**Day01**

# 总体安排

Day01：多线程基础篇

Day02：多线程高级篇

Day03：多线程实际应用（秒杀实现）

# 课程目标

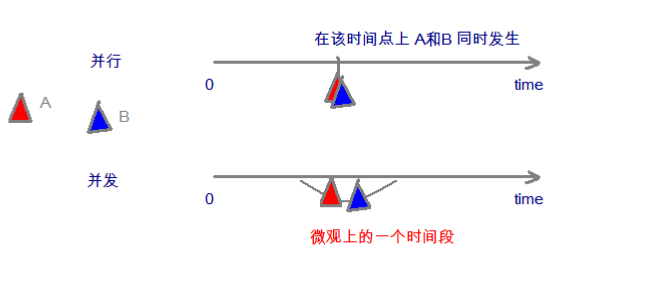
* 什么是并发与并行
* 什么是进程与线程
* 线程创建
* 线程生命周期
* 线程安全问题
* 什么是线程安全问题
* 线程安全问题解决方案
* 线程死锁
* 死锁必要条件
* 如何避免死锁
* 线程通讯

# 什么是并发与并行

要想学习多线程，必须先理解什么是并发与并行

并行：指两个或多个事件在**同一时刻**发生（同时发生）。

并发：指两个或多个事件在**同一个时间段**内发生。



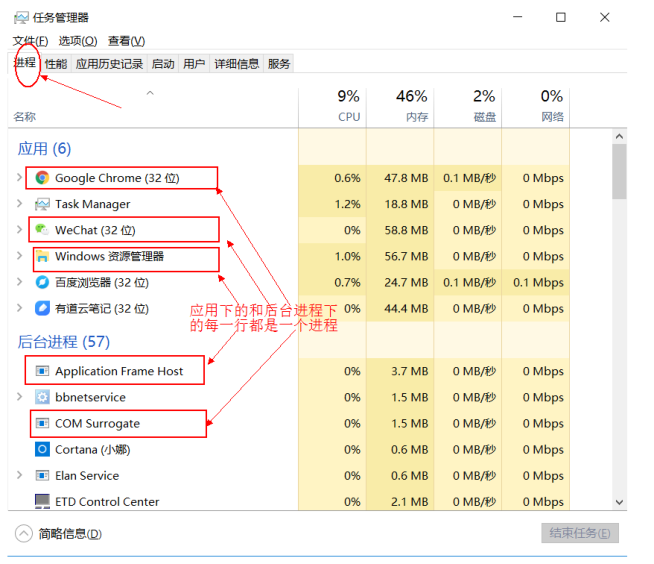
# 什么是进程、线程

进程：

进程是正在运行的程序的实例。

进程是线程的容器，即一个进程中可以开启多个线程。

比如打开一个浏览器、打开一个word等操作，都会创建进程。

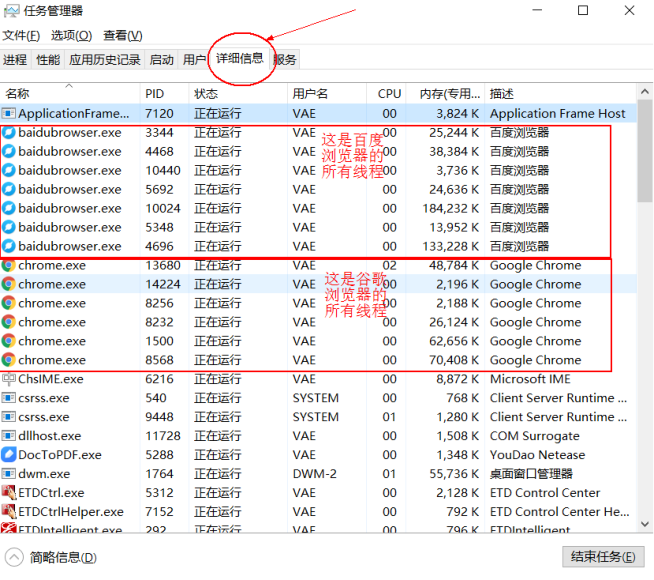


线程：

线程是进程内部的一个独立执行单元；

一个进程可以同时并发运行多个线程；

比如进程可以理解为医院，线程是挂号、就诊、缴费、拿药等业务活动



多线程：多个线程并发执行。

# 线程创建

Java中线程有四种创建方式：

* 继承Thread类
* 实现Runnable接口
* 实现Callable接口
* 线程池

## 继承Thread类

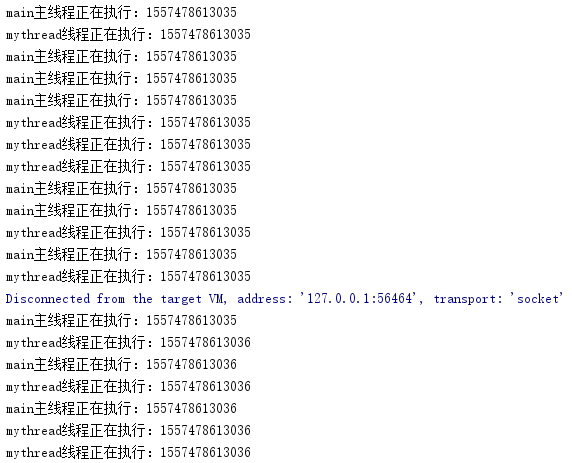
### 第一步：创建自定义线程类

**package** com.multithread.thread;  
  
**import** java.util.Date;  
  
**public class** MyThread **extends** Thread {  
 @Override  
 **public void** run() {  
 **for** (**int** i = 0; i<10; i++){  
 System.***out***.println(**"mythread线程正在执行："**+**new** Date().getTime());  
 }  
 }  
}

### 第二步：创建测试类

**package** com.multithread.demos;  
  
**import** com.multithread.thread.MyThread;  
  
**import** java.util.Date;  
  
**public class** ThreadCreateDemo {  
 **public static void** main(String[] args){  
 *//1.创建自定义线程* MyThread thread = **new** MyThread();  
 thread.start();  
 *//2.主线程循环打印* **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"main主线程正在执行："**+**new** Date().getTime());  
 }  
 }  
}

