPrint("游客:%d 下车，等别人下车.\n");
40          }
41
42          private void waitAndPrint(String message) {
43              System.out.printf(message, touristID);
44              try {
45                  barrier.await();
46              } catch (InterruptedException | BrokenBarrierException e) {
47                  // ignore
48              }
49          }
50
51          // random sleep
52          private void spendSeveralSeconds() {
53              try {
54                  TimeUnit.SECONDS.sleep(current().nextInt(10));
55              } catch (InterruptedException e) {
56                  // ignore
57              }
58          }
59      }
60  }
```

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
- CoundDownLatch的await方法会等待计数器被count down到0，而执行CyclicBarrier的await方法的线程将会等待其他线程到达barrier point。
- CyclicBarrier内部的计数器count是可被重置的，进而使得CyclicBarrier也可被重复使用，而CoundDownLatch则不能

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。

```java
public class ExchangerDemo {
    private static Exchanger exchanger = new Exchanger();
    static String goods = "电脑";
    static String money = "$4000";
    public static void main(String[] args) throws InterruptedException {

        System.out.println("准备交易，一手交钱一手交货...");
        // 卖家
        new Thread(new Runnable() {
            @Override
            public void run() {
                System.out.println("卖家到了，已经准备好货：" + goods);
                try {
                    String money = (String) exchanger.exchange(goods);
                    System.out.println("卖家收到钱：" + money);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();

        Thread.sleep(3000);

        // 买家
        new Thread(new Runnable() {
            @Override
            public void run() {
                try {
                    System.out.println("买家到了，已经准备好钱：" + money);
                    String goods = (String) exchanger.exchange(money);
                    System.out.println("买家收到货：" + goods);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }).start();

    }
```

```
39 }
```

模拟对账场景

```java
public class ExchangerDemo2 {

    private static final Exchanger<String> exchanger = new Exchanger();
    private static ExecutorService threadPool = Executors.newFixedThreadPool(2);

    public static void main(String[] args) {

        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    String A = "12379871924sfkhfksdhfks";
                    exchanger.exchange(A);
                } catch (InterruptedException e) {
                }
            }
        });

        threadPool.execute(new Runnable() {
            @Override
            public void run() {
                try {
                    String B = "32423423jknjkfsbfj";
                    String A = exchanger.exchange(B);
                    System.out.println("A和B数据是否一致: " + A.equals(B));
                    System.out.println("A= "+A);
                    System.out.println("B= "+B);
                } catch (InterruptedException e) {
                }
            }
        });

        threadPool.shutdown();

    }
}
```

模拟队列中交换数据场景

```java
public class ExchangerDemo3 {

    private static ArrayBlockingQueue<String> fullQueue
            = new ArrayBlockingQueue<>(5);
    private static ArrayBlockingQueue<String> emptyQueue
            = new ArrayBlockingQueue<>(5);
    private static Exchanger<ArrayBlockingQueue<String>> exchanger
            = new Exchanger<>();


    public static void main(String[] args) {
        new Thread(new Producer()).start();
        new Thread(new Consumer()).start();

    }

    /**
     * 生产者
     */
    static class Producer implements Runnable {
        @Override
        public void run() {
            ArrayBlockingQueue<String> current = emptyQueue;
            try {
                while (current != null) {
                    String str = UUID.randomUUID().toString();
                    try {
                        current.add(str);
                        System.out.println("producer: 生产了一个序列: " + str + ">>>>>加入到交换区");
                        Thread.sleep(2000);
                    } catch (IllegalStateException e) {
                        System.out.println("producer: 队列已满，换一个空的");
                        current = exchanger.exchange(current);
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
```

```
            }
        }
    }

    /**
     * 消费者
     */
    static class Consumer implements Runnable {
        @Override
        public void run() {
            ArrayBlockingQueue<String> current = fullQueue;
            try {
                while (current != null) {
                    if (!current.isEmpty()) {
                        String str = current.poll();
                        System.out.println("consumer: 消耗一个序列: " + str);
                        Thread.sleep(1000);
                    } else {
                        System.out.println("consumer: 队列空了，换个满的");
                        current = exchanger.exchange(current);
                        System.out.println("consumer: 换满的成功
~~~~~~~~~~~~~~~~~~~~~~");
                    }
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }


}

```

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

阶段性任务：模拟公司团建

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

                }

                // 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
                return registeredParties == 1;
            }
        };

        // 注册主线程 —— 让主线程全程参与
        phaser.register();
        final StaffTask staffTask = new StaffTask();

        // 3个全程参与团建的员工
        for (int i = 0; i < 3; i++) {
            // 添加任务数
            phaser.register();
            new Thread(() -> {
                try {
                    staffTask.step1Task();
```

```
                    //到达后等待其他任务到达
                    phaser.arriveAndAwaitAdvance();

                    staffTask.step2Task();
                    phaser.arriveAndAwaitAdvance();

                    staffTask.step3Task();
                    phaser.arriveAndAwaitAdvance();

                    staffTask.step4Task();
                    // 完成了，注销离开
                    phaser.arriveAndDeregister();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        // 两个不聚餐的员工加入
        for (int i = 0; i < 2; i++) {
            phaser.register();
            new Thread(() -> {
                try {
                    staffTask.step1Task();
                    phaser.arriveAndAwaitAdvance();

                    staffTask.step2Task();
                    System.out.println("员工【" + Thread.currentThread().getName() + "】回家了");
                    // 完成了，注销离开
                    phaser.arriveAndDeregister();
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }
            }).start();
        }

        while (!phaser.isTerminated()) {
            int phase = phaser.arriveAndAwaitAdvance();
```

```
            if (phase == 2) {
                // 到了去餐厅的阶段，又新增4人，参加晚上的聚餐
                for (int i = 0; i < 4; i++) {
                    phaser.register();
                    new Thread(() -> {
                        try {
                            staffTask.step3Task();
                            phaser.arriveAndAwaitAdvance();

                            staffTask.step4Task();
                            // 完成了，注销离开
                            phaser.arriveAndDeregister();
                        } catch (InterruptedException e) {
                            e.printStackTrace();
                        }
                    }).start();
                }
            }
        }
    }

    static final Random random = new Random();

    static class StaffTask {
        public void step1Task() throws InterruptedException {
            // 第一阶段：来公司集合
            String staff = "员工【" + Thread.currentThread().getName() + "】";
            System.out.println(staff + "从家出发了……");
            Thread.sleep(random.nextInt(5000));
            System.out.println(staff + "到达公司");
        }

        public void step2Task() throws InterruptedException {
            // 第二阶段：出发去公园
            String staff = "员工【" + Thread.currentThread().getName() + "】";
            System.out.println(staff + "出发去公园玩");
            Thread.sleep(random.nextInt(5000));
            System.out.println(staff + "到达公园门口集合");

```

```
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。