<https://mp.weixin.qq.com/s?__biz=MzA5MTkxMDQ4MQ==&mid=2648933144&idx=1&sn=7f0cddc92ff39835ea6652ebb3186dbf&chksm=88621b26bf15923039933b127c19f39a76214fb1d5daa7ad0eee77f961e2e3ab5f5ca3f48740&token=773938509&lang=zh_CN&scene=21#wechat_redirect>

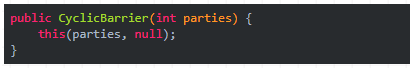
# JUC中的循环栅栏CyclicBarrier的6种使用场景

## 本文主要内容

* 介绍CyclicBarrier
* 6个示例介绍CyclicBarrier的使用
* 对比CyclicBarrier和CountDownLatch

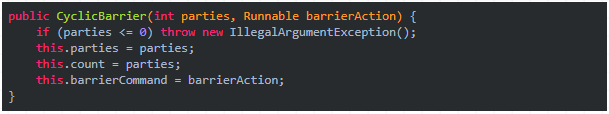
## CyclicBarrier简介

CyclicBarrier通常称为循环屏障。它和CountDownLatch很相似，都可以使线程先等待然后再执行。不过CountDownLatch是使一批线程等待另一批线程执行完后再执行；而CyclicBarrier只是使等待的线程达到一定数目后再让它们继续执行。故而CyclicBarrier内部也有一个计数器,计数器的初始值在创建对象时通过构造参数指定,如下所示：



每调用一次await()方法都将使阻塞的线程数+1，只有阻塞的线程数达到设定值时屏障才会打开，允许阻塞的所有线程继续执行。除此之外，CyclicBarrier还有几点需要注意的地方:

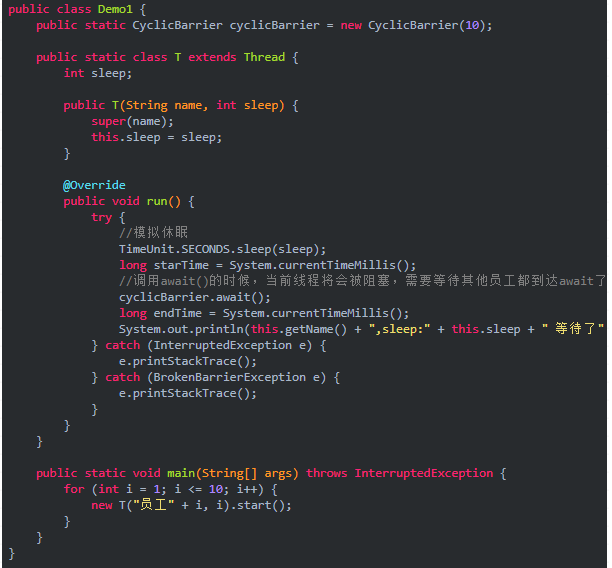
* CyclicBarrier的计数器可以重置而CountDownLatch不行，这意味着CyclicBarrier实例可以被重复使用而CountDownLatch只能被使用一次。而这也是循环屏障循环二字的语义所在。
* CyclicBarrier允许用户自定义barrierAction操作，这是个可选操作，可以在创建CyclicBarrier对象时指定



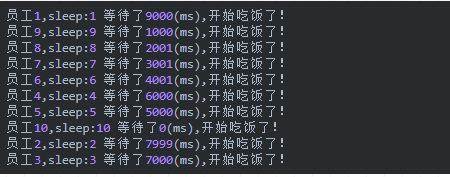
一旦用户在创建CyclicBarrier对象时设置了barrierAction参数，则在阻塞线程数达到设定值屏障打开前，会调用barrierAction的run()方法完成用户自定义的操作。