执行效果如下：



## 实现Runnable接口

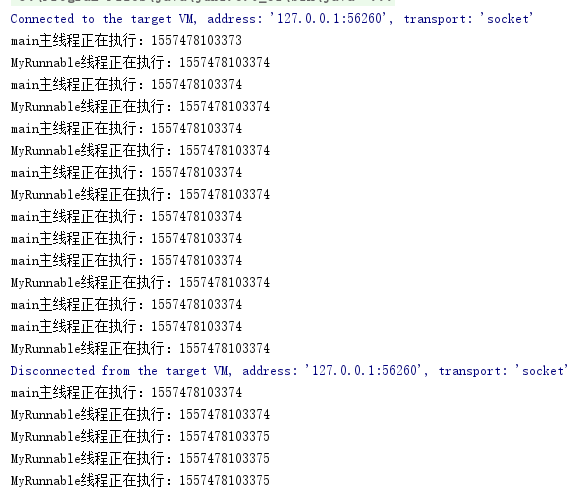
### 第一步：创建自定义类实现Runnable接口

**package** com.multithread.thread;  
  
**import** java.util.Date;  
  
**public class** MyRunable **implements** Runnable {  
 **public void** run() {  
 **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"MyRunnable线程正在执行："**+**new** Date().getTime());  
 }  
 }  
}

### 第二步：创建测试类

**package** com.multithread.demos;  
  
**import** com.multithread.thread.MyRunable;  
**import** com.multithread.thread.MyThread;  
  
**import** java.util.Date;  
  
**public class** ThreadCreateDemo {  
**public static void** main(String[] args){  
 *//1.创建自定义线程（把事项runable接口的类梵高Thread中去执行）* Thread thread = **new** Thread(**new** MyRunable());  
 thread.start();  
 *//2.主线程循环打印* **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"main主线程正在执行："**+**new** Date().getTime());  
 }  
 }  
}

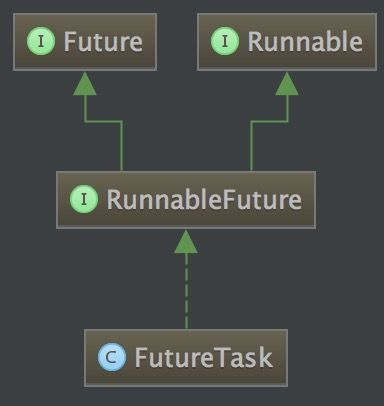
执行效果如下：



## 实现Callable接口

### FutureTask介绍

Callable需要使用FutureTask类帮助执行，FutureTask类结构如下：



Future接口：

判断任务是否完成：isDone()

能够中断任务：cancel()

能够获取任务执行结果：get()

### 第一步：创建自定义类实现Callable接口（Callable带返回值）

**package** com.multithread.thread;  
  
**import** java.util.Date;  
**import** java.util.concurrent.Callable;  
  
**public class** MyCallable **implements** Callable<String> {  
 **public** String call() **throws** Exception {  
 **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"MyCallable正在执行："**+**new** Date().getTime());  
 }  
 **return "MyCallable执行完毕！"**;  
 }  
}

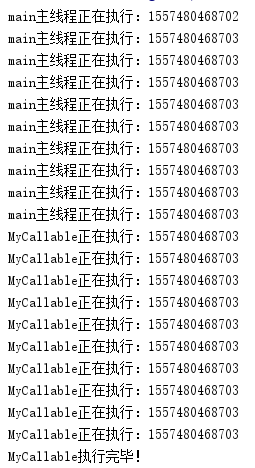
### 第二步：创建测试类

**package** com.multithread.demos;  
  
**import** com.multithread.thread.MyCallable;  
**import** com.multithread.thread.MyRunable;  
**import** com.multithread.thread.MyThread;  
  
**import** java.util.Date;  
**import** java.util.concurrent.ExecutionException;  
**import** java.util.concurrent.FutureTask;  
  
**public class** ThreadCreateDemo {  
**public static void** main(String[] args) **throws** ExecutionException, InterruptedException {

//FutureTask中的泛型与callable中的泛型一致  
 FutureTask<String> task = **new** FutureTask<String>(**new** MyCallable());

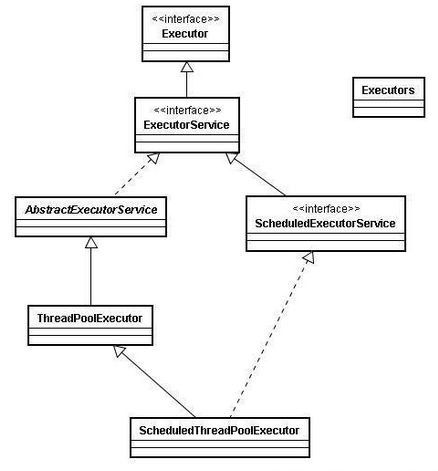
//因为FutrueTast实现了runable接口同理*把实现runable接口的类梵高Thread中去执行*  
 Thread thread = **new** Thread(task);  
 thread.start();  
  
 **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"main主线程正在执行："**+**new** Date().getTime());  
 }  
 System.***out***.println(task.get());  
 }  
}

运行效果图如下：



## 线程池-Executor

### 线程池线类关系图



Executor接口：

声明了execute(Runnable runnable)方法，执行任务代码

ExecutorService接口：

继承Executor接口，声明方法：submit、invokeAll、invokeAny以及shutDown等

AbstractExecutorService抽象类：

实现ExecutorService接口，基本实现ExecutorService中声明的所有方法

ScheduledExecutorService接口：

继承ExecutorService接口，声明定时执行任务方法

ThreadPoolExecutor类：

继承类AbstractExecutorService，实现execute、submit、shutdown、shutdownNow方法

ScheduledThreadPoolExecutor类：

继承ThreadPoolExecutor类，实现ScheduledExecutorService接口并实现其中的方法

Executors类：

提供快速创建线程池的方法

### 第一步：创建自定义类实现Runnable接口

**package** com.multithread.thread;  
  
**import** java.util.Date;  
  
**public class** MyRunable **implements** Runnable {  
 **public void** run() {  
 **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"MyRunnable线程正在执行："**+**new** Date().getTime());  
 }  
 }  
}

### 第二步：创建测试类

**package** com.multithread.demos;  
  
**import** com.multithread.thread.MyCallable;  
**import** com.multithread.thread.MyRunable;  
**import** com.multithread.thread.MyThread;  
  
**import** java.util.Date;  
**import** java.util.concurrent.\*;  
  
**public class** ThreadCreateDemo {  
**public static void** main(String[] args) **throws** ExecutionException, InterruptedException {  
 *//1.使用Executors创建线程池* ExecutorService executorService = Executors.*newFixedThreadPool*(10);  
 *//2.通过线程池执行线程* executorService.execute(**new** MyRunable());  
 *//3.主线程循环打印* **for** (**int** i=0; i<10; i++){  
 System.***out***.println(**"main主线程正在执行："**+**new** Date().getTime());  
 }  
 }  
}

## 小结

### 实现接口和继承Thread类比较

* 接口更适合多个相同的程序代码的线程去共享同一个资源。（意思就是开启多个Thread去执行同一个runable实例）
* 接口可以避免java中的单继承的局限性。
* 接口代码可以被多个线程共享，代码和线程独立。
* 线程池只能放入实现Runable或Callable接口的线程，不能直接放入继承Thread的类。

扩充（面试会问）：

在java中，每次程序运行至少启动2个线程。一个是main线程，一个是垃圾收集线程。

### Runnable和Callable接口比较

相同点：

* 两者都是接口；
* 两者都可用来编写多线程程序；
* 两者都需要调用Thread.start()启动线程；

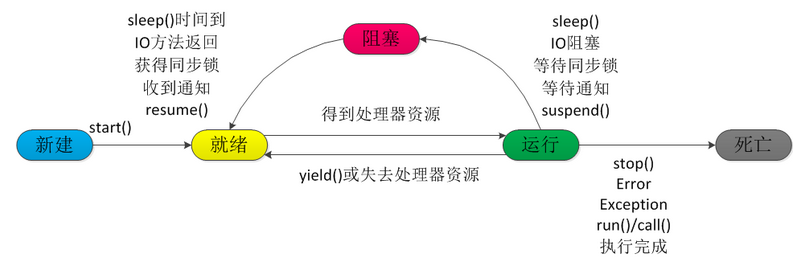
不同点：

* 实现Callable接口的线程能返回执行结果；而实现Runnable接口的线程不能返回结果；
* Callable接口的call()方法允许抛出异常；而Runnable接口的run()方法的不允许抛异常；
* 实现Callable接口的线程可以调用Future.cancel取消执行 （isDone来判断是否执行），而实现Runnable接口的线程不能

注意点：

Callable接口支持返回执行结果，此时需要调用FutureTask.get()方法实现，此方法会阻塞主线程直到获取‘将来’结果；当不调用此方法时，主线程不会阻塞！

# 线程生命周期



## 新建

* new关键字创建了一个线程之后，该线程就处于新建状态
* JVM为线程分配内存，初始化成员变量值

## 就绪

* 当线程对象调用了start()方法之后，该线程处于就绪状态
* JVM为线程创建方法栈和程序计数器，等待线程调度器调度

## 运行

* 就绪状态的线程获得CPU资源，开始运行run()方法，该线程进入运行状态

## 阻塞

当发生如下情况时，线程将会进入阻塞状态

* 线程调用sleep()方法主动放弃所占用的处理器资源
* 线程调用了一个阻塞式IO方法，在该方法返回之前，该线程被阻塞
* 线程试图获得一个同步锁（同步监视器），但该同步锁正被其他线程所持有。
* 线程在等待某个通知（notify）
* 程序调用了线程的suspend()方法将该线程挂起。但这个方法容易导致死锁，所以应该尽量避免使用该方法

## 死亡

线程会以如下3种方式结束，结束后就处于死亡状态：

* run()或call()方法执行完成，线程正常结束。
* 线程抛出一个未捕获的Exception或Error。
* 调用该线程stop()方法来结束该线程，该方法容易导致死锁，不推荐使用。

# 线程安全问题

## 什么是线程安全

如果有多个线程同时运行同一个实现了Runnable接口的类，程序每次运行结果和单线程运行的结果是一样的，而且其他的变量的值也和预期的是一样的，就是线程安全的；反之，则是线程不安全的。

## 问题演示

为了演示线程安全问题，我们采用多线程模拟多个窗口同时售卖《孙悟空大战超人》电影票。

### 第一步：创建售票线程类

**package** com.multithread.thread;  
  
**public class** Ticket **implements** Runnable {  
 **private int ticktNum** = 100;  
   
 **public void** run() {  
 **while**(**true**){  
 **if**(**ticktNum** > 0){  
 *//1.模拟出票时间* **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//2.打印进程号和票号，票数减1* String name = Thread.*currentThread*().getName();  
 System.***out***.println(**"线程"**+name+**"售票："**+**ticktNum**--);  
 }  
 }  
 }  
}

### 第二步：创建测试类

**package** com.multithread.demos;  
  
**import** com.multithread.thread.Ticket;  
  
**public class** TicketDemo {  
 **public static void** main(String[] args){  
 Ticket ticket = **new** Ticket();  
 Thread thread1 = **new** Thread(ticket, **"窗口1"**);  
 Thread thread2 = **new** Thread(ticket, **"窗口2"**);  
 Thread thread3 = **new** Thread(ticket, **"窗口3"**);  
  
 thread1.start();  
 thread2.start();  
 thread3.start();  
 }  
}

运行结果如下：



程序出现了两个问题：

1. 相同的票数,比如5这张票被卖了两回。

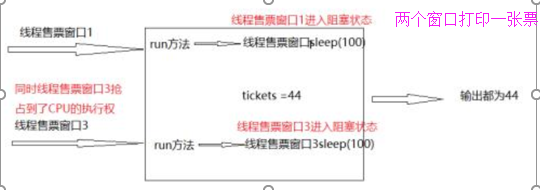
2. 不存在的票，比如0票与-1票，是不存在的。

## 问题分析

线程安全问题都是由全局变量及静态变量引起的。

若每个线程对全局变量、静态变量只读，不写，一般来说，这个变量是线程安全的；

若有多个线程同时执行写操作，一般都需要考虑线程同步，否则的话就可能影响线程安全。



综上所述，线程安全问题根本原因：

* 多个线程在操作共享的数据；
* 操作共享数据的线程代码有多条；（操作共享数据分三段代码执行，先读，然后减一，最后更新）
* 多个线程对共享数据有写操作；

## 问题解决-线程同步

要解决以上线程问题，只要在某个线程修改共享资源的时候，其他线程不能修改该资源，等待修改完毕同步之后，才能去抢夺CPU资源，完成对应的操作，保证了数据的同步性，解决了线程不安全的现象。