## 示例1：简单使用CyclicBarrier

公司组织旅游，大家都有经历过，10个人，中午到饭点了，需要等到10个人都到了才能开饭，先到的人坐那等着，代码如下：



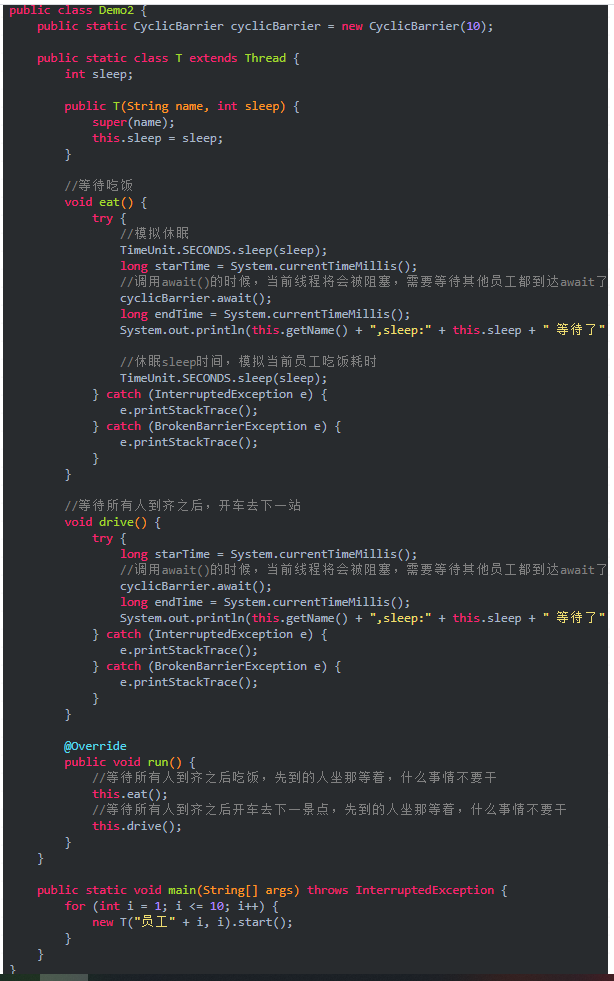
输出：



代码中模拟了10个员工上桌吃饭的场景，等待所有员工都到齐了才能开发，可以看到第10个员工最慢，前面的都在等待第10个员工，员工1等待了9秒，上面代码中调用cyclicBarrier.await();会让当前线程等待。当10个员工都调用了cyclicBarrier.await();之后，所有处于等待中的员工都会被唤醒，然后继续运行。

## 示例2：重复使用CyclicBarrier

对示例1进行改造一下，吃饭完毕之后，所有人都去车上，待所有人都到车上之后，驱车去下一景点玩。



坑，又是员工10最慢，要提升效率了，不能吃的太多，得减肥。

代码中CyclicBarrier相当于使用了2次，第一次用于等待所有人到达后开饭，第二次用于等待所有人上车后驱车去下一景点。注意一些先到的员工会在其他人到达之前，都处于等待状态（cyclicBarrier.await();会让当前线程阻塞），无法干其他事情，等到最后一个人到了会唤醒所有人，然后继续。

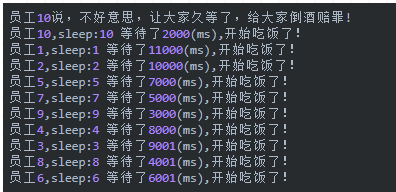
CyclicBarrier内部相当于有个计数器（构造方法传入的），每次调用await();后，计数器会减1，并且await()方法会让当前线程阻塞，等待计数器减为0的时候，所有在await()上等待的线程被唤醒，然后继续向下执行，此时计数器又会被还原为创建时的值，然后可以继续再次使用。

## 示例3：最后到的人给大家上酒，然后开饭

还是示例1中的例子，员工10是最后到达的，让所有人都久等了，那怎么办，得给所有人倒酒，然后开饭，代码如下：



输出：



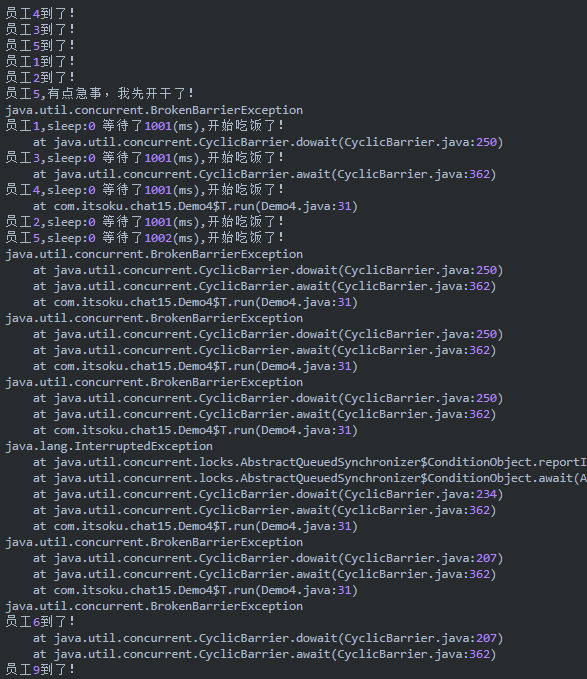
代码中创建CyclicBarrier对象时，多传入了一个参数（内部是倒酒操作），先到的人先等待，待所有人都到齐之后，需要先给大家倒酒，然后唤醒所有等待中的人让大家开饭。从输出结果中我们发现，倒酒操作是由最后一个人操作的，最后一个人倒酒完毕之后，才唤醒所有等待中的其他员工，让大家开饭。