为了保证每个线程都能正常执行共享资源操作,Java引入了7种线程同步机制。今天重点介绍前三种，后边的第二天介绍。

1. 同步代码块（synchronized）
2. 同步方法（synchronized）
3. 同步锁（ReenreantLock）
4. 特殊域变量（volatile）
5. 局部变量（ThreadLocal）
6. 阻塞队列（LinkedBlockingQueue）
7. 原子变量（Atomic\*）

### 同步代码块（synchronized）QQQQQQQQQQ

同步代码块 ：

synchronized 关键字可以用于方法中的某个区块中，表示只对这个区块的资源实行互斥访问。

语法:

synchronized(同步锁){

     需要同步操作的代码

}

同步锁:

对象的同步锁只是一个概念,可以想象为在对象上标记了一个锁.

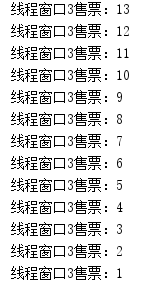
* 锁对象可以是任意类型。
* 多个线程要使用同一把锁。

注意：在任何时候,最多允许一个线程拥有同步锁,谁拿到锁就进入代码块,其他的线程只能在外等着(BLOCKED)。

使用同步代码块代码如下：

**package** com.multithread.thread;  
  
**public class** Ticket **implements** Runnable {  
 **private int ticktNum** = 100;  
  
 *//定义锁对象* Object **obj** = **new** Object();  
  
 **public void** run() {  
 **while**(**true**){  
 **synchronized** (**obj**){  
 **if**(**ticktNum** > 0){  
 *//1.模拟出票时间* **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//2.打印进程号和票号，票数减1* String name = Thread.*currentThread*().getName();  
 System.***out***.println(**"线程"**+name+**"售票："**+**ticktNum**--);  
 }  
 }  
 }  
 }  
}

执行结果如下：线程的安全问题，解决了。



### 同步方法（synchronized）

同步方法：

使用synchronized修饰的方法,就叫做同步方法,保证A线程执行该方法的时候,其他线程只能在方法外等着。

格式：

public synchronized void method(){

   可能会产生线程安全问题的代码

}

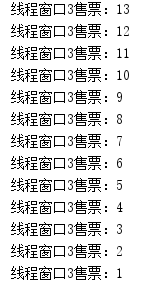
同步锁是谁?

* 对于非static方法,同步锁就是this。
* 对于static方法,同步锁是当前方法所在类的字节码对象(类名.class)。

使用同步方法代码如下：

**package** com.multithread.thread;  
  
**public class** Ticket **implements** Runnable {  
 **private int ticktNum** = 100;  
  
 *//定义锁对象* Object **obj** = **new** Object();  
  
 **public void** run() {  
 **while**(**true**){  
 sellTicket();  
 }  
 }  
  
 **private synchronized void** sellTicket(){  
 **if**(**ticktNum** > 0){  
 *//1.模拟出票时间* **try** {  
 Thread.*sleep*(100);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//2.打印进程号和票号，票数减1* String name = Thread.*currentThread*().getName();  
 System.***out***.println(**"线程"**+name+**"售票："**+**ticktNum**--);  
 }  
 }  
}

执行结果如下：线程的安全问题，解决了。



### 同步锁（ReenreantLock）

同步锁：

java.util.concurrent.locks.Lock 机制提供了比synchronized代码块和synchronized方法更广泛的锁定操作，同步代码块/同步方法具有的功能Lock都有,除此之外更强大,更体现面向对象。

同步锁方法：

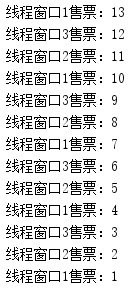
public void lock() :加同步锁。

public void unlock() :释放同步锁。

使用重入锁代码如下：

**package** com.multithread.thread;  
  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class** Ticket **implements** Runnable {  
 **private int ticktNum** = 100;  
  
 *//定义锁对象：构造函数参数为线程是否公平获取锁true-公平；false-不公平，即由某个线程独占，默认是false* Lock **lock** = **new** ReentrantLock(**true**);  
  
 **public void** run() {  
 **while**(**true**){  
 **lock**.lock();  
 **try**{  
 *//加锁* **if**(**ticktNum** > 0){  
 *//1.模拟出票时间* **try** {  
 Thread.*sleep*(50);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 *//2.打印进程号和票号，票数减1* String name = Thread.*currentThread*().getName();  
 System.***out***.println(**"线程"**+name+**"售票："**+**ticktNum**--);  
 }  
 } **finally** {  
 *//放锁* **lock**.unlock();  
 }  
 }  
 }  
}

执行效果如下：



## 小结

Synchronized和Lock区别

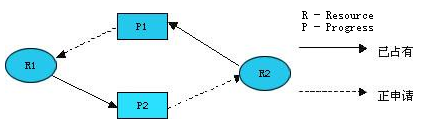
* synchronized是java内置关键字，在jvm层面，Lock是个java类；
* synchronized无法判断是否获取锁的状态，Lock可以判断是否获取到锁；
* synchronized会自动释放锁(a 线程执行完同步代码会释放锁 ；b 线程执行过程中发生异常会释放锁)，Lock需在finally中手工释放锁（unlock()方法释放锁），否则容易造成线程死锁；
* 用synchronized关键字的两个线程1和线程2，如果当前线程1获得锁，线程2线程等待。如果线程1阻塞，线程2则会一直等待下去，而Lock锁就不一定会等待下去，如果尝试获取不到锁，线程可以不用一直等待就结束了；
* synchronized的锁可重入、不可中断、非公平，而Lock锁可重入、可判断、可公平（两者皆可）
* Lock锁适合大量同步的代码的同步问题，synchronized锁适合代码少量的同步问题。

# 线程死锁

## 什么是死锁

多线程以及多进程改善了系统资源的利用率并提高了系统的处理能力。然而，并发执行也带来了新的问题--死锁。

所谓死锁是指多个线程因竞争资源而造成的一种僵局（互相等待），若无外力作用，这些进程都将无法向前推进。



## 死锁产生的必要条件

以下这四个条件是死锁的必要条件，只要系统发生死锁，这些条件必然成立，而只要上述条件之一不满足，就不会发生死锁。

### 互斥条件

进程要求对所分配的资源（如打印机）进行排他性控制，即在一段时间内某资源仅为一个进程所占有。此时若有其他进程请求该资源，则请求进程只能等待。

### 不可剥夺条件

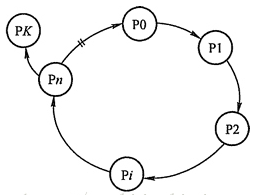
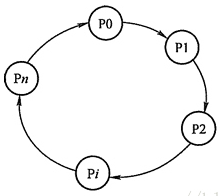
进程所获得的资源在未使用完毕之前，不能被其他进程强行夺走，即只能由获得该资源的进程自己来释放（只能是主动释放)。

### 请求与保持条件

进程已经保持了至少一个资源，但又提出了新的资源请求，而该资源已被其他进程占有，此时请求进程被阻塞，但对自己已获得的资源保持不放。

### 循环等待条件

存在一种进程资源的循环等待链，链中每一个进程已获得的资源同时被 链中下一个进程所请求。即存在一个处于等待状态的进程集合{Pl, P2, …, pn}，其中Pi等 待的资源被P(i+1)占有（i=0, 1, …, n-1)，Pn等待的资源被P0占有，如图所示。



循环等待 满足条件但无死循环

### 死锁示例代码

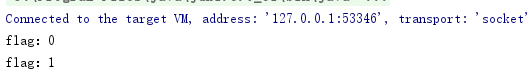
DeadLock.java

**package** com.multithread.thread;  
  
**public class** DeadLock **implements** Runnable {  
 **private static** Object *obj1* = **new** Object();*//定义成静态变量，使线程可以共享实例* **private static** Object *obj2* = **new** Object();*//定义成静态变量，使线程可以共享实例* **public int flag** = 0;  
 **public void** run() {  
 **if**(**flag** == 0){  
 System.***out***.println(**"flag："**+**flag**);  
 **synchronized** (*obj1*){  
 **try** {  
 Thread.*sleep*(500);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **synchronized** (*obj2*){  
 System.***out***.println(**"flag："**+**flag**);  
 }  
 }  
 }  
 **if**(**flag** == 1){  
 System.***out***.println(**"flag："**+**flag**);  
 **synchronized** (*obj2*){  
 **try** {  
 Thread.*sleep*(500);  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 **synchronized** (*obj1*){  
 System.***out***.println(**"flag："**+**flag**);  
 }  
 }  
 }  
 }  
}

DeadLockDemo.java

**package** com.multithread.demos;  
  
**import** com.multithread.thread.DeadLock;  
  
**public class** DeadLockDemo {  
 **public static void** main(String[] args){  
 DeadLock deadLock1 = **new** DeadLock();  
 DeadLock deadLock2 = **new** DeadLock();  
 deadLock2.**flag** = 1;  
  
 Thread thread1 = **new** Thread(deadLock1);  
 Thread thread2 = **new** Thread(deadLock2);  
  
 thread1.start();  
 thread2.start();  
 }  
}

执行效果如下：只打印两个flag值，表示死锁产生



## 死锁处理

* 预防死锁：通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或几个条件，来防止死锁的发生。
* 避免死锁：在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免死锁的发生。
* 检测死锁：允许系统在运行过程中发生死锁，但可设置检测机构及时检测死锁的发生，并采取适当措施加以清除。
* 解除死锁：当检测出死锁后，便采取适当措施将进程从死锁状态中解脱出来。

### 死锁预防

预防死锁是设法至少破坏产生死锁的四个必要条件之一,严格的防止死锁的出现。

#### 破坏“互斥”条件

“互斥”条件是无法破坏的。因此，在死锁预防里主要是破坏其他几个必要条件，而不去涉及破坏“互斥”条件。

#### 破坏“占有并等待”条件

破坏“占有并等待”条件，就是在系统中不允许进程在已获得某种资源的情况下，申请其他资源。即要想出一个办法，阻止进程在持有资源的同时申请其他资源。

* 方法一：一次性分配资源，即创建进程时，要求它申请所需的全部资源，系统或满足其所有要求，或什么也不给它。
* 方法二：要求每个进程提出新的资源申请前，释放它所占有的资源。这样，一个进程在需要资源S时，须先把它先前占有的资源R释放掉，然后才能提出对S的申请，即使它可能很快又要用到资源R。

#### 破坏“不可抢占”条件

破坏“不可抢占”条件就是允许对资源实行抢夺。

* 方法一：如果占有某些资源的一个进程进行进一步资源请求被拒绝，则该进程必须释放它最初占有的资源，如果有必要，可再次请求这些资源和另外的资源。
* 方法二：如果一个进程请求当前被另一个进程占有的一个资源，则操作系统可以抢占另一个进程，要求它释放资源。只有在任意两个进程的优先级都不相同的条件下，方法二才能预防死锁。

#### 破坏“循环等待”条件

破坏“循环等待”条件的一种方法，是将系统中的所有资源统一编号，进程可在任何时刻提出资源申请，但所有申请必须按照资源的编号顺序（升序）提出。这样做就能保证系统不出现死锁。