## 示例4：其中一个人等待中被打断了

员工5等待中，突然接了个电话，有点急事，然后就拿起筷子开吃了，其他人会怎么样呢？看着他吃么？代码如下：



输出：





输出的信息看着有点乱，给大家理一理，员工5遇到急事，拿起筷子就是吃，这样好么，当然不好，他这么做了，后面看他这么做了都跟着这么做（这种场景是不是很熟悉，有一个人拿起筷子先吃起来，其他人都跟着上了），直接不等其他人了，拿起筷子就开吃了。CyclicBarrier遇到这种情况就是这么处理的。前面4个员工都在await()处等待着，员工5也在await()上等待着，等了1秒（TimeUnit.SECONDS.sleep(1);），接了个电话，然后给员工5发送中断信号后（t.interrupt();），员工5的await()方法会触发InterruptedException异常，此时其他等待中的前4个员工，看着5开吃了，自己立即也不等了，内部从await()方法中触发BrokenBarrierException异常，然后也开吃了，后面的6/7/8/9/10员工来了以后发现大家都开吃了，自己也不等了，6-10员工调用await()直接抛出了BrokenBarrierException异常，然后继续向下。

**结论：**

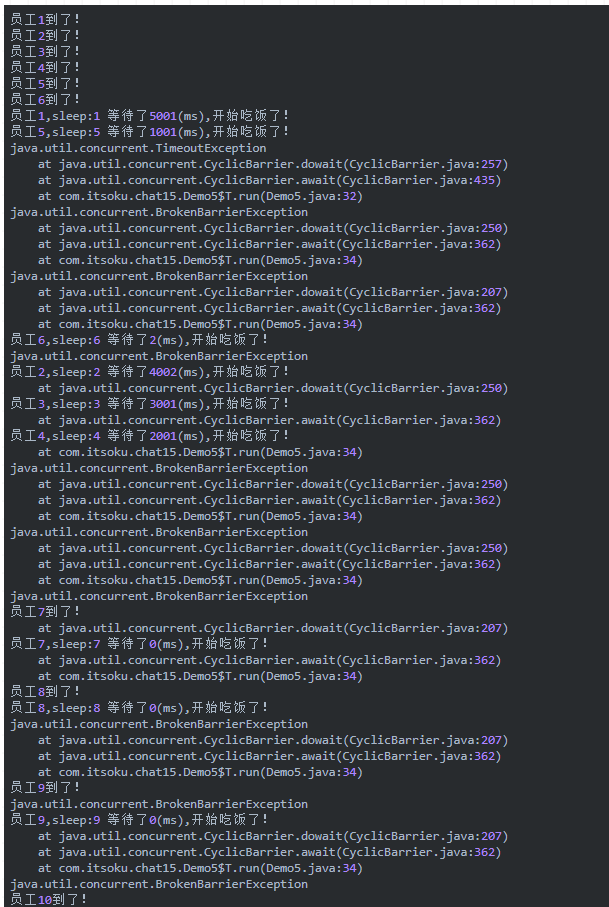
1. 内部有一个人把规则破坏了（接收到中断信号），其他人都不按规则来了，不会等待了
2. 接收到中断信号的线程，await方法会触发InterruptedException异常，然后被唤醒向下运行
3. 其他等待中 或者后面到达的线程，会在await()方法上触发`BrokenBarrierException`异常，然后继续执行

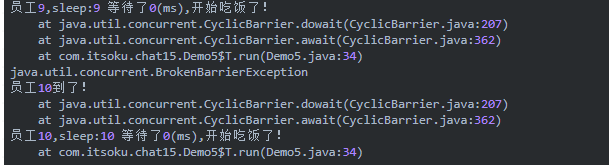
## 示例5：其中一个人只愿意等待5秒

基于示例1，员工1只愿意等的5秒，5s后如果大家还没到期，自己要开吃了，员工1开吃了，其他人会怎么样呢？



输出：





从输出结果中我们可以看到：1等待5秒之后，开吃了，其他等待人都开吃了，后面来的人不等待，直接开吃了。

员工1调用有参await方法等待5秒之后，触发了TimeoutException异常，然后继续向下运行，其他的在5开吃之前已经等了一会的的几个员工，他们看到5开吃了，自己立即不等待了，也开吃了（他们的await抛出了BrokenBarrierException异常）；还有几个员工在5开吃之后到达的，他们直接不等待了，直接抛出BrokenBarrierException异常，然后也开吃了。