### 死锁避免

避免死锁不严格限制产生死锁的必要条件的存在,因为即使死锁的必要条件存在,也不一定发生死锁。

#### 有序资源分配法

该算法实现步骤如下：

* 必须为所有资源统一编号，例如打印机为1、传真机为2、磁盘为3等
* 同类资源必须一次申请完，例如打印机和传真机一般为同一个机器，必须同时申请
* 不同类资源必须按顺序申请

例如：有两个进程P1和P2，有两个资源R1和R2

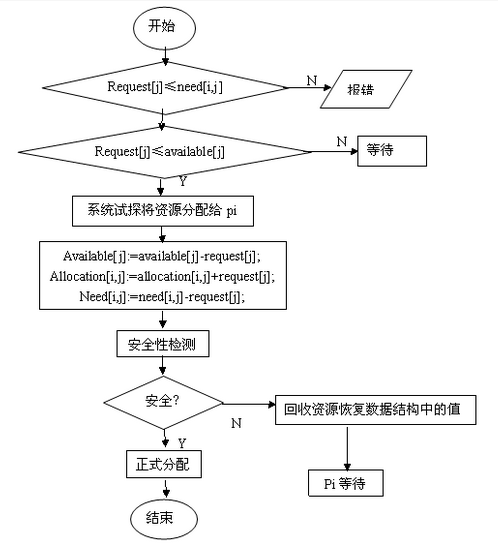
P1请求资源：R1、R2

P2请求资源：R1、R2

这样就破坏了环路条件，避免了死锁的发生。

#### 银行家算法

[银行家算法](https://baike.baidu.com/item/%E9%93%B6%E8%A1%8C%E5%AE%B6%E7%AE%97%E6%B3%95)（Banker's Algorithm）是一个避免死锁（Deadlock）的著名算法，是由艾兹格·迪杰斯特拉在1965年为T.H.E系统设计的一种避免死锁产生的算法。它以银行借贷系统的分配策略为基础，判断并保证系统的安全运行。流程图如下：



银行家算法的基本思想是分配资源之前，判断系统是否是安全的；若是，才分配。它是最具有代表性的避免[死锁](https://baike.baidu.com/item/%E6%AD%BB%E9%94%81)的算法。

设进程i提出请求REQUEST [i]，则银行家算法按如下规则进行判断。

1. 如果REQUEST [i]<= NEED[i，j]，则转（2)；否则，出错。
2. 如果REQUEST [i]<= AVAILABLE[i]，则转（3)；否则，等待。
3. 系统试探分配资源，修改相关数据：

AVAILABLE[i]-=REQUEST[i];//可用资源数-请求资源数

ALLOCATION[i]+=REQUEST[i];//已分配资源数+请求资源数

NEED[i]-=REQUEST[i];//需要资源数-请求资源数

1. 系统执行安全性检查（安全性检查就是银行现有的资源，分给其中一个贷款家，确保这个贷款家用完之后还回来，依次类推，让所有的贷款家都成功，这个存在的序列就是安全序列），如安全，则分配成立；否则试探险性分配作废，系统恢复原状，进程等待。

#### 顺序加锁

当多个线程需要相同的一些锁，但是按照不同的顺序加锁，死锁就很容易发生。

例如以下两个线程就会死锁：

Thread 1:

lock A (when C locked)

lock B (when C locked)

wait for C

Thread 2:

wait for A

wait for B

lock C (when A locked)

如果能确保所有的线程都是按照相同的顺序获得锁，那么死锁就不会发生。 例如以下两个线程就不会死锁

Thread 1:

lock A

lock B

lock C

Thread 2:

wait for A

wait for B

wait for C

按照顺序加锁是一种有效的死锁预防机制。但是，这种方式需要事先知道所有可能会用到的锁，但总有些时候是无法预知的，所以该种方式只适合特定场景。

#### 限时加锁

限时加锁是线程在尝试获取锁的时候加一个超时时间，若超过这个时间则放弃对该锁请求，并回退并释放所有已经获得的锁，然后等待一段随机的时间再重试

以下是一个例子，展示了两个线程以不同的顺序尝试获取相同的两个锁，在发生超时后回退并重试的场景：

Thread 1 locks A

Thread 2 locks B

Thread 1 attempts to lock B but is blocked

Thread 2 attempts to lock A but is blocked

Thread 1’s lock attempt on B times out

Thread 1 backs up and releases A as well

Thread 1 waits randomly (e.g. 257 millis) before retrying.

Thread 2’s lock attempt on A times out

Thread 2 backs up and releases B as well

Thread 2 waits randomly (e.g. 43 millis) before retrying.

在上面的例子中，线程2比线程1早200毫秒进行重试加锁，因此它可以先成功地获取到两个锁。这时，线程1尝试获取锁A并且处于等待状态。当线程2结束时，线程1也可以顺利的获得这两个锁。

这种方式有两个缺点：

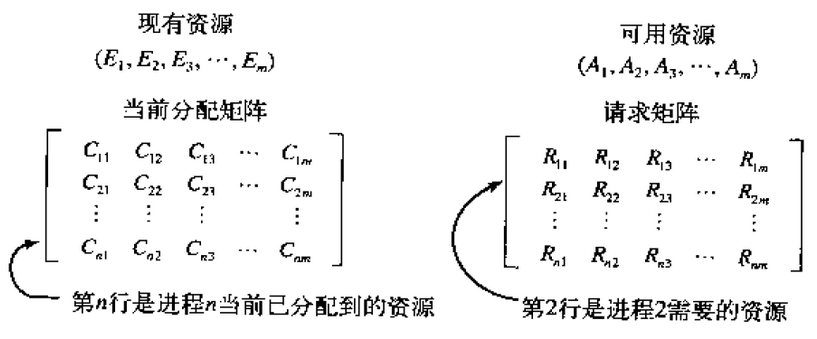
1. 当线程数量少时，该种方式可避免死锁，但当线程数量过多，这些线程的加锁时限相同的概率就高很多，可能会导致超时后重试的死循环。
2. Java中不能对synchronized同步块设置超时时间。你需要创建一个自定义锁，或使用Java5中java.util.concurrent包下的工具。

### 死锁检测

预防和避免死锁系统开销大且不能充分利用资源，更好的方法是不采取任何限制性措施，而是提供检测和解脱死锁的手段，这就是死锁检测和恢复。

死锁检测数据结构：

* E是现有资源向量（existing resource vector），代码每种已存在资源的总数
* A是可用资源向量（available resource vector），那么Ai表示当前可供使用的资源数（即没有被分配的资源）
* C是当前分配矩阵（current allocation matrix），C的第i行代表Pi当前所持有的每一种类型资源的资源数
* R是请求矩阵（request matrix），R的每一行代表P所需要的资源的数量



死锁检测步骤：

1. 寻找一个没有结束标记的进程Pi，对于它而言R矩阵的第i行向量小于或等于A。
2. 如果找到了这样一个进程，执行该进程，然后将C矩阵的第i行向量加到A中，标记该进程，并转到第1步
3. 如果没有这样的进程，那么算法终止
4. 算法结束时，所有没有标记过的进程都是死锁进程。

### 死锁恢复

利用抢占恢复。

临时将某个资源从它的当前所属进程转移到另一个进程。

这种做法很可能需要人工干预，主要做法是否可行需取决于资源本身的特性。

利用回滚恢复

周期性的将进程的状态进行备份，当发现进程死锁后，根据备份将该进程复位到一个更早的，还没有取得所需的资源的状态，接着就把这些资源分配给其他死锁进程。

通过杀死进程恢复

最直接简单的方式就是杀死一个或若干个进程。

尽可能保证杀死的进程可以从头再来而不带来副作用。

# 线程通讯

## 为什么要线程通信

多个线程并发执行时，在默认情况下CPU是随机切换线程的，有时我们希望CPU按我们的规律执行线程，此时就需要线程之间协调通信。

## 线程通讯方式

线程间通信常用方式如下：

* 休眠唤醒方式：

Object的wait、notify、notifyAll

Condition的await、signal、signalAll

* CountDownLatch：用于某个线程A等待若干个其他线程执行完之后，它才执行
* CyclicBarrier：一组线程等待至某个状态之后再全部同时执行
* Semaphore：用于控制对某组资源的访问权限

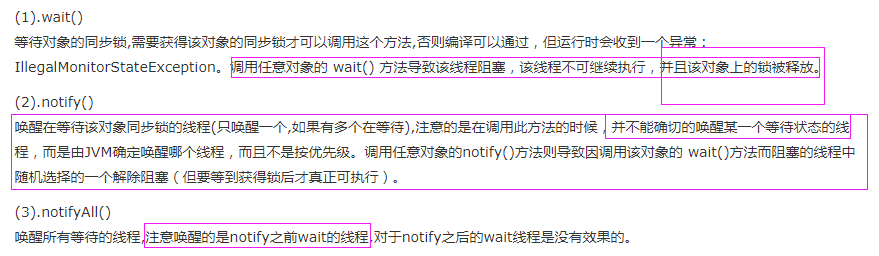
### 休眠唤醒方式

多线程打印10以内的奇偶数

i从0开始，当时奇数时，奇数线程打印，偶数线程等待，当是偶数时，偶数线程打印，奇数线程等待。

在调用一个Object的wait与notify/notifyAll的时候，必须保证调用代码对该Object是同步的，也就是说必须在作用等同于synchronized(object){......}的内部才能够去调用obj的wait与notify/notifyAll三个方法，否则就会报错：java.lang.IllegalMonitorStateException:current thread not owner(意思是因为没有同步，所以线程对对象锁的状态是不确定的，不能调用这些方法)。

Object的wait、notify、notifyAll



**package** com.multithread.thread;  
  
**public class** WaitNotifyRunnable{

//使用object的唤醒和等待方法  
 **private** Object **obj** = **new** Object();  
 **private** Integer **i**=0;  
 **public void** odd() {  
 **while**(**i**<10){  
 **synchronized** (**obj**){  
 **if**(**i**%2 == 1){  
 System.***out***.println(**"奇数："**+**i**);  
 **i**++;  
 **obj**.notify();//唤醒偶数线程打印  
 } **else** {  
 **try** {  
 **obj**.wait();//等待偶数线程打印  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 }  
  
 **public void** even(){  
 **while**(**i**<10){  
 **synchronized** (**obj**){  
 **if**(**i**%2 == 0){  
 System.***out***.println(**"偶数："**+**i**);  
 **i**++;  
 **obj**.notify();  
 } **else** {  
 **try** {  
 **obj**.wait();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 }  
 **public static void** main(String[] args){  
 **final** WaitNotifyRunnable runnable = **new** WaitNotifyRunnable();  
 Thread t1 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 runnable.odd();  
 }  
 }, **"偶数线程"**);  
 Thread t2 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 runnable.even();  
 }  
 }, **"奇数线程"**);  
  
 t1.start();  
 t2.start();  
 }  
}

Condition的await、signal、signalAll

**package** com.multithread.thread;  
  
**import** java.util.concurrent.locks.Condition;  
**import** java.util.concurrent.locks.Lock;  
**import** java.util.concurrent.locks.ReentrantLock;  
  
**public class** WaitNotifyRunnable{  
 **private** Lock **lock** = **new** ReentrantLock();  
 **private** Condition **condition** = **lock**.newCondition();  
 **private** Integer **i**=0;  
 **public void** odd() {  
 **while**(**i**<10){  
 **lock**.lock();  
 **try**{  
 **if**(**i**%2 == 1){  
 System.***out***.println(**"奇数："**+**i**);  
 **i**++;  
 **condition**.signal();  
 } **else** {  
 **condition**.await();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
  
 }  
 }  
  
 **public void** even(){  
 **while**(**i**<10){  
 **lock**.lock();  
 **try**{  
 **if**(**i**%2 == 0){  
 System.***out***.println(**"偶数："**+**i**);  
 **i**++;  
 **condition**.signal();  
 } **else** {  
 **condition**.await();  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **finally** {  
 **lock**.unlock();  
 }  
  