**结论：**

* 等待超时的方法：

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

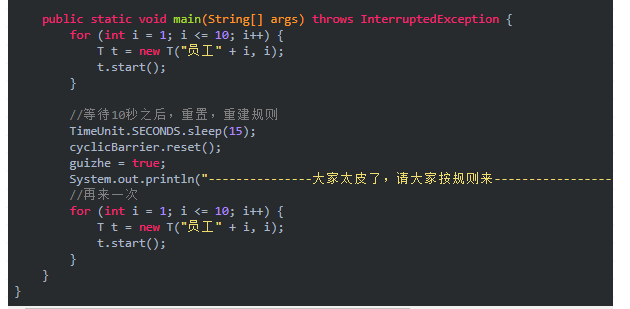
* 内部有一个人把规则破坏了（等待超时），其他人都不按规则来了，不会等待了
* 等待超时的线程，await方法会触发TimeoutException异常，然后被唤醒向下运行
* 其他等待中或者后面到达的线程，会在await()方法上触发`BrokenBarrierException`异常，然后继续执行

## 示例6：重建规则

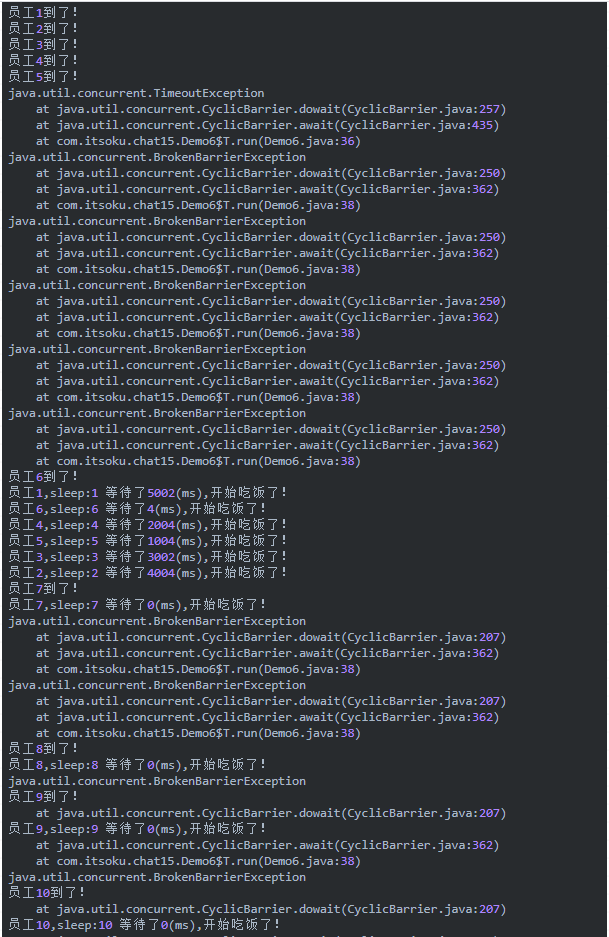
示例5中改造一下，员工1等待5秒超时之后，开吃了，打破了规则，先前等待中的以及后面到达的都不按规则来了，都拿起筷子开吃。过了一会，导游重新告知大家，要按规则来，然后重建了规则，大家都按规则来了。

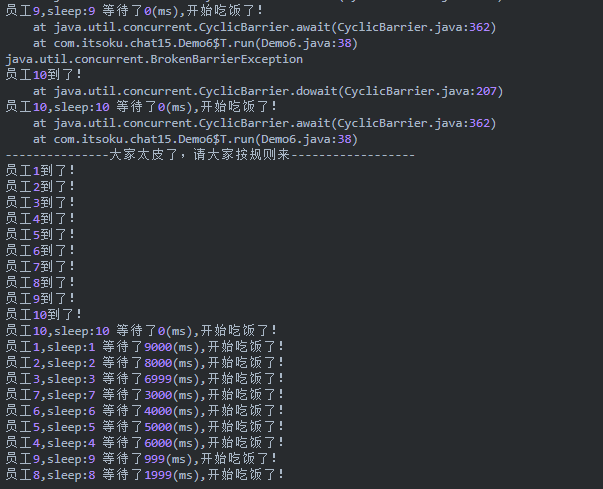
代码如下：





输出：





第一次规则被打乱了，过了一会导游重建了规则（cyclicBarrier.reset();），接着又重来了一次模拟等待吃饭的操作，正常了。

## CountDownLatch和CyclicBarrier的区别

还是举例子说明一下：

主管相当于 CountDownLatch，干活的小弟相当于做事情的线程。老板交给主管了一个任务，让主管搞完之后立即上报给老板。主管下面有10个小弟，接到任务之后将任务划分为10个小任务分给每个小弟去干，主管一直处于等待状态（主管会调用await()方法，此方法会阻塞当前线程），让每个小弟干完之后通知一下主管（调用countDown()方法通知主管，此方法会立即返回），主管等到所有的小弟都做完了，会被唤醒，从await()方法上苏醒，然后将结果反馈给老板。期间主管会等待，会等待所有小弟将结果汇报给自己。

而CyclicBarrier是一批线程让自己等待，等待所有的线程都准备好了，自己才能继续。