 }  
 }  
 **public static void** main(String[] args){  
 **final** WaitNotifyRunnable runnable = **new** WaitNotifyRunnable();  
 Thread t1 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 runnable.odd();  
 }  
 }, **"偶数线程"**);  
 Thread t2 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 runnable.even();  
 }  
 }, **"奇数线程"**);  
  
 t1.start();  
 t2.start();  
 }  
}

Object和Condition休眠唤醒区别

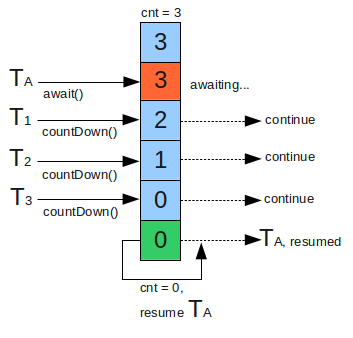
* object wait()必须在synchronized（同步锁）下使用，
* object wait()必须要通过Nodify()方法进行唤醒
* condition await() 必须和Lock（互斥锁/共享锁）配合使用
* condition await() 必须通过 signal() 方法进行唤醒

### CountDownLatch方式

CountDownLatch是在java1.5被引入的，存在于java.util.concurrent包下。

CountDownLatch这个类能够使一个线程等待其他线程完成各自的工作后再执行。

CountDownLatch是通过一个计数器来实现的，计数器的初始值为线程的数量。

[](http://www.importnew.com/15731.html/countdownlatch_example)

每当一个线程完成了自己的任务后，计数器的值就会减1。当计数器值到达0时，它表示所有的线程已经完成了任务，然后在闭锁上等待的线程就可以恢复执行任务。

示例代码：

**package** com.multithread.thread;  
  
**import** java.util.concurrent.CountDownLatch;  
  
**public class** CountDown {  
 **private** Integer **i** = 0;  
 **private** CountDownLatch **countDownLatch** = **new** CountDownLatch(1);  
  
 **public void** odd(){  
 **while**(**i** < 10){  
 **if**(**i**%2 == 1){  
 System.***out***.println(**"奇数："**+**i**);  
 **i**++;  
 **countDownLatch**.countDown();  
 } **else** {  
 **try** {  
 **countDownLatch**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
  
 **public void** even(){  
 **while**(**i** < 10){  
 **if**(**i**%2 == 0){  
 System.***out***.println(**"偶数："**+**i**);  
 **i**++;  
 **countDownLatch**.countDown();  
 } **else** {  
 **try** {  
 **countDownLatch**.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
 **public static void** main(String[] args){  
 **final** CountDown countDown = **new** CountDown();  
 Thread t1 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 countDown.odd();  
 }  
 },**"奇数"**);  
 Thread t2 = **new** Thread(**new** Runnable() {  
 **public void** run() {  
 countDown.even();  
 }  
 },**"偶数"**);  
  
 t1.start();  
 t2.start();  
 }  
}

### CyclicBarrier方式

CyclicBarrier是在java1.5被引入的，存在于java.util.concurrent包下。

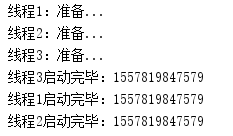
CyclicBarrier实现让一组线程等待至某个状态之后再全部同时执行。

CyclicBarrier底层是

三个线程同时启动，示例代码如下：

**package** com.multithread.thread;  
  
**import** java.util.Date;  
**import** java.util.concurrent.BrokenBarrierException;  
**import** java.util.concurrent.CyclicBarrier;  
  
**public class** CyclicBarrierDemo {  
 **public static void** main(String[] args){  
 **final** CyclicBarrier cyclicBarrier = **new** CyclicBarrier(3);  
  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName()+**"：准备..."**);  
 **try** {  
 cyclicBarrier.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName()+**"启动完毕："**+**new** Date().getTime());  
 }  
 },**"线程1"**).start();  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName()+**"：准备..."**);  
 **try** {  
 cyclicBarrier.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName()+**"启动完毕："**+**new** Date().getTime());  
 }  
 },**"线程2"**).start();  
 **new** Thread(**new** Runnable() {  
 **public void** run() {  
 System.***out***.println(Thread.*currentThread*().getName()+**"：准备..."**);  
 **try** {  
 cyclicBarrier.await();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 } **catch** (BrokenBarrierException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(Thread.*currentThread*().getName()+**"启动完毕："**+**new** Date().getTime());  
 }  
 },**"线程3"**).start();  
 }  
}

执行效果如下：三个线程同时启动



### Semaphore方式

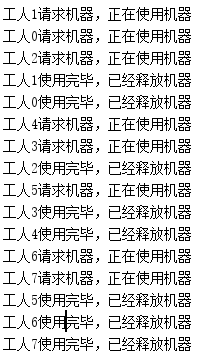
Semaphore是在java1.5被引入的，存在于java.util.concurrent包下。

Semaphore用于控制对某组资源的访问权限。

工人使用机器工作，示例代码如下：

**package** com.multithread.thread;  
  
**import** java.util.concurrent.Semaphore;  
  
**public class** SemaphoreDemo {  
  
 **static class** Machine **implements** Runnable{  
 **private int num**;  
 **private** Semaphore **semaphore**;  
  
 **public** Machine(**int** num, Semaphore semaphore) {  
 **this**.**num** = num;  
 **this**.**semaphore** = semaphore;  
 }  
  
 **public void** run() {  
 **try** {  
 **semaphore**.acquire();*//请求机器* System.***out***.println(**"工人"**+**this**.**num**+**"请求机器，正在使用机器"**);  
 Thread.*sleep*(1000);  
 System.***out***.println(**"工人"**+**this**.**num**+**"使用完毕，已经释放机器"**);  
 **semaphore**.release();*//释放机器* } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
  
 **public static void** main(String[] args){  
 **int** worker = 8;*//工人数* Semaphore semaphore = **new** Semaphore(3);*//机器数* **for** (**int** i=0; i< worker; i++){  
 **new** Thread(**new** Machine(i, semaphore)).start();  
 }  
 }  
}

执行效果如下：



## 小结

### sleep和wait区别



### wait和notify区别

wait和notify都是Object中的方法

wait和notify执行前线程都必须获得对象锁

wait的作用是使当前线程进行等待

notify的作用是通知其他等待当前线程的对象锁的线程